CHAPTER 1

ニューラルネットワークを用いた手書き文字認識

人間の視覚がどんなに不思議なものか、考えたことはありますか？次の手書き数列を読んでみてください：

大抵の人にとってはごく簡単に504192と読めると思います。でも、脳の中で起 こっていることは簡単どころではありません。脳のふたつの半球にはそれぞれ、一次視覚野---V1とも呼ばれる、一億四千万のニューロンと何十億ものシナ プスからなる領域が存在しています。さらに、人間の視覚に関わっている領域 はV1だけではなく、V1、V2、V3、V4、V5という一連の視覚野が、順次複雑な画 像処理に携わっています。私たちの頭部には、数億年にわたる進化によって洗練され、視覚世界を理解するのに驚くべき適応をとげたスーパーコンピューターが内蔵されているのです。手書き数字認識が簡単なのではありません。どちらかというと、私たち人類が、目に見えるものを解釈するという作業をとても、とても得意としているのです。しかもその作業はほとんど無意識のうちに行われるのです。ですから、私たちは普段、自分の視覚系がいかに複雑な問題を解いてくれているかに、感謝を払うこともないのです。

ひとたび、さっきの手書き数字を認識するプログラムを書こうとすれば、視覚パターン認識の困難さが明らかになります。自分でやればこんなに簡単に思えることが、突然ものすごく難しくなったように感じるでしょう。数字を認識するための直感的で単純なルール---「数字の9は、上に輪があって、右下から下に向かって線が生えている形」---をアルゴリズムで表現するのはけっして単純ではないことに気づくでしょう。このようなルールを正確にプログラムとして表現しようとすれば、すぐに膨大な例外、落とし穴、特殊ケースに気づくはずです。絶望的です。

ニューラルネットワークはこのような問題に違った角度から迫ります。ニューラルネットワークの発想は、手書き数字のデータをあらかじめ沢山用意して(このようなデータを訓練例といいます)

この本は目下のところベータ版で、開発続行中です。エラーレポートは mn@michaelnielsen.org まで、日本語版に関する質問は muranushi@gmail.com までお送りください。その他の質問については、まずはFAQをごらんください。

Resources

Code repository

Mailing list for book announcements

Michael Nielsen's project announcement mailing list

著:Michael Nielsen / 2014年9月-12月
訳:「ニューラルネットワークと深層学習」翻訳プロジェクト

その上で、訓練例から学習することのできるシステムを開発する、というものです。言い換えれば、ニューラルネットワークは、訓練例をもとに、数字認識のルールを自動的に推論します。さらに、訓練例を増やすほど、ニューラルネットワークは手書き文字に関する知識をより多く獲得し、精度が向上します。上図ではわずか100個の訓練例を示しましたが、何千、何万、何億個という訓練例を与えることで、よりよい手書き数字認識機を作ることができるかもしれません。

この章では、手書き数字認識を学習するニューラルネットワークを実装することを目標とします。実装するプログラムはたったの74行に収まり、しかも特別なニューラルネットワークライブラリを使うわけではありません。それでも、この短いプログラムは人手の介入なしに、96%以上の精度で数字を認識することができます。2章以降で導入する新しいアイデアを組み込めば、この性能はさらに99%を上回るまで向上します。実は、現在最高レベルの商用ニューラルネットワークは、銀行での小切手の処理や郵便局での住所認識に使われるほどの高い性能に達しています。

手書き文字認識は、ニューラルネットワーク一般について解説するうえでうってつけの題材なので、まずは手書き文字認識に話を絞ることにします。というのも、手書き文字認識というのは、決して一筋縄ではゆかない歯ごたえのある課題です。それでいて例えば極めて複雑な解法を必要とするとか、莫大な計算資源を必要とするとかの、非常に困難な課題というわけでもなく、ちょうどいい難易度の課題なのです。さらに、手書き文字認識は、深層学習といった発展的な技術の題材としても適しています。というわけで、この本では繰り返し、手書き文字認識という課題に立ち戻ることにします。この本の後のほうでは、これらのアイデアのコンピュータ視覚、音声、自然言語処理、その他の分野への応用をあつかいます。

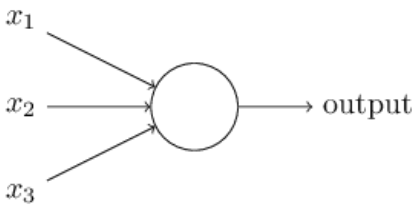
もちろん、この章の目的がただ手書き数字を認識するプログラムを書くことだけだったなら、この章はもっと短くなったでしょう！しかし、この章で

は、手書き文字認識を実装する過程で、ニューラルネットワークの鍵となるアイデアをいくつも開発します。その中には、二種類の重要な人工ニューロン(パーセプトロンと、シグモイドニューロン)や、ニューラルネットワークの標準的な学習アルゴリズムである確率的勾配降下法が含まれます。本書を通じて、私は現行の手法を紹介するだけでなく*なぜ*その手法が選ばれたのかについて解説することで、あなたのニューラルネットワークにまつわる直感を鍛えてゆければと思います。おかげで、本章は流行りのトピックをただ並べた解説などよりはもずいぶん長くなってしまいますが、あなたがより深い理解に達することを思えばその価値はあると思います。とりわけ、本章を読み終わるころには、私たちは、深層学習とは何なのか、なぜ重要なのか、の理解に到達するでしょう。

パーセプトロン

ニューラルネットワークとは何か、という解説を始めるにあたり、まずは**パーセプトロン**と呼ばれる種類の人工ニューロンから話を始めたいと思います。パーセプトロンは、1950年代から1960年代にかけて、Warren McCullochとWalter Pittsらの 先行研究に触発されたFrank Rosenblattによって 開発されました。今日では、パーセプトロン以外の人工ニューロンモデルを扱うことが一般的です。この本では、そして現代のニューラルネットワーク研究の多くでは、**シグモイドニューロン**と呼ばれるモデルが主に使われています。この本でも、もうすぐシグモイドニューロンが登場します。しかし、なぜシグモイドニューロンが今の姿をしているのか知るためにも、まずはパーセプトロンを理解することに時間をさく価値があると言えるでしょう。

さて、パーセプトロンとは何でしょうか？パーセプトロンは複数の二進数(0または1) $x_1, x_2, \ldots$ を入力にとり、ひとつの二進数(0または1)を出力します。



上図の例では、パーセプトロンは三つの入力 x_1, x_2, x_3 をとっています。一般的には、入力はいくつでも構いません。ローゼンブラット(カタカナ表記は正しい？)は、出力を計算する簡単なルールを提案しました。彼は**重み**, $w_1, w_2, \ldots$という概念を導入しました。重みとは、それぞれの入力が出力に及ぼす影響の大きさを表す実数です。パーセプトロンの出力が0になるか1になるかは、入力の重みつき和 $\sum_j w_j x_j$ と**閾値**の大小

比較で決まります。重みと同じく、閾値もパーセプトロンの挙動を決める実数パラメータです。より正確に、数式で表現するなら、

$$\text{output} = \begin{cases} 0 & \text{if } \sum_j w_j x_j \leq \text{ threshold} \\ 1 & \text{if } \sum_j w_j x_j > \text{ threshold} \end{cases} \tag{1}$$

パーセプトロンを動かすルールは、たったこれだけです！

まずは基礎的となる数学モデルをご紹介しましたが、直感的にいえば、パーセプトロンとは、複数の情報に、重みをつけながら決定をくだす機械だと言えます。例を出しましょう。今から出すのは簡単な例ですが、あまり現実的な例ではありません。すぐに、もっと現実的な例が出てきます。

週末が近づいているとしましょう。週末には、あなたの住んでいる街で「チーズ祭り」が催されるそうです。あなたはチーズが好物で、チーズ祭りに行くかどうか決めようとしています。あなたの判断に影響を及ぼしそうなファクターは、三つあります。

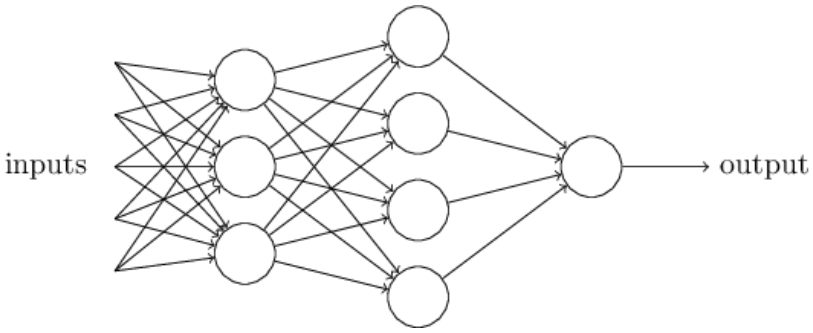
1. 天気はいいか？
2. あなたの恋人も一緒に行きたがっているか？
3. 祭りの会場は駅から近いか？（あなたは自家用車を持っていません。）

これらの三つのファクターは、対応する二進数値$x_1, x_2\ x_3$で表現することができます。例えば、天気が良いなら$x_1 = 1$、天気が悪いなら$x_1 = 0$と決めましょう。同じく、$x_2 = 1$ならあなたの恋人は行きたがっており、$x_2 = 0$ならそうではありません。x_3と駅も同様です。

さて、あなたはチーズが大好物で、あなたの大切な人が何と言おうが、会場が駅から遠かろうが、喜んでチーズ祭りに行くつもりだとしましょう。いっぽう、あなたは雨が何より苦手で、もし天気が悪かったら絶対に行くつもりはありません。パーセプトロンは、このような意思決定を表現することができます。一つの方法は、天気の条件の重みを $w_1 = 6$、他の重みを $w_2 = 2$ と $w_3 = 2$ にすることです。w_1の値が大きいことは、あなたにとって天気がとても重要であること---恋人の意思や駅からの距離よりもずっとずっと重要であることを表しています。最後に、パーセプトロンの閾値を 5 に設定します。以上のパラメータ設定により、パーセプトロンであなたの意思決定モデルを実装できました。このパーセプトロンは、天気が良ければ必ず1を出力し、天気が悪ければ必ず0を出力します。あなたの恋人の意思や、駅からの距離によって結論が変わることはありません。

重みと閾値とを変化させることで、様々に異なった意思決定モデルを得ることができます。たとえば、閾値を5から3に変えましょうか。すると、パーセプトロンが「祭りにいくべき」と判断する条件は「天気が良い」**または**「会場が駅から近く、**かつ**あなたの恋人が一緒に行きたがっている」となります。つまり、意思決定モデルが変化したのです。閾値を下げることは、あなたが祭りに行きたがっていることを意味します。

もちろん、パーセプトロンは人間の意思決定モデルの完全なモデルというわけではありません！ とはいえ、パーセプトロンは異なる種類の情報を考慮し、重みをつけたうえで判断を下す能力があることを、先ほどの例は示しています。となれば、パーセプトロンを複雑に組み合わせたネットワークなら、かなり微妙な判断も扱えそうです：



上図のネットワークでは、まず一列目の三つのパーセプトロン - 第一層のパーセプトロンと呼ぶことにしましょう - が、入力情報に重みをつけて、とても単純な判断を行っています。それでは、第二層のパーセプトロンは何をしているのでしょう？これらのパーセプトロンは、第一層のパーセプトロンの下した判断に重みをつけることで、判断を下しています。これら第二層のパーセプトロンは、第一層のパーセプトロンよりも複雑で、抽象的な判断を下しているといえそうです。第三層のパーセプトロンは、さらに複雑な判断を行っています。このように、多層のニューラルネットワークは高度な意思決定を行うことができるのです。

先ほどパーセプトロンを定義した時には、パーセプトロンは出力をひとつしか持たないと言いました。ところが上図のネットワークの中のパーセプトロンは、複数の出力を持つように描かれていますね。でも、あくまでもパーセプトロンの出力はひとつなんです。出力の矢印が複数あるのは、ただ、あるパーセプトロンの出力が複数のパーセプトロンへの入力として使われることを示しているにすぎません。こうすれば、一つの出力矢印を描いてから分岐させるよりも、若干見やすくなりますからね。

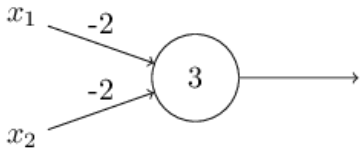
パーセプトロンの記法をもっと簡潔にしましょう。パーセプトロンが1を出力する条件式、$\sum_j w_j x_j > \text{threshold}$ は何だか煩雑です。そこで、これをもっと簡単に書ける記法を導入することにします。まず、$\sum_j w_j x_j$ という和は内積を使って、$w \cdot x \equiv \sum_j w_j x_j$と書くことにします。ここで、$w$と

xはそれぞれ重みと入力を要素にもつベクトルです。次に、閾値を不等式の左辺に移項し、パーセプトロンの**バイアス** $b \equiv -\text{threshold}$と呼ばれる量に置き換えます。閾値の代わりにバイアスを使うと、パーセプトロンのルールはこのように書き換えられます:

$$\text{output} = \begin{cases} 0 & \text{if } w \cdot x + b \leq 0 \\ 1 & \text{if } w \cdot x + b > 0 \end{cases} \tag{2}$$

バイアスは、パーセプトロンが1を出力する傾向の高さを表す量だとみなすことができます。あるいは、生物学の例えを使えば、バイアスとは、パーセプトロンというニューロンが**発火** する傾向の高さを表すといえます。もし、あるパーセプトロンのバイアスがとても大きければ、パーセプトロンが1を出力するのはとても簡単なことでしょう。逆に、パーセプトロンのバイアスが負の数なら、1を出力させるのは骨が折れそうです。見てのとおり、閾値の代わりにバイアスを使うのは、パーセプトロンの表記をほんの少し変更するにすぎません。しかし、バイアスを使ったほうがもっとシンプルになる場合がのちほど出てきます。というわけで、この本では今後、閾値ではなくバイアスを使うことにします。

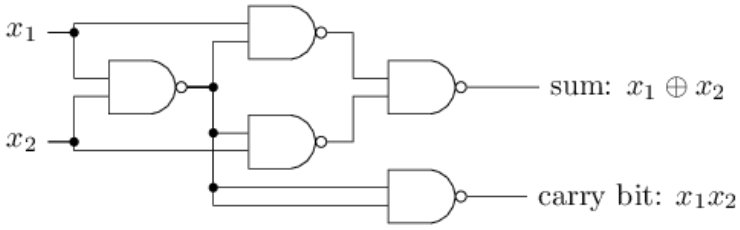
ここまでの解説では、パーセプトロンを入力情報に重みをつけて判断を行う手続きとして用いてきました。パーセプトロンには他の用途もあります。それは、論理関数を計算することです。あらゆる計算は、AND、OR、そしてNANDといった基本的な論理関数から構成されている、とみなすことができます。パーセプトロンは、こういった論理関数を表現できるのです。例えば、二つの入力をとり、どちらも重みが-2で、全体のバイアスが3であるようなパーセプトロンを考えてみましょう。図にすると、こうなります:



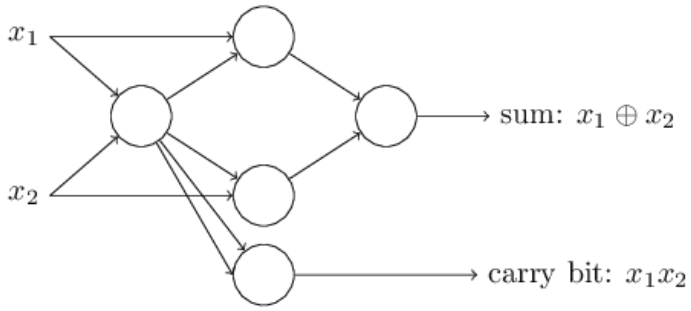
このパーセプトロンは、00 を入力されると 1 を出力することがわかります。なぜなら、$(-2) * 0 + (-2) * 0 + 3 = 3$ は正の数だからです。(紛らわしくないように、掛け算を記号 $*$ で表しました。) 同じように計算すると、このパーセプトロンは 01 や 10 を入力してもやっぱり1 を出力することがわかります。ところが、11 を入力した場合だけは 0 が出力されます。なぜなら $(-2) * 1 + (-2) * 1 + 3 = -1$ は正の数ではないからです。ということは、このパーセプトロンはNANDゲートを実装していることになります！

NAND ゲートの例から、パーセプトロンが単純な論理関数を計算できることが分かります。それどころか、パーセプトロンのネットワークさえあれば**任意の**論理関数を計算できることまで分かるのです。なぜなら NAND ゲートは論理計算において万能だからです。万能だ、とは、NAND ゲートさえあ

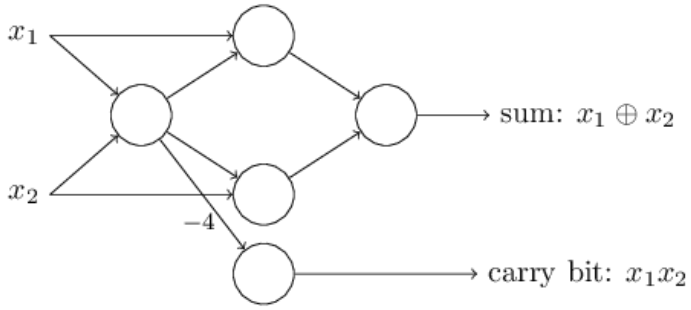
ればどんな計算でも構成できる、という意味です。たとえば、NAND ゲートを使って1ビットの二進数どうしを加算する回路を作ることができます。入力の二進数を x_1 と x_2 としましょう。これらの和を表現するには二進数で二桁が必要です。一桁目は入力の排他的論理和 $x_1 \oplus x_2$ になります。二桁目は x_1 と x_2 がともに 1の場合だけ 1 になる繰り上がりビットです。繰り上がりビットは、ただの論理積 x_1 AND x_2 である、ともいえます。



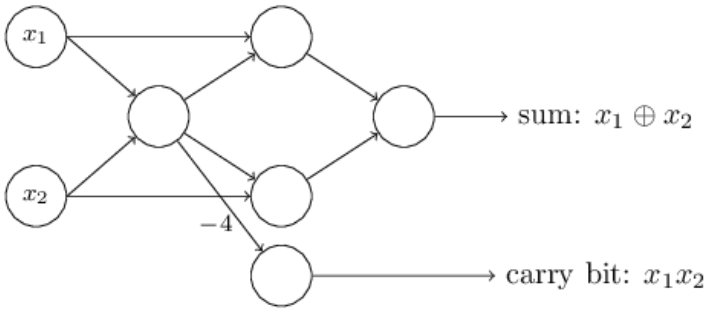
この論理回路と等価なパーセプトロンを得るには、回路内の NANDゲートをすべて、重み -2の入力を二つ持ち、バイアスが3であるパーセプトロンに置き換えます。この置き換えを施すと、以下のようなネットワークができます。ただし、右下にあったNANDゲートに対応するパーセプトロンだけは、矢印が見やすいように少し動かしてあります:



このニューラルネットワークの中でひとつ注目すべき点は、一番左のパーセプトロンからの出力が一番下のパーセプトロンの入力として二度使われている点です。パーセプトロンの定義を与えたとき、このような、同じ箇所に同一の出力が二回入力される場合が許されるのか否かについては言及しませんでした。実のところ、このような重複入力を許すかどうかは問題になりません。仮に、重複入力を許さないことにしたとしても、二つの入力をくっつけて、重みが-2である入力をふたつ用いるかわりに、重みが-4の入力をひとつ使えばいいのです。(もし、あなたがこれを自明に思えないなら、ここで立ち止まって、等価性を自分で証明してみるべきです。) この変更によって、ニューラルネットワークは以下のようになります。ここで、重みが書いてない矢印の重みはすべて-2で、すべてのバイアスは3で、ひとつだけ重みが書いてある矢印の重みは-4です。



ここまで、x_1 や x_2 といった入力はパーセプトロンネットワークの左に浮いている変数として描いてきました。実は、入力を表現するには、**入力層**と呼ばれる追加の層を設けるやり方が標準的です。



このような、入力がなく、出力がひとつしかない記法は、

入力パーセプトロンを表す省略記法です。本気で入力をもたないパーセプトロンを意味しているわけではありません。このことを見るには、入力をもたないパーセプトロンが本当にあったとしましょう。すると、重み付き和 $\sum_j w_j x_j$ は常に0ですから、そのようなパーセプトロンは $b > 0$であれば常時1を、$b \leq 0$であれば常時0を出力することになります。つまり、そのようなパーセプトロンは単に定数を出力するだけで、望みの値 (上図の例ではx_1)を出力するものではないことがわかります。入力パーセプトロンは、実際まったくパーセプトロンではなく、望みの値$x_1, x_2, \ldots$を出力するよう定義された特殊ユニットであると考えたほうがよいのです。

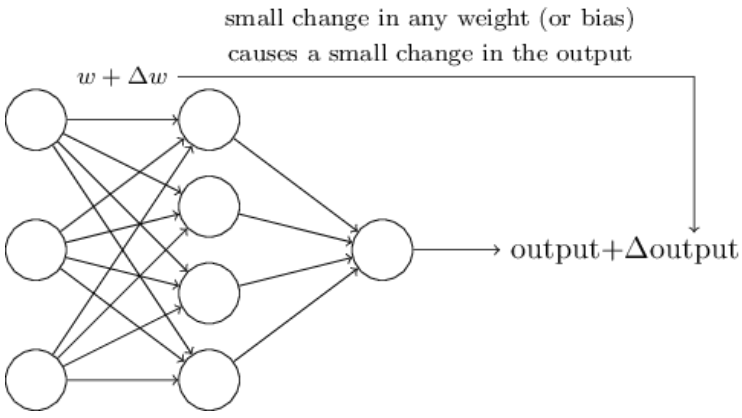
加算機の例は、パーセプトロンのネットワークがNANDゲートを多数含む回路をシミュレートすることに使える、ということの実例でした。そして、NANDゲートの万能性(それさえあればどんな関数でも計算できるという性質)から、パーセプトロンもまた万能である、ということが導けます。

パーセプトロンが計算論的万能性を持つということは、心強いと同時に残念な事実です。心強い、というのは、パーセプトロンが他のどの計算装置にも負けない強力さを持つことを、この事実は教えてくれるからです。残念だ、というのは、パーセプトロンはNANDゲートの亜種、四角い車輪の再発明に過ぎない、と感じられるからです。これでは、とても大したニュースとはいえません！

ところが、現実はそれほど残念ではないのです。なぜなら我々は、ニューラルネットワークの重みとバイアスを自動的に最適化するような、**学習アルゴリズム**を開発することができるからです。この最適化は、プログラマの直接介入なしに、外部刺激に反応して勝手に起こるものです。これらの学習アルゴリズムのおかげで、人工ニューロンは、従来の論理ゲートとは全く異なった使い方ができます。NANDゲートや他の種類の論理ゲートはすべて手動で配線してやる必要があったのに対し、ニューラルネットワークは問題の解き方を自発的に学習してくれます。ときには、従来型の回路を設計するのが非常に難しいような問題に対してさえも。

シグモイドニューロン

学習アルゴリズムとは大変すばらしい。でも、ニューラルネットワークに対してそのようなアルゴリズムをどう設計すればいいのでしょう？ かりに、私たちがある種の問題をパーセプトロンのネットワークを使って解こうとしている、としましょう。例えば、入力は手書き文字のスキャン画像の生ピクセルデータである、とか。そして、ニューラルネットワークには、数字を正しく分類できるよう、重みとバイアスを学習してほしいわけです。学習がどのように働くのか知るために、ネットワークの中のいくつかの重みやバイアスを少しだけ変更するとしましょう。私たちとしては、このような小さな変更に対応する、ニューラルネットワークからの出力の変化もまた小さなものであってほしいわけです。まもなく出てきますが、この性質こそが学習を可能にするのです。図示すれば、こんな感じです（もちろん、図のニューラルネットワークは手書き文字認識をするには小規模すぎます！）



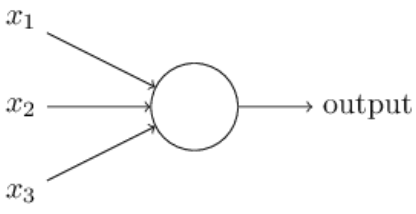
もし、重みやバイアスを微小に変化させた場合の出力の変化もまた微小である、という性質が本当に成り立っていれば、その性質をつかって、ニューラルネットワークがより自分の思ったとおりの挙動を示すように重みとバイアスを修正できます。たとえば、ニューラルネットワークがある「9」であるべき手書き文字を、間違って「8」に分類したとします。私たちは重みやバイアスに小さな変化を与えて、どうすればこのニューラルネットワークが

この画像を正しく「9」と分類する方向に近づくか探ることができます。この過程を繰り返し、重みとバイアスを変化させ続ければ、生成される結果は次第に改善されてゆくでしょう。ニューラルネットワークはこうして学習するのです。

問題は、ニューラルネットワークがパーセプトロンで構成されていたとすると、このような学習は起こらない、ということです。実際、ニューラルネットワーク内のパーセプトロンのうち、どれか1つの重みやバイアスを少し変えてやると、そのパーセプトロンの出力は、変化がないか、もしくは0から1へというようにすっかり反転してしまいます。このように出力が反転すれば、ニューラルネットワーク内の他の部分の挙動も、連動して複雑に変わっていってしまいます。つまり、先程の手書き文字の「9」を、なんとか正しく数字の「9」に分類させることができたとしても、今度は「9」以外の全ての手書き文字に対するニューラルネットワークの挙動までもが完全に変わってしまい、その変化をコントロールすることは困難となります。もしかしたら、この問題を回避することのできる何らかの賢い方法があるかもしれませんが、今のところ、パーセプトロンで構成されたニューラルネットワークに上手に学習させる方法は明らかになっていません。

この問題は、**シグモイド**ニューロンと呼ばれる、新しいタイプの人工ニューロンを導入することによって克服することができます。シグモイドニューロンはパーセプトロンと似ていますが、シグモイドニューロンの重みやバイアスに微小な変化を与えたとき、それに応じて生じる出力の変化も微小なものに留まるように調整されています。このことは、シグモイドニューロンで構成されているニューラルネットワークの学習を可能にする、決定的な違いとなります。

よし、それではシグモイドニューロンをご説明しましょう。シグモイドニューロンは、パーセプトロンと同じような見た目で描くことにします：



ちょうどパーセプトロンがそうであるように、シグモイドニューロンは$x_1, x_2, \ldots$といった入力を取ります。しかし、これらの入力値は、単に0や1だけではなく、0から1**の間**のあらゆる値をとることができます。そのため、たとえば、$0.638\ldots$といった値も、シグモイドニューロンにとっては有効な入力値となります。パーセプトロンがそうであるように、シグモイドニューロンはそれぞれの入力に対して、重み$(w_1, w_2, \ldots)$を持ち、またニューロン全体に対するバイアスと呼ばれる値(b)を持っています。しかし、出力は、

0や1だけではありません。代わりに、出力としては$\sigma(w \cdot x + b)$という値をとります。σは**シグモイド関数*** と呼ばれており、次の式で定義されます：

$$\sigma(z) \equiv \frac{1}{1+e^{-z}}. \tag{3}$$

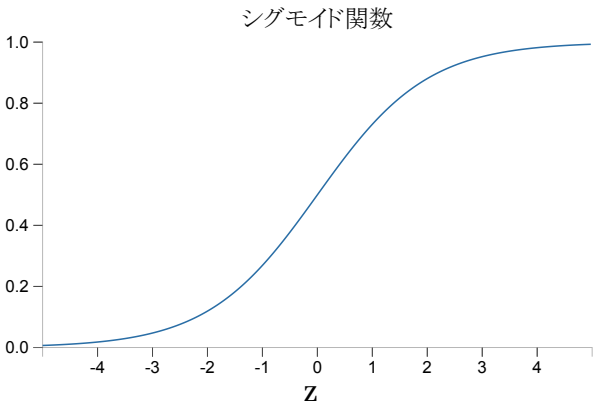
より明確に表現すると、シグモイドニューロンの出力は、入力が$x_1, x_2, \ldots$で、重みが$w_1, w_2, \ldots$で、そしてバイアスがbのとき、次の形を取ります。

$$\frac{1}{1+\exp(-\sum_j w_j x_j - b)}. \tag{4}$$

一見すると、シグモイドニューロンはパーセプトロンとは大きく異なるように見えます。シグモイド関数の数式は、こういった表現方法に慣れていない人にとっては、理解困難で近づき難く感じられるかもしれません。しかし実は、パーセプトロンとシグモイドニューロンには多くの共通点があり、シグモイド関数が代数形式で表現されていることは、真の理解の妨げになるどころか、技術的な細部を伝えてくれるものとなるでしょう。

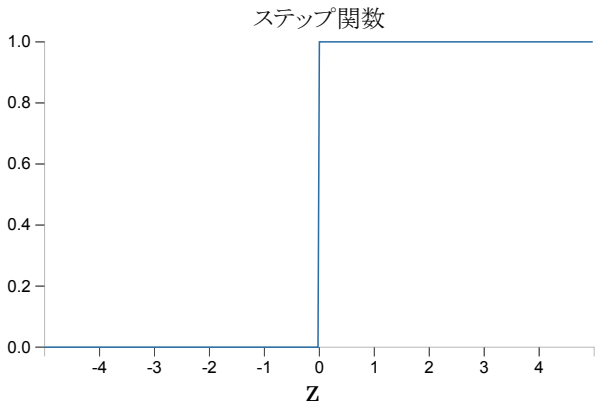
パーセプトロンとの共通点を理解するために、$z \equiv w \cdot x + b$を大きな正の数としてみましょう。このとき、$e^{-z} \approx 0$、つまり$\sigma(z) \approx 1$となります。言い換えると、$z = w \cdot x + b$を大きな数であるとき、シグモイドニューロンの出力はほぼ1となり、パーセプトロンと同じになります。逆に、$z = w \cdot x + b$は大きな負の数とします。そのとき$e^{-z} \rightarrow \infty$であり、$\sigma(z) \approx 0$になります。つまり、$z = w \cdot x + b$が大きな負の数となるときも、シグモイドニューロンはパーセプトロンとほぼ同じ動きをします。ただし、$w \cdot x + b$がそこまで大きな数でない場合はパーセプトロンと同じにはなりません。

σについてですが、代数的視点から私達はこれをどう理解すればいいのでしょうか？ 実は、σがなんであるかはそこまで重要ではありません。重要なのはどういう形のグラフになるかです。これがそのグラフの形です。



*たまに、σを**ロジスティック関数**と呼び、この新しいニューロンを**ロジスティック・ニューロン**と呼ぶことがあります。こちらの用語を使うニューラルネット研究者も大勢いますので、この用語を覚えておくと便利です。とはいえ、私たちは一貫してシグモイド関数という用語の方を使うことにします。

このグラフはステップ関数のなめらか版です:



もしσが実際にステップ関数であれば、シグモイドニューロンはパーセプトロンと**等しくなります**。これは、$w \cdot x + b$が負か正かになることで出力が1か0となるからです。本当のσ関数を使うことによって、上にあるように、なめらかなパーセプトロンになります。確かに、σ関数の滑らかさは重大な事実ですが、本質ではありません。σの滑らかさは、重みについてΔw_j、バイアスについてΔbの小さな変化は、ニューロンの出力についてΔoutputの小さな変化を生み出す、ということを意味しています。実際下記の計算から、Δoutputは大体上手くいっているとわかります。

実は、$w \cdot x + b = 0$のとき、ステップ関数の出力が1に対して、パーセプトロンの出力は0です。正確に言うと、この一点においてステップ関数を変更する必要があります。しかし、わかりますよね。

$$\Delta \text{output} \approx \sum_j \frac{\partial \, \text{output}}{\partial w_j} \Delta w_j + \frac{\partial \, \text{output}}{\partial b} \Delta b, \tag{5}$$

ここではsumは全ての重みw_jの和、$\partial \, \text{output}/\partial w_j$と$\partial \, \text{output}/\partial b$はoutputの偏微分を表し、それぞれに$w_j$と$b$をかけています。偏微分について知らなくてもパニックにならないでください！この記法は複雑に見えますが、全ての偏微分は実は非常にシンプルなことを表現しています(そしてとてもいいことです)。つまり、Δoutputは重みとバイアスにおいて、Δw_jとΔbの変化に対して**線形**である、と言っているのです。この線形性は、欲しいoutputがどんな小さな変化でも、重みとバイアスを小さく変化させることで簡単に得られることを示しています。このことから、シグモイドニューロンはパーセプトロンとほぼ同等な動きをするにも関わらず、より容易に重みとバイアスの変化がoutputを変化させるかがわかります。

もし、本当に重要であるのがσのグラフの形であり、その式自体でないのであれば、なぜσで使われるような特定の等式を使うのでしょうか (3)? 実際、後に時折別の**アクティベーション関数**$f(\cdot)$を使った$f(w \cdot x + b)$の出力を持つニューロンを考えます。別のアクティベーション関数を使った時の主な違いは、等式における偏微分の特定の値です。(5) 後々これらの偏微分を計算するとき、σを使うことで代数的考えが楽になります。これは単に、指数関数の値は指数によって決まるからです。とにかく、σはニ

ューラルネットでよく使われており、この本で最もよく使うアクティベーション関数です。

シグモイドニューロンからの出力をどう変換すべきでしょうか？明らかなことですが、パーセプトロンとシグモイドニューロンの大きな違いの一つは、シグモイドニューロンの出力がちょうど0または1ではないことです。シグモイドニューロンは0から1の間のあらゆる実数を出力することが出来ます。例えば$0.173\ldots$や$0.689\ldots$は正当な出力と言えます。このことはとても有用となりえます。例えば、出力値をニューラルネットワークに対する入力画像のピクセル平均色度合いとして表したい時です。しかし、時々厄介なものともなりえます。ネットワークからの出力を「入力画像が9」もしくは「入力画像が9でない」として示したいとします。明らかに、最も簡単は方法はパーセプトロンのように出力を0もしくは1とすることです。しかし実際のところ、この例を扱うためにルールを設定することが出来ます。例えば、0.5より大きな出力は"9"とみなし、0.5以下の出力は"9ではない"とみなす方法です。混乱がないように、このようなルールを使うときは常に明示することにします。

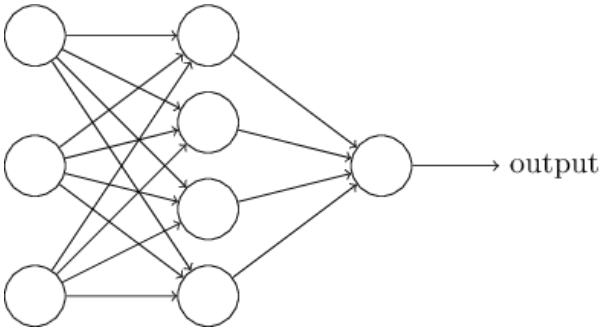
Exercises

- **シグモイドニューロンのシミュレーション パート I**
  今、あるパーセプトロンのネットワークのすべての重みとバイアスをとって、ある正の定数$c > 0$で定数倍するとします。このときネットワークの振る舞いは変わらないことを示してみてください。

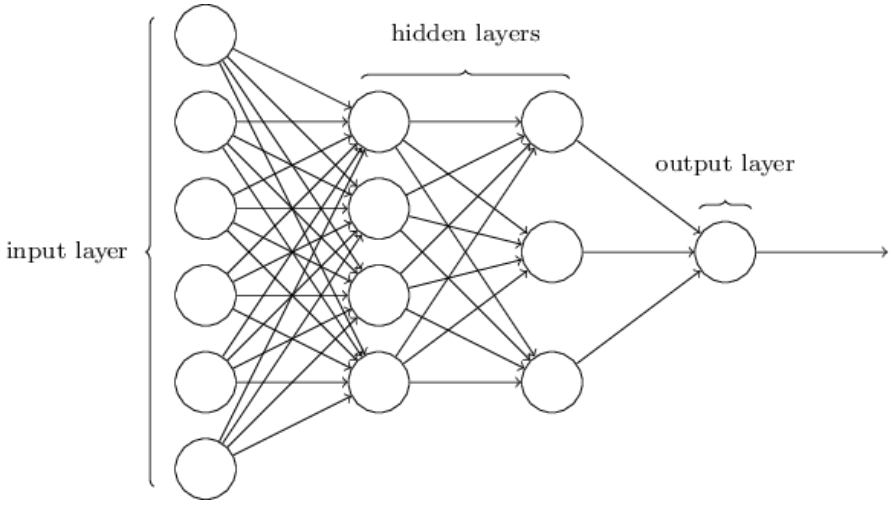
- **シグモイドニューロンのシミュレーション パート II**
  先ほどの問題(=パーセプトロンのネットワーク)と同じ設定で考えます。さらにパーセプトロンネットワークへの入力全体はすでに選ばれているとします。ここで、入力の具体的な値は必要ではなく、入力値が固定されてさえいれば問題ありません。重みとバイアスは、ネットワーク内の任意のパーセプトロンにおいて、入力xに対し$w \cdot x + b \neq 0$を満たしているものとします。今、ネットワーク内の全てのパーセプトロンをシグモイドニューロンで置き換え、重みとバイアスを全て$c > 0$となるような正の数で定数倍するとします。このとき$c \to \infty$の極限において、このシグモイドニューロンのネットワークはパーセプトロンの場合と全く同じように振る舞うことを示して下さい。またパーセプトロンのうちの1つが$w \cdot x + b = 0$を満たす場合にはこの性質は成り立ちません。なぜでしょうか？

ニューラルネットワークのアーキテクチャ

次の章では、手書きの数字の分類においてとても上手く働くニューラルネットワークを紹介します。その準備として、いくつかの専門用語を説明するためにニューラルネットワークのそれぞれの部分に名前をつけておきましょう。



以前述べた通り、一番左の層は**入力層(input layer)**と呼ばれ、その中のニューロンを**入力ニューロン(input neurons)**と言います。一番右の層または**出力層(output layer)**は、**出力ニューロン(output neurons)**から構成されています。上の場合では出力ニューロンは1つですね。中央の層は入力でも出力でもないことから、**隠れ層(hidden layer)**と呼ばれます。この"隠れ"という用語は少し不思議に聞こえるでしょう。私が初めてこの用語を聞いた時、何か哲学的または数学的意味があるのだと思いました。しかしながらこれは"入出力以外"ということを意味しているにすぎません。上記のニューラルネットワークはただ1つの隠れ層からできていますが、複数の隠れ層をもったニューラルネットワークも存在します。例として、下の4層ネットワークは2つの隠れ層をもっています。



紛らわしいことに、歴史的理由から、このような複数層のネットワークをときおり**多層パーセプトロン(multilayer perceptrons)**、または**MLPs**と呼びます。しかしこれらはパーセプトロンではなく、シグモイドニューロンです。これらの用語は混乱を招くためこの本では用いませんが、その存在は知っておいてください。

ニューラルネットワークの入出力層の設計はしばしば単純です。例えば、手書きの画像が9かそうでないかを判断したいとします。設計の自然な方法は、その画像のピクセルあたりの色の度合いを入力ニューロンにエンコードすることです。もしその画像が64×64の白黒画像であれば、入力ニューロンの数は$4,096 = 64 \times 64$になり、色の度合いは明度を0から1の適切な値で表します。出力層は1つのニューロンからなり、出力値が0.5以上なら"入力画像は9である"ということを示し、0.5以下なら"入力画像は9ではない"ということを示します。

入出力層の設計が単純なのに対し、隠れ層の設計はかなり創造的なものになり得ます。とりわけ、隠れ層の設計の過程をいくつかの単純で大雑把な方法で行うのは不可能です。そのかわり、ニューラルネットワークの研究者らは、多くの隠れ層の設計ヒューリスティクスを開発してきました。そしてそれらは人々を解放しました。例として、これらのヒューリスティクスは学習時間と隠れ層の数とのトレードオフに折り合いをつけることができます。私たちも後に、この本の中でそのいくつかの設計に触れることになるでしょう。

これから、ある層の出力が次の層の入力になるようなニューラルネットワークについて考察してみましょう。このようなネットワークは**フィードフォワードニューラルネットワーク(feedforward neural networks)**と呼ばれます。これはネットワーク内にループがないということを意味しています。情報は常に前へ伝わり、後ろへは戻りません。もしループするならば、σ関数の入力はその出力に依存した状態になってしまうでしょう。そうなってはわけがわかりません。そのために私たちはそのようなループを許さないのです。

しかしながら、フィードバックループを用いることが可能な、人工ニューラルネットワークモデルも存在します。それらのモデルは**再帰型ニューラルネットワーク(recurrent neural networks)**と呼ばれます。これらのモデルの着想は、静止するまでの限られた時間に発火するようなニューロンをもったモデルというものです。その発火が他のニューロンを刺激し、そのニューロンもまた限られた時間の中で少し遅れて発火します。そうやってさらなる発火を引き起こし、私たちは発火の連なりを得ることができます。これらのモデルにおいてループは問題にはなりません、なぜならその出力は即時ではなく、いくぶんか遅れてその入力に影響されるからです。

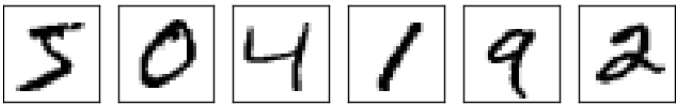
再帰型ニューラルネットワークはフィードフォワードニューラルネットワークに比べてあまり影響力がありませんでした。その理由の一つは、再帰型ネットワークの学習アルゴリズムが非力だったことです。それでも再帰型

ネットワークは非常に興味深いと言えます。それらはフィードフォワードネットワークに比べ、私たちの脳の働き方に近いのです。そしてフィードフォワードネットワークでは困難な問題を、再帰型ネットワークでは解くことができるという可能性は十分にあります。しかしながらこの本では、より広く使われているフィードフォワードニューラルネットワークに焦点を当てたいと思います。

手書き数字を分類する単純なネットワーク

ニューラルネットワークの定義を終えて、いよいよ手書き数字の認識に戻ります。私たちは手書き数字認識の問題を2つの下位問題にわけることができます。1つは、複数桁の数字からなる画像を、それぞれの数字からなる分かれた画像の列にすることです。例えばこの画像を、

6つの分かれた画像にします。



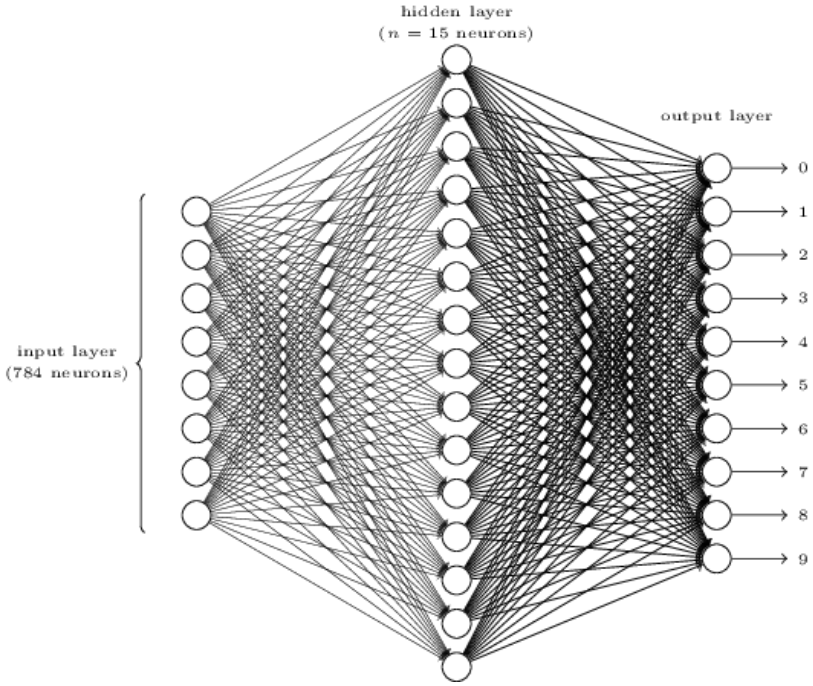
私たち人間はこの**分割問題(segmentation problem)**を容易に解くことができますが、コンピュータプログラムが正確に画像を分解することは容易ではありません。一度画像を分けてしまえば、あとは個々の数字を分類するだけです。つまり上記の数字で、最初にプログラムに認識させるのは、

5です。

これから2つ目の問題、つまり各数字の分類問題を解くプログラムに焦点を当てます。なぜなら、この問題を解く良い方法がわかれば、1つ目の問題、つまり分割問題はそれほど難しくなくなるからです。分割問題へのアプローチは多数あります。そのひとつは、多くの異なる方法で画像を分割して、その画像の分類の結果から、それぞれの分割法を評価する方法です。すべての分割された画像で分類がうまくいけば、その分割法は高いスコアを得ます。逆にうまくいかなければそれはスコアの低い分割法となります。これは、分類でなにか問題が起これば、おそらくそれは誤った分

割法を用いているからだ、というアイディアです。このアイディアやその他の派生した方法は、分割問題をうまく解くことができます。要するに私たちは、分割法に悩む代わりに、もっと面白くて難しい問題、つまり個々の手書き数字の認識問題を解くニューラルネットワークを開発していきます。

それぞれの数字を認識するために、3層のニューラルネットワークを用います。



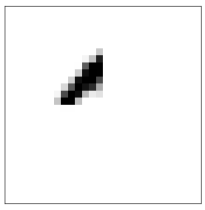
ネットワークの入力層はピクセルの値をエンコードするニューロンを持っています。次の章で論じますが、私たちが使う訓練用データは、手書き数字の28×28ピクセルの画像です。つまり入力層は$28 \times 28 = 784$ニューロンからなるということです。簡単のため、上記の図ではニューロンの数を省略して書いています。入力ピクセルはグレースケールで、0.0は白を、1.0は黒を表し、その間の値はそれに応じた濃さのグレーを表します。

二番目の層は隠れ層です。この隠れ層のニューロンの数をnとし、nの値を変えて実験します。この例では$n = 15$ニューロンだけもつ小規模な隠れ層を表しています。

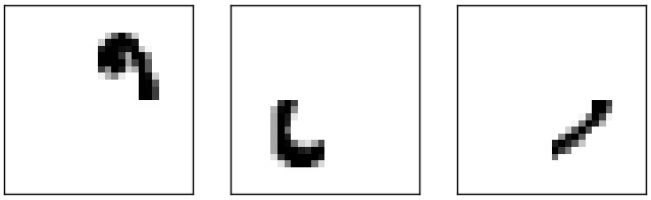
出力層は10ニューロンから構成されています。もし最初のニューロンが発火(出力 ≈ 1)したら、それは、ネットワークがその数字を0だと思っていることを示しています。もし二番目のニューロンなら1、その他も同様です。もう少し正確に言えば、私たちは0から9の出力ニューロンをもっていて、どのニューロンが最も高く活性化するかを計算します。例えばそのニューロンが6だとすると、ネットワークは入力の数字が6であると推測していることになります。他の出力ニューロンについても同様です。

あなたは、なぜ私たちが10個の出力ニューロンを用いるか疑問に思ったでしょう。結局のところ、このネットワークのゴールは、入力の画像に対してそれがどの数字($0, 1, 2, \ldots, 9$)なのか示すことなのです。これを行うのに自然だと思われる方法は、4つの出力ニューロンを用いて、それぞれのニューロンで出力が0または1に近いかどうかに応じてバイナリの値をとることです。答えをエンコードするには4つのニューロンで十分です。なぜなら$2^4 = 16$は入力の数字の10より多くの値をとることができるからです。では**なぜ**私たちはその代わりに10個のニューロンを用いているのでしょうか。非効率的ではないのでしょうか。その究極の正当化は経験に基づくものです。私たちはそのどちらの方法も試し、この特定の問題においては、10個の出力ニューロンをもったネットワークの方が4つのそれよりうまく学習するということがわかったのです。しかし同時に、なぜ10個の出力ニューロンを用いた方がうまくいくのかという疑問が残りました。なにか、私たちが4-出力エンコーディングの代わりに10-出力エンコーディングを用いるべきだというヒューリスティクスがあるのでしょうか。

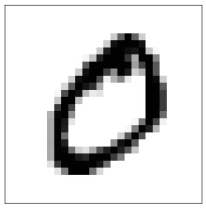
なぜこうするのか理解するために、ニューラルネットワークが何をしているか、その原理から考えます。最初のケース、つまり10個の出力ニューロンを用いた場合を考察してください。最初の出力ニューロンに焦点を当ててみると、これはその数字が0かどうかを決めようとしていることがわかります。これは隠れ層からの情報を考量して行います。それらの隠れ層は何をしているのでしょうか。そうですね、議論のためにとりあえず、隠れ層の最初のニューロンが、下記のような画像が存在するかどうかを検出すると思ってください。



その検出は、その画像と重なった入力ピクセルに重く重み付けし、他の入力には軽い重み付けをすることで、行うことができます。同様の方法で、二番目、三番目、四番目の隠れニューロンも、下記のような画像が存在するかどうかを検出すると思ってください。



おそらくもうあなたが気付いているように、これらの4つの画像は合わせると、私たちが前に見た、数字の列の0の画像になります。



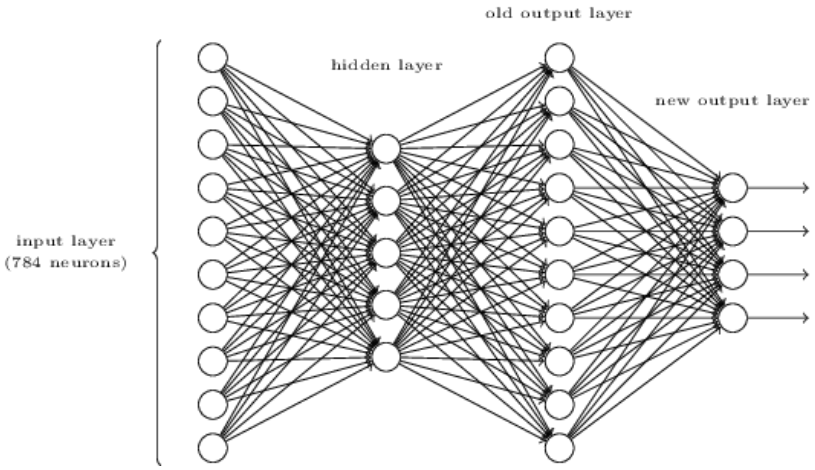
つまり、もしその4つ全ての隠れニューロンが発火したら、私たちはその数字が0であると結論づけることができます。もちろんそれ**だけ**が、その画像が0であると結論づく証拠ではありません。私たちは、その他の多くの方法で(例えば上記の画像の変換や僅かな歪みによって)合理的に0を得ることができます。しかし少なくともこの場合では入力は0だと結論づけて差し支えないでしょう。

ニューラルネットワークがこの方法で機能するとすれば、なぜ10出力ニューロンの方が4よりも良いのかについての尤もな説明を得ることができます。もし4つの出力だとすると、最初の出力ニューロンはその数字の最上位ビットが何なのか決めようとするでしょう。しかしその最上位ビットを、上に示したような単純な形状に関連づける方法はありません。数字を構成する要素の形状が出力の最上位ビットに深く関係する、という歴史的な理由は想像しがたいでしょう。

今言及したこれらのことは、すべてただのヒューリスティクスです。三層ニューラルネットワークは必ずしもこの方法、つまり私が説明した、隠れ層が単純な構成要素を検出するような方法で行う必要はありません。おそらく、巧妙な学習アルゴリズムが、4出力ニューロンを使うような重みの割当てを見つけるでしょう。しかし経験則的に、私が説明してきた考え方はとてもうまくいき、良いニューラルネットワークアーキテクチャを設計する上で、あなたの時間を節約することができます。

エクササイズ

- 3層ネットワークの上にもう一つ層を追加することで、数字のビットワイズ表現を定める方法があります。追加した層は、下記の図のように、前の層からの出力を二進数の表現に変換します。新しい出力層のための重みとバイアスを見つけてください。ただし、最初の3層は、3層目(すなわち古い出力層)の正しい出力が少なくとも0.99で活性化し、誤った出力が0.01以下で活性化するようなものと仮定してください。



勾配降下法を用いた学習

今や私たちはニューラルネットワークのデザインを手に入れましたが、それはどのように数字の認識を学習することができるのでしょうか。最初に必要になるものはそれを用いて学習するための所謂トレーニングデータセットです。私たちは数万件の手書き数字スキャン画像とその正しい分類からなるMNISTデータセットを用います。MNISTという名称は、それがアメリカ国立標準技術研究所（NIST）によって収集および修正（Modify）された二つのデータセットから成り立っていることに由来しています。以下にMNISTの画像をいくつか示します。

実はご覧になられている数字はbeginning of this chapterで用いたものです。もちろん、私たちのネットワークのテストには訓練用ではないものを用います！

MNISTは二つの要素からなっています。一つ目は60,000個の訓練用の画像です。これらの画像は250人の手書きの標本からスキャンされたものであり、250人のうち半数はCensus Bureauの従業員で残り半数は高校生です。これらの画像は28×28ピクセルのグレースケールとなっています。二つ目は10,000個のテスト用画像です。これらの画像も同様に28×28ピクセルのグレースケールとなっています。これらのテストデータを使ってニューラルネットワークが数字の認識についてどれくらい学習できているかを評価します。テストの精度を良くするため、テストデータは訓練用データとは**異なる**250人から採取されています(既にCensus Bureauの従業員と高校生とでグループ分けされているにも関わらずです)。これによりシステムが認識できる数字を訓練中に経験していないと確信できます。

ここで訓練入力を x と定義します。これで各入力 $28 \times 28 = 784$-次元ベクトルを x とみなせ好都合です。ベクトルの各成分は一つのピクセルの濃淡値を表しています。ここで出力を $y = y(x)$ と定義し、この y を10次元のベクトルとします。仮に訓練用画像のxが6を示している場合 $y(x) = (0,0,0,0,0,0,1,0,0,0)^T$ が期待されるネットワークからの出力です。ここでTは転置演算子であり行ベクトルと列ベクトルを入れ替えます。

私たちが得たいもの、それは全訓練入力 x について、ネットワークの出力が $y(x)$ になるべく近くなるような重みとバイアスを見つけるアルゴリズムです。この目標をどれだけ達成できたか測るため**コスト関数**を定義します*:

$$C(w,b) \equiv \frac{1}{2n} \sum_x \|y(x) - a\|^2. \tag{6}$$

*しばしば **損失** 関数 または **目的** 関数とも呼ばれます。私たちはこの本では一貫してコスト関数という用語を用いますが、ニューラルネットワークの論文や議論では他方の用語も頻繁に使われるので心に留めておいて下さい。

ここで w はネットワーク中の全ての重み、b は全バイアス、n は訓練入力の総数、a は入力が x の時にネットワークから出力されるベクトル、和は全ての訓練入力 x です。もちろん出力 a は w と b そして x に依存しますが表記をシンプルにするためここでは敢えて明示しません。$\|v\|$はベクトル v の距離関数を示す記号です。C は**2次**コスト関数と呼びましょう。これはしばしば**平均二乗誤差**あるいは単に**MSE**(mean squared error)としても知られるものです。2次コスト関数の式を見てみると総和の中の全項目が非負であるため $C(w,b)$ は非負になることが分かります。また、$C(w,b)$ が小さくなる時、すなわち $C(w,b) \approx 0$ の時は全訓練入力において $y(x)$ と出力がほぼ等しくなると分かります。つまり、$C(w,b) \approx 0$ となるような重みとバイアスを見つけられれば、私たちの訓練アルゴリズムは上手く機能した、と言えます。対照的に $C(w,b)$ が大きいとき-大多数の入力において $y(x)$ と出力が近似しない場合は上手く機能ているとは言えません。したがって、訓練アルゴリズムの狙いは重みとバイアスの関数 $C(w,b)$ の最小化だと言えます。言い換えれば可能な限りコストを小さくできる重みとバイアスの組を見つけたいのです。それを私たちは**勾配降下法**というアルゴリズムを使って行います。

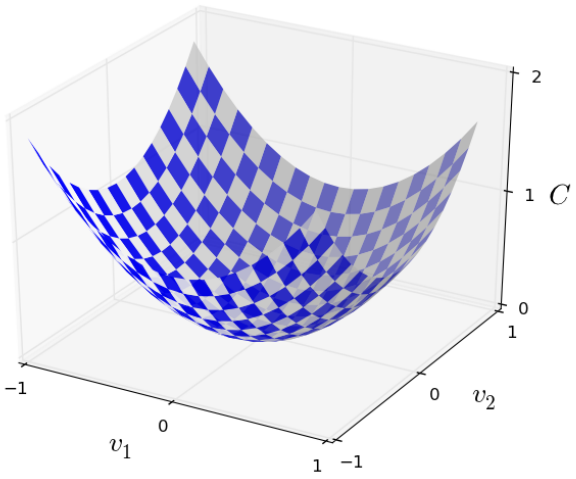
しかし、なぜ2次コストを導出するのでしょうか？結局のところ私たちが知りたいのはどれだけの画像がネットワークによって正しく分類されたかではないでしょうか？直接分類の正解数を最大化せずに2次コストを最小化するのはなぜでしょうか？その理由は分類の正解数がネットワーク中の重みとバイアスの滑らかな関数にならないことです。重みとバイアスに小さな変更を加えても正解数が変化することがほとんどないため、コストを改善するのに重みとバイアスをどう変更したら良いか分からないのです。代わりに2次コストのような滑らかなコスト関数を用いた場合、重みと

バイアスに対してどう小変更を加えればコストを改善できるのかが簡単に分かるようになります。これが2次コストの最小化を用いる理由であり、2次コストの最小化をした後ではじめて分類の精度を調べることにします。

たとえ、滑らかなコスト関数を用いたいとしても、あなたは等式(6) を使った2次コスト関数を選択する理由についてはまだ不思議に思っているかもしれません。これはずいぶんと**アドホックな**選 択ではないでしょうか？もし、仮に違うコスト関数を選んだ場合、最小化する重みとバイアスの組は全く異なってくるのではないでしょうか？それはもっともな心配で、後で私たちはコスト関数を再訪していくつかの修正を行うことになります。しかしながら、この2次コスト関数の等式(6)はニューラルネットワークの学習の基礎を理解するのにとても良く機能するので、今はこのまま続けることにします。

要約すると、ニューラルネットワークの訓練における私たちのゴールは2次コスト関数 $C(w,b)$ を最小化する重みとバイアスを見つけることです。これは良設定問題ですが、いまの定式化のままではたくさんの注意をそらす構造を持っています。重みとバイアスの解釈、背後に潜んでいるシグモイド関数、ネットワーク構造の選択、MNIST等があります。これらが明らかにするのは、私たちは膨大な構成の大部分を無視して単に最小化の面に集中しているということです。そこで、私たちはコスト関数の詳細な式やニューラルネットワークとのつながり、その他諸々については一旦忘れましょう。その代わり、私たちにはたくさんの変数からなる関数がシンプルに与えられており、その関数を最小化したいと考えることにします。私たちは、このような最小化問題を解決できる**勾配降下法**と呼ばれるテクニックを開発するのです。その後、最小化したいニューラルネットワークの具体的な関数に戻ってきましょう。

それでは、私たちは関数 $C(v)$ を最小化しようとしているとしましょう。$C(v)$は複数の引数 $v = v_1, v_2, \ldots$ を取って実数の値を返す関数なら何でもかまいません。ここで私は、どんな関数でも良いということを強調するために w と b の記号を v に置き換えました。もう私たちはニューラルネットワークに特化した文脈で考えているのではありません。ここで $C(v)$ を最小化するのに C が二つの変数からなる関数だと考えることが効果的です。二つの変数を v_1 と v_2 と呼ぶことにしましょう。



私たちが見つけ出したいもの、それは C の大域最小値です。もちろん、今ここで与えられた関数であれば、私たちはグラフを眺めて最小値を見つけられます。そういう意味では、幾分**簡単すぎる**関数を示してしまいました！おそらく一般的な関数 C はたくさんの変数からなる複雑な関数であるためグラフを眺めるだけでは最小値を見つけられないでしょう。

この問題の一つの攻略法は、微積分を使って解析的に最小値を見つけることです。導関数の計算結果から私たちは C の極値を見つけられるでしょう。運よく関数 C が一つの変数、あるいは少数の変数であればおそらく上手く行きます。しかし、変数が大量にある場合は悪夢に変わるでしょう。また、ニューラルネットワークはしばしば **膨大な** 変数を必要とします-もっとも巨大なニューラルネットワークのコスト関数は10億の重みとバイアスを持っており極めて複雑になります。こういった場合、微積分による最小化は機能しません！

(C を二つの変数の関数で考えれば洞察があると主張した後に"変数が二つ以上の場合はどうなってしまうでしょう？"と、二つの段落の中で立場を変えて申し訳ありません。それでも、C を二つの変数の関数で考えることが効果的だという私の言うことを信じてください。最後の2段落は概観の分析を行っているのです。しばしば数学に関する名案では、複数の直観的イメージを巧みに扱い、学習する際のイメージの適切な使い分けを伴うのです。)

さて、微積分は機能しません。幸いなことに、非常に良く機能する一つのアルゴリズムを示唆する見事な例え話があります。手始めに関数が谷であるかのように想像してみましょう。上のグラフを見てボールが谷の斜面を転がり落ちていくところを想像してください。普段の経験から、ボールは最終的に谷底まで転がっていくと分かるでしょう。この考え方を関数の最

小化に使えないでしょうか？私たちは(想像上の)ボールのスタート地点をランダムに選び、その後ボールが谷底へ転がっていく動きをシミュレーションするのです。おそらく、単に C の導関数(あるいは二次導関数)を微分すればこのシミュレーションが行えるでしょう。これらの微分係数は、谷の局所形状やボールがどう転がるかといった私たちが知るべき全てのことを教えてくれます。

あなたは私が述べた内容に基づき、私たちが摩擦力や重力の影響等を考慮し、ニュートンの運動方程式を書き始めると思うかもしれません。実際には、ボールの転がりの例えをそう深刻に扱ったりはしません。私たちは C の最小化アルゴリズムを考案しようとしているのであり、物理法則の精密なシミュレーションを開発するわけではありません。ボール目線の観点は想像力を刺激するためのものであり、思考を制限するためではありません。そういうわけで、物理学の詳細には入っていかずにシンプルな問い掛けをします:もし私たちが一日神様を任命され、物理法則を好きに決めていいことになったら、ボールにどう動くよう命令すべきでしょう？どんな運動法則を選べばボールは谷底へと転がり続けていくでしょう？

この問いをもう少し詳細化するため、v_1 方向に微小な量 Δv_1 、v_2 方向に微小な量 $\Delta v2$ だけボールを動かした時に何が起こるか考えてみましょう。計算の結果、C は次のようになります:

$$\Delta C \approx \frac{\partial C}{\partial v_1} \Delta v_1 + \frac{\partial C}{\partial v_2} \Delta v_2. \tag{7}$$

ここで ΔCが負の値;すなわち、ボールが谷を転がり降りていくような Δv_1 と Δv_2 を選ぶ方法を見つけましょう。これを明らかにするため、Δv を vの変化のベクトルとして、$\Delta v \equiv (\Delta v_1, \Delta v_2)^T$ と定義し、ここで T は転置演算子(再掲)なので、行ベクトルと列ベクトルを入れ替えます。同様に、C の**勾配**についても偏導関数のベクトル $\left(\frac{\partial C}{\partial v_1}, \frac{\partial C}{\partial v_2}\right)^T$ として定義します。ここで勾配ベクトルを ∇C と記して:

$$\nabla C \equiv \left(\frac{\partial C}{\partial v_1}, \frac{\partial C}{\partial v_2}\right)^T. \tag{8}$$

すぐに私たちは ΔC を Δv と勾配 ∇C に書き換えるのですが、これに着手する前に、勾配のハマりやすい箇所を明らかにしておきたいと思います。∇C の表記に出会った時、しばしば人々は ∇ の記号をどう考えて良いのか分からずに戸惑います。∇ の正確な意味は何でしょう？実際、∇C が数学における一つの記号ということは自明で-上記定義のベクトル-それは二つの記号を使って表記されています。この観点で言えば ∇ は" ∇C は勾配ベクトル"とあなたに教えるため旗を振る記号の一つです。更に踏み込んだ観点では ∇ はそれ自体で独立した数学の構成要

素(例えば微分演算子のようなもの)であると見なせもしますが、こういった観点は私たちには必要ありません。

これまでの定義から式 (7) の ΔC を次のように変形できます。

$$\Delta C \approx \nabla C \cdot \Delta v. \tag{9}$$

この等式は ∇C がなぜ勾配ベクトルと呼ばれるかを教えてくれます: ∇C は C を変化させる V の変化に関わっており、これはちょうど私たちが勾配と呼んでいるものです。しかし、本当に面白いのはこの等式が ΔC を負にする Δv の選び方を教えてくれるということです。とりわけ、次の仮定を与えれば

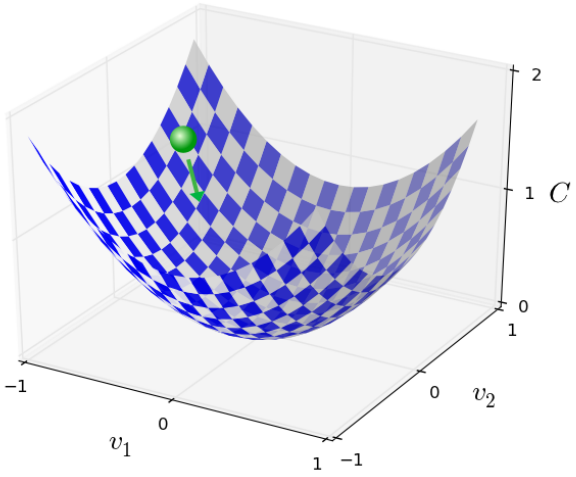
$$\Delta v = -\eta \nabla C, \tag{10}$$

η は小さい正のパラメータ(**学習率**として知られるもの)です。ここで等式 (9) から $\Delta C \approx -\eta \nabla C \cdot \nabla C = -\eta \|\nabla C\|^2$ となることが分かります。$\|\nabla C\|^2 \geq 0$ であることから $\Delta C \leq 0$ が成り立つため(10)の前提に従い v を変更する限り C は常に減少し、決して増加しないことが保証されます(勿論(9) の等式が近似する限りです)。これはまさしく私たちが求めていた特性です！そこで等式(10)を私たちの勾配降下アルゴリズムのボールの"運動の法則"と定義しましょう。つまり、私たちは等式(10)を使い Δv の値を計算し、ボールの位置を v から次のように動かすのです:

$$v \rightarrow v' = v - \eta \nabla C. \tag{11}$$

その後、私たちは以降もこの規則を使い続けます。もし私たちがこれを続けて、何度も繰り返すと、C は減少を続け、やがては - 待望の - 大域最小値に到達します。

要約すると、勾配降下法は勾配 ∇C を計算し**逆の**方向へと動かすことを繰り返すことで谷の斜面へと"降下"させる方法です。これを視覚化すると以下のようになります。



ここで留意すべきは勾配降下法の規則が現実世界の物理的な運動を再現していないということです。現実世界のボールは運動量を持っているので、斜面を転がり、(少しの間)そのまま登っていくかもしれません。その後、摩擦力によってはじめて谷を降り始めるでしょう。これに対して、私たちが Δv を選ぶ規則は"今この瞬間だけ降りなさい"というものです。これはやはり最小値を見つけるのにとても良い規則です！

勾配降下法を正しく動作させるには十分小さな学習率 η を選んで等式(9)をよく近似させる必要があります。さもなければ $\Delta C > 0$ となり明らかに良くありません！その一方で η が小さすぎる場合は Δv の変化がとても小さくなり勾配降下法の動きは非常に遅くなってしまいます。実用的な実装では、等式(9)の近似を維持できるように η を頻繁に変更してアルゴリズムが遅くなりすぎないようにします。これがどのように行われるかは後の章で理解することにしましょう。

私は勾配降下法の C がちょうど二つの変数の関数である場合を説明しました。しかし、実際には、C がもっと多くの変数の関数であっても何も問題はありません。ここで C が m 変数 $v_1, \ldots, v_m$ の関数であると仮定します。この時、微小な変化 $\Delta v = (\Delta v_1, \ldots, \Delta v_m)^T$ によって持たされる C の変化 ΔC は

$$\Delta C \approx \nabla C \cdot \Delta v, \tag{12}$$

ここで勾配 ∇C のベクトルは

$$\nabla C \equiv \left(\frac{\partial C}{\partial v_1}, \ldots, \frac{\partial C}{\partial v_m} \right)^T. \tag{13}$$

二変数の時と同様に、次のように設定します。

$$\Delta v = -\eta \nabla C, \tag{14}$$

これで等式(12)の(近似)式の ΔC が負の値となるように保証されます。C が複数の変数の関数であっても、この定義を繰り返し更新して当てはめていけば、最小値への勾配に繋がる道が得られます。

$$v \rightarrow v' = v - \eta \nabla C. \tag{15}$$

この規則は勾配降下法の**定義**とみなすことができます。この規則によってv の位置を繰り返し変更して関数 C を最小化する 方法が分かります。この規則はどんな時でも機能する訳ではありません - しばしば間違い、勾配降下法が大域最小値の発見を妨げる場合があります。この点は、後の章でまた戻って綿密に調べます。しかし、実際には勾配降下法はほとんどの場合とても良く機能し、ニューラルネットワークのコスト関数の非常に強力な最小化手段でありネットワークの学習を助けてくれます。

実際、勾配降下法は最小値を探索する最適戦略であるとさえ感じます。私たちは C を可能な限り減少する位置へと Δv 動かそうとしていると仮定してみましょう。これは $\Delta C \approx \nabla C \cdot \Delta v$ に等しいです。ここで移動量 $\|\Delta v\| = \epsilon$ に微小な固定値 $\epsilon > 0$ という制約を与えます。言い換えれば、私たちは小刻みな固定値の動きを望んでいて、C を可能な限り減少させる移動方向を見つけようとしているのです。これは $\nabla C \cdot \Delta v$ を最小化する Δv が $\Delta v = -\eta \nabla C$ であり $\eta = \epsilon / \|\nabla C\|$ は制約量 $\|\Delta v\| = \epsilon$ により決まるということから証明できます。つまり、勾配降下法はその瞬間に C を最も減少させる方向へと小刻みに動く方法と見なすことができます。

演習

- 最後の段落の主張を証明してください。**ヒント:** もしあなたがまだコーシーシュワルツの不等式について詳しくなければ、習熟することが理解に役立つでしょう。

- 私は勾配降下法の C が二つの変数である場合と二つ以上の変数の関数の場合について説明しました。C がただ一つの関数の場合は何が起こるでしょう？あなたは一次元の場合の勾配降下法の動きについて幾何学的な説明が出来ますか？

より現実のボールに近い形で物理法則を摸倣する種類を含め、これまで様々な勾配降下法が研究されてきました。そういったボールを模倣する種類はいくつか長所を持っているものの、大きな欠点も持っています：二階偏微分の計算が必要であり、その計算が非常に大変なのです。なぜ計算が大変か明らかにするため、全ての二階偏微分 $\partial^2 C / \partial v_j \partial v_k$ を計算したいと仮定してみましょう。仮に100万の変数がある時、私たちはおよそ1兆回(つまり100万の2乗)の二階微分*！の計算が必要で計算量的に

*実際は、1兆の半分、なぜなら $\partial^2 C / \partial v_j \partial v_k = \partial^2 C / \partial v_k \partial v_j$ だからです。

重くなります。そうは言っても、こういった問題を回避する手段はいくつか存在していますし、また勾配降下法の代替手段の調査は研究が盛んな分野になっています。しかし、私たちはこの本では勾配降下法(とその派生形)をニューラルネットワークの学習への主要なアプローチに使いましょう。

私たちはどうすればニューラルネットワークの学習に勾配降下法を適用できるでしょう？その考え方は等式(6)のコストを最小化する重みとバイアスの探索に勾配降下法を用いるというものです。これがどう行われるかを理解するため、変数 v_j を重みとバイアスに置き換えて勾配降下法の更新規則を再定義しましょう。つまり、私たちの"位置"は要素として w_k と b_l を持っており、勾配ベクトル ∇C は要素として $\partial C/\partial w_k$ と $\partial C/\partial b_l$ を持っていることに一致します。これらの要素の用語で勾配降下法の更新規則を書き直すと、

$$w_k \rightarrow w_k' = w_k - \eta \frac{\partial C}{\partial w_k} \tag{16}$$

$$b_l \rightarrow b_l' = b_l - \eta \frac{\partial C}{\partial b_l}. \tag{17}$$

この更新規則を繰り返し適用することで"坂を転がり降りる"ことができ、上手くいけばコスト関数の最小値を見つけられます。言い換えれば、この規則をニューラルネットワークの学習に使うことが出来ます。

勾配降下法の規則の適用にはいくつか課題があります。この詳細は後の章で見ることにしましょう。それより今は一つの問題にだけ言及したいと思います。問題が何であるかを理解するため、等式(6)の二次コスト関数を振り返りましょう。ここでコスト関数は $C = \frac{1}{n} \sum_x C_x$ という形をしており、個々の訓練データ $C_x \equiv \frac{\|y(x)-a\|^2}{2}$ の総和になっていることが分かると思います。実際には、私たちは勾配 ∇C を計算するため、個々の訓練入力 xの勾配 ∇C_x を計算し、その後その平均を取って $\nabla C = \frac{1}{n} \sum_x \nabla C_x$ とします。不運にも、訓練入力の数が非常に大きい場合はとても時間が掛かり、その結果学習は非常に遅くなってしまいます。

学習の高速化に使えるアイディアの一つに**確率的勾配降下法**と呼ばれるものがあります。この考え方は訓練入力から無作為に抽出した小さな標本群 ∇C_x を計算して勾配 ∇C を推定するというものです。小さな標本群の平均を取ることで速やかに正しい勾配 ∇C を推定でき、勾配降下法が高速化され、ひいては学習を高速化できます。

この考え方をより正確に述べると、確率的勾配降下法は、小さい数 m を無作為抽出し、訓練入力をその数だけ無作為に選ぶことで動くということです。ここでランダムに選んだ訓練入力を $X_1, X_2, \ldots, X_m$ とラベル付け

し、これらを**ミニバッチ**と呼ぶことにしましょう。標本サイズ m が十分大きければ ∇C_{X_j} の平均値は全ての ∇C_x の平均とほぼ同等になることが期待でき、すなわち、

$$\frac{\sum_{j=1}^{m} \nabla C_{X_j}}{m} \approx \frac{\sum_x \nabla C_x}{n} = \nabla C, \tag{18}$$

ここで二つ目の総和は全ての訓練データです。端と入れ替えると、

$$\nabla C \approx \frac{1}{m} \sum_{j=1}^{m} \nabla C_{X_j}, \tag{19}$$

ランダムに選んだミニバッチを計算して全体の勾配を推定できることが確認できます。

これを明確にニューラルネットワークの学習と紐付けるため、私たちのニューラルネットワークにおける重みとバイアスの表記 w_k と b_l で考えてみましょう。この時、確率的勾配降下法は無作為に選んだ訓練入力のミニバッチによって動き、それらで訓練を行います。

$$w_k \rightarrow w_k' = w_k - \frac{\eta}{m} \sum_j \frac{\partial C_{X_j}}{\partial w_k} \tag{20}$$

$$b_l \rightarrow b_l' = b_l - \frac{\eta}{m} \sum_j \frac{\partial C_{X_j}}{\partial b_l}, \tag{21}$$

総和は現在のミニバッチにおける全ての訓練サンプル X_j です。次に、私たちは別の無作為に選んだミニバッチで訓練を行います。同じように、訓練入力がなくなるまで続ければ、1回の訓練の**エポック**(訳注:訓練データ全体を1巡する事)が完了します。この時点で私たちは新しい訓練エポックをやり直します。

ちなみに、コスト関数とミニバッチで重みとバイアスを更新する縮尺の取り方によって、更新規則が異なってくることは注目に値します。等式(6)で私たちは全てのコスト関数を $\frac{1}{n}$ の縮尺にしました。しばしば人々は $\frac{1}{n}$ を省略し、個々の訓練例のコストの平均を取る代わりに総和を取ります。これはとりわけ訓練例の総数が事前に分かっていない場合に有効です。これは、例えば、リアルタイムに訓練データが生成されている場合に生じ得ます。そして、同じように、ミニバッチの更新規則 (20) と (21) でもしばしば総和の前にある $\frac{1}{m}$ を省略します。これは学習率 η の縮尺の大きさを変更することに相当するので概念的には大差がありません。しかし、両者の動作の詳細な比較について気にすることは価値のあることです。

確率的勾配降下法は世論調査のように考えることができます:国民総選挙よりも世論調査を実施する方が簡単であるように、全部一括処理で実施するより小さな標本のミニバッチの方が勾配降下法の適用は簡単で

す。例えば、仮に私たちがMNIST等で $n = 60,000$ の訓練セットを持っておりミニバッチの大きさが(例えば) $m = 10$とすると、勾配の推定を $6,000$ 倍速く出来ます！勿論、この推定は完璧ではありません - これには統計変動があるでしょう - しかし、完璧である必要はありません：私たちが気にするのは C が減少する大まかな移動方向だけなので、勾配の正確な計算は必要ありません。実際、確率的勾配降下法はニューラルネットワークの学習によく用いられている強力な手法であり、また、私たちがこの本で開発する学習テクニックにおける大部分の基礎になります。

演習

- 極端な勾配降下法は大きさ1のミニバッチを使います。つまり、与えられた一つの訓練入力 x について、重みとバイアスを規則に従い更新して $w_k \rightarrow w_k' = w_k - \eta \partial C_x / \partial w_k$ と $b_l \rightarrow b_l' = b_l - \eta \partial C_x / \partial b_l$。その次に、別の訓練入力を選び、そして再びバイアスを更新します。この手続きは**オンライン**学習または**逐次**学習として知られるものです。オンライン学習は、ニューラルネットワークの一回の学習を一つの訓練入力で行います(ちょうど人間がそうするように)。オンライン学習の長所と短所を一つずつ、確率的勾配降下法のミニバッチの大きさが 20 の場合と比較して挙げてください。

本節の結びに勾配降下法に慣れていない人をしばしば悩ませる点を議論させてください。ニューラルネットワークにおけるコスト C は、当然ながら、複数の変数の関数であり - 全ての重みとバイアス - それゆえに非常に高次元な空間上の曲面であるともいえます。一部の人はこのとき、"私はこれらの超次元の可視化ができるようになる必要がある"と考え、困ってしまいます。そして彼らは心配しはじめます："私は四次元で考えることができないし、五次元(あるいは五百万次元)なんてもっと無理だ"。彼らには"真の"数学者が持っている何か特別な能力が欠けているのでしょうか？勿論、答えはノーです。本職の数学者でも四次元の可視化はできませんし、それ以上についてはなおさらです。彼らは別の表現方法を開発するというトリックを使っているのです。それはちょうどこれまで私たちがやってきた：C を減少させる方法を明らかにする ΔC の代数表現です(視覚化ではなく)。高次元について考えるのが得意な人は頭の中にこの種の多種多様なテクニックをライブラリとして持っています；私たちの代数的トリックもその一つです。これらのテクニックには私たちの慣れている三次元を視覚化する時の簡素性は持ってませんが、ひとたびそういったテクニックのライブラリを確立すれば、あなたは高次元について考えることがとても得意になるでしょう。私はこれ以上この詳細に入っていきませんが、もしあなたに興味があるなら、本職の数学者が高次元の思考に用いるテク

ニックのこの議論 を読んで楽しめるでしょう。いくつかのテクニックに関する議論は非常に難解ですが、たくさんの素晴らしいコンテンツが直観的でとっつきやすく、そして、全ての人が習得できるものです。

数字を分類するニューラルネットワークの実装

それでは、手書き数字を認識する方法を学ぶプログラムを作成していきましょう。その際、確率的勾配降下法とMNISTの訓練データを使用します。最初に私たちはMNISTデータを手に入れる必要があります。もしあなたがgitのユーザーならば、下記のリポジトリからクローンすることでデータを取得できます。

```
git clone https://github.com/mnielsen/neural-networks-and-deep-learning.git
```

もしgit のユーザーではない場合は、あなたはここからデータとコードをダウンロードすることができます。

そういえば、以前MNISTデータを説明したとき、MNISTデータは60,000枚の訓練用画像と10,000枚の試験用画像に分かれているって言いましたよね。じつは、データの分け方を少々変えようと思います。試験用画像はそのままにして、60,000枚の訓練用画像のうち、50,000枚を訓練のために使い、10,000枚は **検証データセット**としてとっておきます。検証データはこの章では使いませんが、本書の後の方で **ハイパーパラメータ**を設定するのに重宝します。ハイパーパラメータとは、学習率など、学習アルゴリズムで直接選択できないもののことです。検証データセットは元々のMNIST仕様には含まれていませんが、多くの人はMNISTをこのやり方で使っていますし、検証データはニューラルネットワークではよく使われます。今後、「MNIST訓練データ」と言った場合は、オリジナルの60,000枚の画像セットではなく、今作った 50,000枚の画像からなるデータセットのことを指すこととします。*

MNISTデータとは別に、高速に線形代数を解くことができるNumpyと呼ばれるPythonライブラリーが必要です。もしあなたがNumpyをインストールしていないならば、ここから手に入れてください。

それでは、完全なプログラムリストを示す前に、ニューラルネットワークのコードのコア機能の説明を以下でしましょう。コードの中心部はNetworkクラスであり、ニューラルネットワークを表現するために使います。以下が、Networkを初期化するためのコードです。

```
class Network():

    def __init__(self, sizes):
```

*前述したように、MNISTデータセットは、アメリカ国立標準技術研究所(NIST)によって定義されているものです。MNISTを構築するために、NISTデータはYann LeCun, Corinna Cortes, Christopher J. C. Burgesによって整理・より便利なフォーマットを追加されました。詳しいことには、こちらのリンクを見てください。私のレポジトリの中のこのデータセットは、Pythonによって読み込みしやすく、操作しやすいフォーマットになっています。このデータのフォーマットは、モントリオール大学のLISA machine learning laboratory (link)から取得しました。

```
        self.num_layers = len(sizes)
        self.sizes = sizes
        self.biases = [np.random.randn(y, 1) for y in sizes[1:]]
        self.weights = [np.random.randn(y, x)
                        for x, y in zip(sizes[:-1], sizes[1:])]
```

sizesは、それぞれの層におけるニューロンの数を表しています。もし1層目に2つのニューロン、2層目に3つのニューロン、最終層に1つのニューロンを持つNetworkを作りたいならば、以下のようにコードを定義します。

```
net = Network([2, 3, 1])
```

Networkの中のバイアスと重みは、Numpyのnp.random.randnによって生成された平均値0・標準偏差1のガウス分布の乱数に初期化されます。この初期化のための乱数は、確率的勾配降下法の開始点として使用します。今回は乱数によって初期値を決めることにしますが、後半の章では、重みとバイアスの初期化のより良い方法を解説します。Networkの初期化のコードでは、ニューロンの1層目は入力層であり、バイアスは後半の層から出力を計算するときにだけ使われるので、入力層のニューロンのバイアスは省略する仮定をしていることについて注意してください。

Numpy内の行列のリストとしてバイアスと重みは保存されることについても注意してください。なので、net.weights[1]は、2層目と3層目をつなぐ重みを保存するNumpyの行列です。(Pythonのインデックスは0から開始されるので、1層目と2層目を繋ぐ重みではありません。) net.weights[1]という記述は冗長なので、ここではwという行列として示しましょう。それは、w_{jk}という行列で表現されていて、2層目のk番目のニューロンと3層目のj番目のニューロンを繋ぐ重みです。このjとkの順序は奇妙に見えるかもしれません。確かにjとkを交換するほうが理に適っていそうです。この順序の大きな利点は、ニューロン3層目の活性化のベクトルは以下を意味することです。

$$a' = \sigma(wa + b). \tag{22}$$

この式では、かなり多くの振る舞いがあるので一つ一つ紐解いてみましょう。aは2層目の活性化のベクトルです。a'を得るために、私たちはaと重み行列wを掛け算し、バイアスのベクトルbを足し算します。私たちは、ベクトル$wa + b$に関数σを作用させます。(これは関数σの**vectorizing**と呼ばれます。) 等式 (22) が、シグモイドニューロンの出力を計算するための 等式 (4) と同じ結果になることを確認するのは簡単です。

Exercise

- Equation (22) をベクトルの要素を記述し、そしてシグモイドニューロンの出力を計算するための Equation (4) と同じ結果を与えること

を確認しましょう。

こうした流れで、Network から出力を計算するコードを記述するのは簡単なことがわかります。シグモイド関数を定義することからはじめます。このとき、シグモイド関数はベクトル形式でNumpyを使って定義します。

```
def sigmoid(z):
    return 1.0/(1.0+np.exp(-z))

sigmoid_vec = np.vectorize(sigmoid)
```

ネットワークの入力a が与えられたら、対応した出力を返すfeedforwardをNetworkクラスに追加します。このメソッドは、層ごとに等式 (22) を適用します。

```
    def feedforward(self, a):
        """Return the output of the network if "a" is input."""
        for b, w in zip(self.biases, self.weights):
            a = sigmoid_vec(np.dot(w, a)+b)
        return a
```

もちろん、Networkにしてほしいことは学習することです。そのために確率的勾配降下法(SGD)を使用します。コードはここに記します。少しばかり不可解な場所がありますが、それについては下記で解説していきます。

```
    def SGD(self, training_data, epochs, mini_batch_size, eta,
            test_data=None):
        """Train the neural network using mini-batch stochastic
        gradient descent.  The "training_data" is a list of tuples
        "(x, y)" representing the training inputs and the desired
        outputs.  The other non-optional parameters are
        self-explanatory.  If "test_data" is provided then the
        network will be evaluated against the test data after each
        epoch, and partial progress printed out.  This is useful for
        tracking progress, but slows things down substantially."""
        if test_data: n_test = len(test_data)
        n = len(training_data)
        for j in xrange(epochs):
            random.shuffle(training_data)
            mini_batches = [
                training_data[k:k+mini_batch_size]
                for k in xrange(0, n, mini_batch_size)]
            for mini_batch in mini_batches:
                self.update_mini_batch(mini_batch, eta)
            if test_data:
                print "Epoch {0}: {1} / {2}".format(
                    j, self.evaluate(test_data), n_test)
            else:
                print "Epoch {0} complete".format(j)
```

training_dataは、訓練入力と対応した目的出力の組(x, y)のリストです。変数epochsとmini_batch_sizeは訓練のための世代数と、サンプリングするときに使用するミニバッチの大きさです。変数etaは学習率ηです。もしオプションの引数test_dataがある場合、プログラムは各訓練のエポックのあとにネットワークを評価して、現在の進行状況を出力します。この機能は

性能改善の進行状況を確認するときに役に立ちますが、計算に少し時間がかかるようになります。

コードは以下のように機能します。各エポックでは、訓練データをランダムにシャッフルすることによって開始し、適切なサイズのミニバッチに分割します。このコードは、訓練データからランダムにサンプルする簡単な方法になります。各ミニバッチに、勾配降下法を1ステップ実行します。これは、コードself.update_mini_batch(mini_batch, eta)によって行われ、ミニバッチの訓練データだけを使用して勾配降下法を実行し、ネットワークの重みとバイアスを更新します。ここに、update_mini_batchのコードを示します。

```
    def update_mini_batch(self, mini_batch, eta):
        """Update the network's weights and biases by applying
        gradient descent using backpropagation to a single mini batch.
        The "mini_batch" is a list of tuples "(x, y)", and "eta"
        is the learning rate."""
        nabla_b = [np.zeros(b.shape) for b in self.biases]
        nabla_w = [np.zeros(w.shape) for w in self.weights]
        for x, y in mini_batch:
            delta_nabla_b, delta_nabla_w = self.backprop(x, y)
            nabla_b = [nb+dnb for nb, dnb in zip(nabla_b, delta_nabla_b)]
            nabla_w = [nw+dnw for nw, dnw in zip(nabla_w, delta_nabla_w)]
        self.weights = [w-(eta/len(mini_batch))*nw
                        for w, nw in zip(self.weights, nabla_w)]
        self.biases = [b-(eta/len(mini_batch))*nb
                       for b, nb in zip(self.biases, nabla_b)]
```

作業の多くは下記のコードで行われます

```
            delta_nabla_b, delta_nabla_w = self.backprop(x, y)
```

このコードは、コスト関数の勾配を計算する高速な方法である誤差逆伝播法(**backpropagation**)アルゴリズムを起動する部分です。update_mini_batchは単純にミニバッチ内の訓練データごとに勾配を計算し、self.weightsとself.biasesを適切に更新します。

self.backpropのコードは今すぐには説明しません。次の章で誤差逆伝播法について勉強し、その際にself.backpropのコードを紹介します。なので今は、訓練データのxに関連するコストに対して適切な勾配を返す働きをするということを前提にします。

それでは、下記の完全なプログラムを見てください。この際、説明を省略した部分や説明文を含んでいます。self.backpropを除いて、プログラムは明快であり、すでにお話しした通り、全ての処理の重い部分はself.SGDとself.update_mini_batchで行われています。self.backpropは勾配を計算することを手助けするためのいくつかの追加機能を使用しており、詳細はここでは説明しませんが、σ関数の導関数を計算するsigmoid_prime、ベクトル形式のsigmoid_prime_vecとself.cost_derivativeです。次の章で詳細を確認しますが、コードと説明を見る事によって要点を理解する事ができま

す。プログラムが長いように見えますが、多くはプログラム内の理解を促すための解説文であり、コード自体は理解が簡単に書いているつもりです。実際、プログラムはたった空行と解説を除いて74行です。コードのすべてはGitHubで見つけることができます。

```
"""


network.py
~~~~~~~~~~

A module to implement the stochastic gradient descent learning
algorithm for a feedforward neural network.  Gradients are calculated
using backpropagation.  Note that I have focused on making the code
simple, easily readable, and easily modifiable.  It is not optimized,
and omits many desirable features.
"""

#### Libraries
# Standard library
import random

# Third-party libraries
import numpy as np

class Network():

    def __init__(self, sizes):
        """The list ``sizes`` contains the number of neurons in the
        respective layers of the network.  For example, if the list
        was [2, 3, 1] then it would be a three-layer network, with the
        first layer containing 2 neurons, the second layer 3 neurons,
        and the third layer 1 neuron.  The biases and weights for the
        network are initialized randomly, using a Gaussian
        distribution with mean 0, and variance 1.  Note that the first
        layer is assumed to be an input layer, and by convention we
        won't set any biases for those neurons, since biases are only
        ever used in computing the outputs from later layers."""
        self.num_layers = len(sizes)
        self.sizes = sizes
        self.biases = [np.random.randn(y, 1) for y in sizes[1:]]
        self.weights = [np.random.randn(y, x)
                        for x, y in zip(sizes[:-1], sizes[1:])]

    def feedforward(self, a):
        """Return the output of the network if ``a`` is input."""
        for b, w in zip(self.biases, self.weights):
            a = sigmoid_vec(np.dot(w, a)+b)
        return a

    def SGD(self, training_data, epochs, mini_batch_size, eta,
            test_data=None):
        """Train the neural network using mini-batch stochastic
        gradient descent.  The ``training_data`` is a list of tuples
        ``(x, y)`` representing the training inputs and the desired
        outputs.  The other non-optional parameters are
        self-explanatory.  If ``test_data`` is provided then the
        network will be evaluated against the test data after each
        epoch, and partial progress printed out.  This is useful for
        tracking progress, but slows things down substantially."""
```

```
        if test_data: n_test = len(test_data)
        n = len(training_data)
        for j in xrange(epochs):
            random.shuffle(training_data)
            mini_batches = [
                training_data[k:k+mini_batch_size]
                for k in xrange(0, n, mini_batch_size)]
            for mini_batch in mini_batches:
                self.update_mini_batch(mini_batch, eta)
            if test_data:
                print "Epoch {0}: {1} / {2}".format(
                    j, self.evaluate(test_data), n_test)
            else:
                print "Epoch {0} complete".format(j)

    def update_mini_batch(self, mini_batch, eta):
        """Update the network's weights and biases by applying
        gradient descent using backpropagation to a single mini batch.
        The ``mini_batch`` is a list of tuples ``(x, y)``, and ``eta``
        is the learning rate."""
        nabla_b = [np.zeros(b.shape) for b in self.biases]
        nabla_w = [np.zeros(w.shape) for w in self.weights]
        for x, y in mini_batch:
            delta_nabla_b, delta_nabla_w = self.backprop(x, y)
            nabla_b = [nb+dnb for nb, dnb in zip(nabla_b, delta_nabla_b)]
            nabla_w = [nw+dnw for nw, dnw in zip(nabla_w, delta_nabla_w)]
        self.weights = [w-(eta/len(mini_batch))*nw
                        for w, nw in zip(self.weights, nabla_w)]
        self.biases = [b-(eta/len(mini_batch))*nb
                       for b, nb in zip(self.biases, nabla_b)]

    def backprop(self, x, y):
        """Return a tuple ``(nabla_b, nabla_w)`` representing the
        gradient for the cost function C_x.  ``nabla_b`` and
        ``nabla_w`` are layer-by-layer lists of numpy arrays, similar
        to ``self.biases`` and ``self.weights``."""
        nabla_b = [np.zeros(b.shape) for b in self.biases]
        nabla_w = [np.zeros(w.shape) for w in self.weights]
        # feedforward
        activation = x
        activations = [x] # list to store all the activations, layer by layer
        zs = [] # list to store all the z vectors, layer by layer
        for b, w in zip(self.biases, self.weights):
            z = np.dot(w, activation)+b
            zs.append(z)
            activation = sigmoid_vec(z)
            activations.append(activation)
        # backward pass
        delta = self.cost_derivative(activations[-1], y) * ¥
            sigmoid_prime_vec(zs[-1])
        nabla_b[-1] = delta
        nabla_w[-1] = np.dot(delta, activations[-2].transpose())
        # Note that the variable l in the loop below is used a little
        # differently to the notation in Chapter 2 of the book.  Here,
        # l = 1 means the last layer of neurons, l = 2 is the
        # second-last layer, and so on.  It's a renumbering of the
        # scheme in the book, used here to take advantage of the fact
        # that Python can use negative indices in lists.
        for l in xrange(2, self.num_layers):
            z = zs[-l]
            spv = sigmoid_prime_vec(z)
            delta = np.dot(self.weights[-l+1].transpose(), delta) * spv
```

```
            nabla_b[-l] = delta
            nabla_w[-l] = np.dot(delta, activations[-l-1].transpose())
        return (nabla_b, nabla_w)

    def evaluate(self, test_data):
        """Return the number of test inputs for which the neural
        network outputs the correct result. Note that the neural
        network's output is assumed to be the index of whichever
        neuron in the final layer has the highest activation."""
        test_results = [(np.argmax(self.feedforward(x)), y)
                        for (x, y) in test_data]
        return sum(int(x == y) for (x, y) in test_results)

    def cost_derivative(self, output_activations, y):
        """Return the vector of partial derivatives ¥partial C_x /
        ¥partial a for the output activations."""
        return (output_activations-y)

#### Miscellaneous functions
def sigmoid(z):
    """The sigmoid function."""
    return 1.0/(1.0+np.exp(-z))

sigmoid_vec = np.vectorize(sigmoid)

def sigmoid_prime(z):
    """Derivative of the sigmoid function."""
    return sigmoid(z)*(1-sigmoid(z))

sigmoid_prime_vec = np.vectorize(sigmoid_prime)
```

それでは、このコードがどれだけ良く手書き数字を認識できるかを確認していきましょう。まずはMNISTデータをダウンロードするところからはじめてみよう。ここではmnist_loader.pyを使用します。以下のコマンドをpythonシェルで実行してください。

```
>>> import mnist_loader
>>> training_data, validation_data, test_data = ¥
... mnist_loader.load_data_wrapper()
```

もちろん、これは独立したpythonプログラムとして実行できますが、ここまで従ってきたならば、pythonシェルで簡単に処理できるでしょう。

MNISTデータをダウンロードした後、私たちは30個の隠れニューロンをもつNetworkを設定します。私たちはnetworkと名前を付けた上記のpythonプログラムをインポートした後に、この処理を行います。

```
>>> import network
>>> net = network.Network([784, 30, 10])
```

最後に、30世代・ミニバッチサイズ10・訓練率$\eta = 3.0$の条件で、MNISTのtraining_dataから確率的勾配降下法を使用して学習します。

```
>>> net.SGD(training_data, 30, 10, 3.0, test_data=test_data)
```

もしこの文章を読みながらコードを実行しているならば、この計算は少々時間がかかるので注意してください。2014年における一般的なスペックのパソコンならば、訓練の1世代ごとに数分程度かかります。計算を実行しつつ、読み続けて、たまに計算結果を確認することをおすすめします。もしあなたが急いでいる場合は、あなたは世代数を減らすか、隠れニューロンの数を減らすか、訓練データの一部のみ使用することによって計算を速くすることができます。実際の商用コードはより速く計算が可能ですが、このpythonコードはニューラルネットワークを理解することを助けることが目的であるため、計算が早いわけではありません。もちろん、一度ニューラルネットワークを訓練すれば、私たちは多くのコンピュータープラットホーム上で非常に高速に実行することができます。例えば、ニューラルネットワークの重みとバイアスの良いセットがあれば、webブラウザのJavascriptや、携帯デバイスのアプリに移植し、実行するのは簡単です。それでは以下に、ニューラルネットワークのある訓練プロセスの一部の結果を示しましょう。この出力は訓練のエポックごとにニューラルネットワークを使用して適切に訓練データを認識できた数を表しています。最初の世代が終わったあとに10000個中の9129個が正しく認識できており、その後は増加し続けていることがわかります。

```
Epoch 0: 9129 / 10000
Epoch 1: 9295 / 10000
Epoch 2: 9348 / 10000
...
Epoch 27: 9528 / 10000
Epoch 28: 9542 / 10000
Epoch 29: 9534 / 10000
```

訓練されたネットワークは95%の分類率を有しており、ピーク性能は28世代での95.42%でした。この結果は、最初の試みとしては大変有望です。ネットワークをランダムな重みとバイアスによって初期化しているので、このコードを実行したとしても、上記で示した値と全く一緒になるとは限らないことに注意してください。この章では、3回の計算のうちのベストの解を示しています。

それでは、隠れニューロンの数を100個にして上記の実験を再計算してみましょう。この計算も同様に時間がかかりますので、計算を実行しつつ読み進めることが賢明です。(今回の場合、隠れニューロンの数が多いので各世代での計算時間がよりかかるので。)

```
>>> net = network.Network([784, 100, 10])
>>> net.SGD(training_data, 30, 10, 3.0, test_data=test_data)
```

予想通り、この計算では性能が96.59%に向上しました。少なくともこのケースでは、より多くの隠れ層を使用することでより良い結果を得ることが出来ます。*

*読者のフィードバックによると、この実験では性能の違いが報告されており、いくつかの計算結果ではか

なり性能が悪くなるようです。3章で紹介するテクニックを使用することで、計算ごとの予測性能の違いを劇的に減らすことができます。

もちろん、これらの精度を獲得するために、訓練のエポック数、ミニバッチのサイズ、学習率ηを具体的に選択しなくてはなりませんでした。上記のように、学習アルゴリズムによって学習するパラメータ(重みとバイアス)と区別するために、これらはニューラルネットワークのハイパーパラメータとして呼ばれています。もしハイパーパラメータを不適切に選択したならば、悪い結果を得ることになります。例えば、学習率$\eta = 0.001$を選んだとすると、

```
>>> net = network.Network([784, 100, 10])
>>> net.SGD(training_data, 30, 10, 0.001, test_data=test_data)
```

結果の改善の進捗は遅くなってしまいます。

```
Epoch 0: 1139 / 10000
Epoch 1: 1136 / 10000
Epoch 2: 1135 / 10000
...
Epoch 27: 2101 / 10000
Epoch 28: 2123 / 10000
Epoch 29: 2142 / 10000
```

しかしながら、ネットワークの予測性能はゆっくりと良くなっていくことがわかります。これは学習率を大きくすべきということを提案しているので、学習率$\eta = 0.01$にしてみましょう。この設定で計算した場合、より良い結果を得ることができます。この結果は、さらに学習率を増加させたほうが良いことを意味します。(もし変更を与えて性能が改善したならば、さらに変更を大きくさせてみてください。) 数回にわたって再計算を行ったならば、学習率は前の実験で使用した値に近い$\eta = 1.0$程度になるでしょう。最初にハイパーパラメータの不適切な選択をしたにもかかわらず、少なくともハイパーパラメータを選び方について性能を改善する情報を得ることができました。

たいていの場合、ニューラルネットワークをデバッグすることは困難なことであると言えます。ハイパーパラメータの初期の選択が悪く、ランダムノイズとほぼ同然の結果しか得られないときは、特に困難です。私たちは前に使用した隠れニューロンが30個のネットワークにおいて、学習率を$\eta = 100.0$に変更した場合を仮定してみましょう。

```
>>> net = network.Network([784, 30, 10])
>>> net.SGD(training_data, 30, 10, 100.0, test_data=test_data)
```

この設定は実際に度を超しているとは思っていましたが、やはり学習率が高すぎるようです。

```
Epoch 0: 1009 / 10000
Epoch 1: 1009 / 10000
Epoch 2: 1009 / 10000
Epoch 3: 1009 / 10000
...
Epoch 27: 982 / 10000
```

```
Epoch 28: 982 / 10000
Epoch 29: 982 / 10000
```

初めてこの問題に直面したことを、今想像してみてください。もちろん、私たちは前の実験を知っていて、学習率を下げることが正しいことだということを今は**知っています**。しかし、この問題に初めて直面する場合、出力結果は、何をすべきかを教えてはくれません。学習率だけでなく、ニューラルネットワークの他の全ての側面について心配するかもしれません。ネットワークが学習することを難しくさせるような方法で重みとバイアスを初期化したのかもしれないと疑うかもしれません。それとも、意味のある学習をするための十分な訓練データを持っていないと思うかもしれません。十分なエポック数を計算していないのかもしれない？手書き数字を認識するために学習することはニューラルネットワークでは不可能と思うかもしれません。学習率が**低すぎる**かもしれない、それとも高すぎる？はじめて、この問題に直面する際は、常に何もわからない状態にあるのです。

これらの疑問を取り除くためのニューラルネットワークのデバッグとしてのレッスンは些細な問題ではありません、そして通常のプログラミングに関して言えば、関係した技術があります。ニューラルネットワークから良い結果を得るためのデバッグ技術を学ぶ必要があります。より一般的に言えば、私たちは良いハイパーパラメータと良いアーキテクチャを選択するためのヒューリスティック技術を開発する必要があります。この本を通して、上記のハイパーパラメータの設定方法を含むこれらの技術について解説します。

Exercise

- 隠れ層がない入力層と出力層のみの2層のネットワークを作ってみてください。それぞれニューロンは、784個と10個です。そして、確率的勾配法で学習させてみてください。どんな分類精度を達成できるでしょうか。

以前、MNISTデータのロード方法の詳細について説明を省略していました。かなり簡単ではありますが、念のためにコードを以下に載せました。MNISTデータを格納するために使われるデータ構造は、資料内の解説にある通りで、Numpyの`ndarray`オブジェクトです。(もし`ndarray`に馴染みがない方は、ベクトルとして考えてください。)

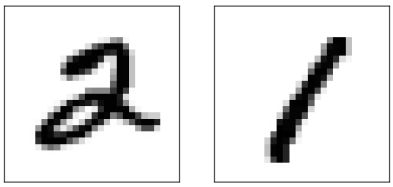
```
"""
mnist_loader
~~~~~~~~~~~~

A library to load the MNIST image data.  For details of the data
structures that are returned, see the doc strings for ``load_data``
```

```
    and ``load_data_wrapper``.  In practice, ``load_data_wrapper`` is the
    function usually called by our neural network code.
    """

    #### Libraries
    # Standard library
    import cPickle
    import gzip

    # Third-party libraries
    import numpy as np

    def load_data():
        """Return the MNIST data as a tuple containing the training data,
        the validation data, and the test data.

        The ``training_data`` is returned as a tuple with two entries.
        The first entry contains the actual training images.  This is a
        numpy ndarray with 50,000 entries.  Each entry is, in turn, a
        numpy ndarray with 784 values, representing the 28 * 28 = 784
        pixels in a single MNIST image.

        The second entry in the ``training_data`` tuple is a numpy ndarray
        containing 50,000 entries.  Those entries are just the digit
        values (0...9) for the corresponding images contained in the first
        entry of the tuple.

        The ``validation_data`` and ``test_data`` are similar, except
        each contains only 10,000 images.

        This is a nice data format, but for use in neural networks it's
        helpful to modify the format of the ``training_data`` a little.
        That's done in the wrapper function ``load_data_wrapper()``, see
        below.
        """
        f = gzip.open('../data/mnist.pkl.gz', 'rb')
        training_data, validation_data, test_data = cPickle.load(f)
        f.close()
        return (training_data, validation_data, test_data)

    def load_data_wrapper():
        """Return a tuple containing ``(training_data, validation_data,
        test_data)``. Based on ``load_data``, but the format is more
        convenient for use in our implementation of neural networks.

        In particular, ``training_data`` is a list containing 50,000
        2-tuples ``(x, y)``.  ``x`` is a 784-dimensional numpy.ndarray
        containing the input image.  ``y`` is a 10-dimensional
        numpy.ndarray representing the unit vector corresponding to the
        correct digit for ``x``.

        ``validation_data`` and ``test_data`` are lists containing 10,000
        2-tuples ``(x, y)``.  In each case, ``x`` is a 784-dimensional
        numpy.ndarry containing the input image, and ``y`` is the
        corresponding classification, i.e., the digit values (integers)
        corresponding to ``x``.

        Obviously, this means we're using slightly different formats for
        the training data and the validation / test data.  These formats
        turn out to be the most convenient for use in our neural network
        code."""
        tr_d, va_d, te_d = load_data()
```

```
    training_inputs = [np.reshape(x, (784, 1)) for x in tr_d[0]]
    training_results = [vectorized_result(y) for y in tr_d[1]]
    training_data = zip(training_inputs, training_results)
    validation_inputs = [np.reshape(x, (784, 1)) for x in va_d[0]]
    validation_data = zip(validation_inputs, va_d[1])
    test_inputs = [np.reshape(x, (784, 1)) for x in te_d[0]]
    test_data = zip(test_inputs, te_d[1])
    return (training_data, validation_data, test_data)

def vectorized_result(j):
    """Return a 10-dimensional unit vector with a 1.0 in the jth
    position and zeroes elsewhere.  This is used to convert a digit
    (0...9) into a corresponding desired output from the neural
    network."""
    e = np.zeros((10, 1))
    e[j] = 1.0
    return e
```

このプログラムはかなり良い結果を得られたと言いました。これはどういう意味でしょうか。何と比較して良いと言っているのでしょうか。お互いに比較したり、何がよい実行結果なのかを理解したりするために、いくつかのニューラルネットワークではない単純な性能基準を持つことは有益です。もちろん、全ての中で最もシンプルな基準は数字をランダムに推測するものです。それは、性能は10%程度になるでしょう。私たちは、それよりも遥かに良い性能を持っています。

取るに足らなくない基準とはなんでしょうか。それでは、とてもシンプルなアイデアに挑戦してみましょう。画像がどれくらいか**暗いか**について見ています。例えば、「2」の画像は「1」の画像に比べて、より暗い画像となります。なぜならば、下記の例を見ればわかる通り、多くのピクセルが黒く塗りつぶされているからです。



これは、各数字($0, 1, 2, \ldots, 9$)の黒色を持つピクセルの平均を計算するために訓練データを使用することを提案しています。新しい画像が示されたとき、画像がどれだけ黒いかを計算し、最も近い黒色を持つピクセルの平均の数字として区別します。これは簡単な手順であり、コーディングも簡単です。私は明示的にコードを書きませんが、GitHubのGitHub repositoryに公開しておきます。しかし、この方法では、10000枚の訓練データのうち2225枚を適切に区別することができ、つまり22.25%の精度、ランダム推測に比べて大きな改善をもたらします。

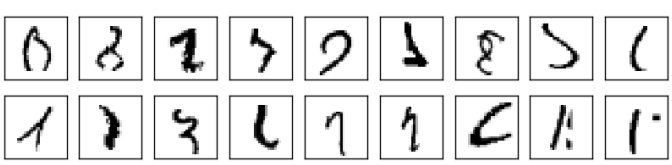
予測精度20から50%程度の他のアイデアを見つけることは難しくありません。もしあなたが少し頑張れば、予測精度50%に到達できるでしょう。

しかしさらに高精度を取得するためには、機械学習アルゴリズムを使用することが手助けになるでしょう。良く知られている手法の一つであるサポートベクターマシン(**SVM**)について挑戦してみましょう。もしSVMに馴染みがなかったとしても、心配しないでください。SVMの働きについて詳細に理解する必要はありません。代わりに、私たちはLIBSVMとして知られているSVMの高速なCベースのライブラリのpythonインターフェースであるscikit-learnというライブラリを使用します。

もしscikit-learnのSVM分類器のデフォルト設定で計算したとすれば、10000個の画像中の9435個を適切に分類できます。(コードはここにあります。) 黒色の平均値に基づく分類から考えると、とても大きな改善です。確かに、SVMはだいたいニューラルネットワークと同じくらいか、ちょっと悪いくらいの性能を持っています。後の章では、私たちはSVMをはるかに超える良い性能を得るために、ニューラルネットワークを改良する新技術を紹介します。

話はこれで終わりではありません。10000個中の9435個の適切な分類は、scikit-learnのデフォルト設定のSVMによるものでした。SVMは調整可能なパラメータをいくつか持っていて、さらに性能の良い判別が行うことができるパラメータを探すことができる可能性があります。この探索を明示的には行いませんが、代わりにもし詳細を知りたいならばAndreas Muellerのブログを確認してください。Muellerは、SVMのパラメータを最適化することで、98.5%の予測性能に到達していることを示しています。言い換えれば、適切にチューニングされたSVMでは間違いを70個しかしないことになります。これは、かなり良い性能です。ニューラルネットワークはこれより良い推定ができるでしょうか。

実際には可能です。現在、よく設計されたニューラルネットワークは、SVMを含めて他のMNISTの識字アルゴリズムの中で最も優れています。2014年現在の記録では、10000個中9979個の画像を適切に分類することが出来ています。これは、Li Wan、Matthew Zeiler,、Sixin Zhang、Yann LeCun、Rob Fergusによって行われたものです。この本の中の後半で、彼らが使用した技術の多くを見ることができます。その性能のレベルは人間とほぼ同等であり、おそらくより良いものです。なぜならば、人間が自信を持って認識することさえ難しいMNIST画像がいくつかあるからです。例えば以下のようなものです。



これらを分類するのは難しいと認めることを確信しています。MNISTデータセットの中でこのような画像があるのにもかかわらず、ニューラルネットワークは10000個の画像の中の21個以外は適切に分類できることは驚くべきことです。プログラミングをするとき、たいていはMNISTの識字を理解するような複雑な問題を解くことは、高度なアルゴリズムが必要だと考えています。しかし、Wanらのペーパーの中で言及したニューラルネットワークはかなりシンプルなアルゴリズムを意味しています。私たちがこの章で見てきたアルゴリズムのバリエーションも含んでいます。

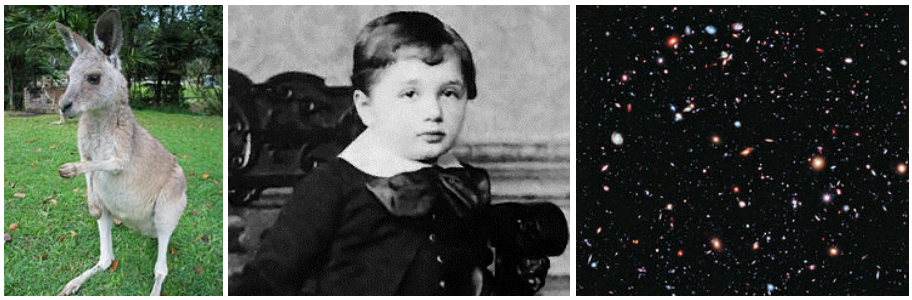
高度なアルゴリズム ≤ シンプルな学習アルゴリズム + 良い訓練データ

Deep Learningに向けて

ニューラルネットワークは印象的な性能を提供していますが、その性能はやや神秘的です。ネットワークの重みとバイアスが自動的に発見されました。これはつまり、ネットワークがどのように機能しているかについて説明できないことを意味します。ネットワークが手書き数字を分類する原理を理解する方法を見つけることができるでしょうか？ そして、そのような原理を考え、さらによりよく出来るでしょうか？

この質問により厳密に答えるために、向こう20〜30年でニューラルネットワークは人工知能(AI)になるだろうと考えてください。私たちは、そんな賢いネットワークがどのように働くかを理解できるでしょうか。もしかしたら、私たちが把握していない重みとバイアスを使ったネットワークは、不透明なものになるかもしれない。というのも、ネットワークは自動的に学習してきたからです。初期のAI研究者は、AIを構築するための努力によって知能の原理（人間の脳機能のような）を理解できるようになる、そんなことを望んでいました。しかし、私たちは脳だけでなく、人工知能の働きさえ理解せずに終わってしまうことになるかもしれません！

これらの問題に答えるために、この章の最初に説明した人工ニューロンのパーセプトロンについて振り返ってみましょう。画像が人の顔を示しているか否かを判断したいとします。



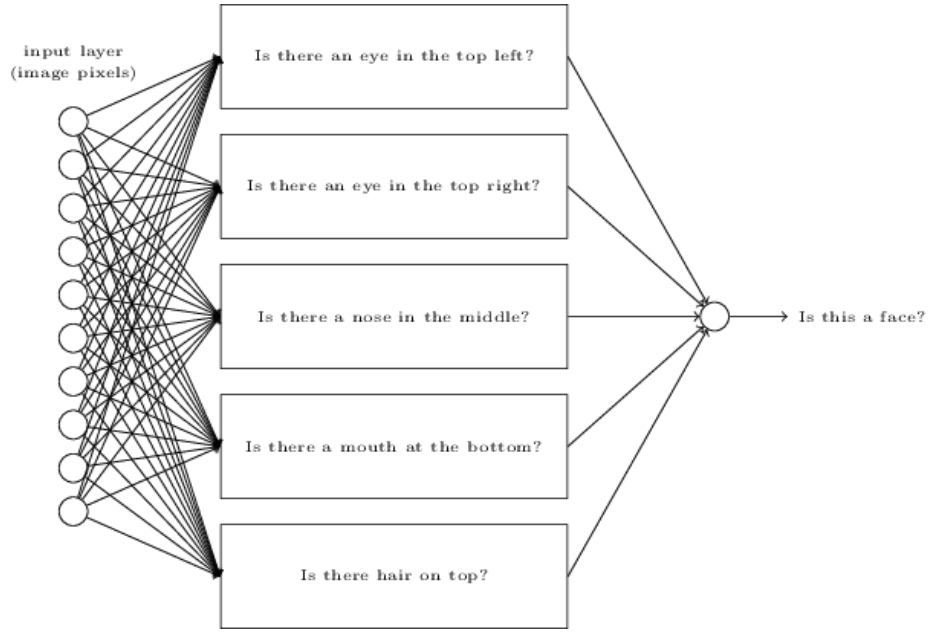
Credits: 1. Ester Inbar. 2. Unknown. 3. NASA, ESA, G. Illingworth, D. Magee, and P. Oesch (University of California, Santa Cruz), R. Bouwens (Leiden University), and the HUDF09 Team. Click on the images for more details.

この問題に対しても、手書き文字認識と同じ方法で取り組むことができます。つまり、ニューラルネットワークの入力として画像のピクセルを使い、1つの出力ニューロンによって「顔である」か「顔でない」かを判定させるのです。

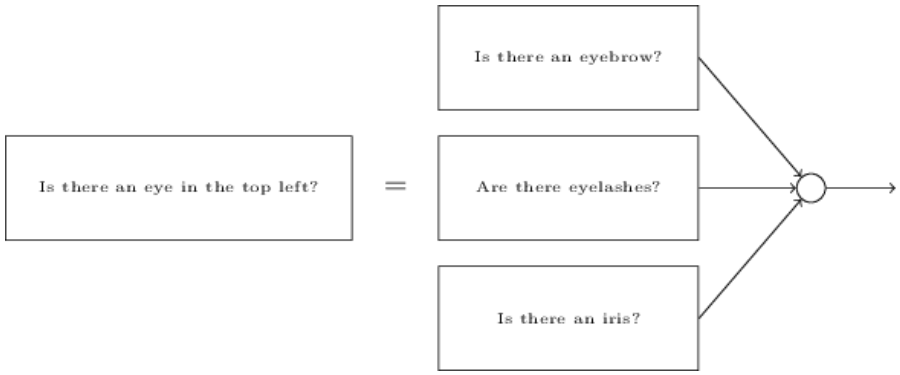
では考えてみましょう。ただし、学習アルゴリズムは使いません。その代わりに、ネットワークを手動で設計し、適切な重みとバイアスを設定していきます。どのように取り組めばよいでしょうか？ニューラルネットワークを一瞬全て忘れるとして、私たちが使うことのできるヒューリスティクスによって問題を小さな問題に分割します。「画像の左上に目はあるか」「画像の右上に目はあるか」「画像の中心に鼻はあるか」「画像の中央下に口はあるか」「髪の毛は上のほうにあるか」などなど。

これらの質問の答えのいくつかが「YES」や「おそらくYES」だとすれば、その画像は顔であると言えるでしょう。逆に、質問の答えがほとんど「NO」だとすれば、それはおそらく顔ではないでしょう。

もちろん、これはかなり荒いヒューリスティクスであり、多くの欠点を持ちます。例えば、禿げた人を考えた場合は、彼らには髪の毛がありません。私たちは、顔の一部のみだったり、顔に角度がついていたり、顔の一部が隠されていたりしても、顔だと判断できます。それでも、このヒューリスティクスによって次のような示唆を得ることができます。つまり、小さい問題をニューラルネットワークを使って解くことができるなら、それらネットワークを組み合わせることで、顔判定のためのネットワークを構築することができる、ということです。以下に、ありうりそうなアーキテクチャを示しましょう。これは、サブネットワークを長方形で表しています。ただし、ここでは顔認識問題を解くための現実的なアプローチは意図していないことに注意してください。むしろこれは、どのようにネットワークが機能するかについて直感的な理解を促してくれます。



サブネットワーク自体も小さく分解できる、というのはいかにもありそうなことです。それでは次のような質問について検討してみましょう。「画像の左上に目があるか？」これは次のような質問にさらに分けることができます。「眉毛はあるか？」「まつ毛はあるか？」「眼球の光彩はあるか？」などです。もちろんこれらの質問は、実際は位置の情報を含んでいるべきです。「画像の右上に眉毛はあるか？それは光彩の上にあるか？」というように。このような感じですが、シンプルにはしておきましょう。このようにして、「左上に目はあるか？」は次のように分解できます。



こういった質問は、複数の層を介して、さらにさらに分解して行くことが可能です。究極的には、単一のピクセルレベルで簡単に答えられるようなシンプルな質問に答えるサブネットワークで作業することになります。例えばその質問は、画像内のある特定の点での、シンプルな形状の有り・無しかもしれません。その手の質問は、画像のピクセルにそのまま直接接続された、単一のニューロンによって答えることができます。

最終的な結果は、非常に込み入った問題（「画像は顔を表しているか否か」のような）を、単一のピクセルレベルで答えられる問題に分解したネットワークになります。以上の結果を得るためには、入力画像についてとても簡単で特定の質問に答える初期の層と、より複雑で抽象的な概念の階

層を構築している後半の層を含む多くの層を通ることが必要です。多くの隠れ層（2つかそれ以上）を含む多層構造のネットワークは、**ディープニューラルネットワーク**と呼ばれています。

もちろん、どのように再帰的にサブネットワークに分けていくのかについては、述べていません。ネットワーク内の重みとバイアスを手動で設計するのは、実用的ではありません。代わりに訓練データから、自動的に重みとバイアス（さらに言えば、概念の階層構造まで）を習得できるような学習アルゴリズムを使います。1980年代と1990年代の研究者は、確率的勾配降下法と誤差逆伝播法をディープネットワークの訓練に使用してみようとしました。残念ながら、いくつかの特別なアーキテクチャを除いて、彼らは良い結果を得られませんでした。学習はするのですが、とても遅く、現実的に使用できませんでした。

2006年になってようやく、ディープニューラルネットを学習可能にする一連の技術が開発されました これらの学習技術は確率的勾配降下法と誤差逆伝播法に基づいてはいますが、新しいアイデアが追加されています。これらの技術は、より深く（そしてより大きな）ネットワークを訓練可能にし、現在では5〜10層のネットワークが当然のように訓練されています。そして、浅いニューラルネットワーク（特に隠れ層が1層のみの場合）よりも、多くの問題解決において非常に良くなっていることがわかりました。もちろんこれは、ディープネットワークが、概念の複雑な階層構造を構築できるからです。従来のプログラム言語で複雑なプログラムを作るときによく使う、モジュール方式のデザインと考え方に少し似ています。ディープネットワークと浅いネットワークの関係は、関数作成と呼び出しが可能なプログラム言語と、そのような能力を持たない言語の関係に少し似ています。抽象化は、従来のプログラミングにおけるものとは異なる形式を取りますが、それは極めて重要なことなのです。

In academic work, please cite this book as: Michael A. Nielsen, "Neural Networks and Deep Learning", Determination Press, 2014

Last update: Tue Sep 2 09:19:44 2014

CHAPTER 2

逆伝播の仕組み

前章では、勾配降下法を用いてニューラルネットワークが重みとバイアスをどのように学習するかを説明しました。しかし、その説明にはギャップがありました。具体的には、コスト関数の勾配をどのように計算するかを議論していません。これはとても大きなギャップです！ 本章では、**逆伝播**と呼ばれる、コスト関数の勾配を高速に計算するアルゴリズムを説明します。

逆伝播アルゴリズムはもともと1970年代に導入されました。しかし逆伝播が評価されたのは、David Rumelhart・Geoffrey Hinton・Ronald Williams による1986年の著名な論文が登場してからでした。その論文では、逆伝播を用いると既存の学習方法よりもずっと早く学習できる事をいくつかのニューラルネットワークに対して示し、それまでニューラルネットワークでは解けなかった問題が解ける事を示しました。今日では、逆伝播はニューラルネットワークを学習させる便利なアルゴリズムです。

本章は他の章に比べて数学的に難解です。よほど数学に対し熱心でなければ、本章を飛ばして、逆伝播を中身を無視できるブラックボックスとして扱いたくなるかもしれません。では、なぜ時間をかけて逆伝播の詳細を勉強するのでしょうか？

その理由はもちろん理解のためです。逆伝播の本質はコスト関数Cのネットワークの重みw（もしくはバイアスb）に関する偏微分$\partial C/\partial w$（$\partial C/\partial b$）です。偏微分をみると、重みとバイアスを変化させた時のコスト関数の変化の度合いがわかります。偏微分の式は若干複雑ですが、そこには美しい構造があり、式の各要素には自然で直感的な解釈を与える事ができます。そうです、逆伝播は単なる高速な学習アルゴリズムではありません。逆伝播をみることで、重みやバイアスを変化させた時のニューラルネットワーク全体の挙動の変化に関して深い洞察が得られます。逆伝播を勉強する価値はそこにあるのです。

そうは言うものの、本章をざっと読んだり、読み飛ばして次の章に進んでも大丈夫です。この本は逆伝播をブラックボックスとして扱っても他の章を理解できるように書いています。もちろん次章以降で本章の結果を参照する部分はあります。しかし、その参照部分の議論をすべて追わなくても、主な結論は理解できるはずです。

著者と共にこの本を作り出してくださったサポーターの皆様に感謝いたします。また、バグ発見者の殿堂に名を連ねる皆様にも感謝いたします。また、日本語版の出版にあたっては、翻訳者の皆様に深く感謝いたします。

この本は目下のところベータ版で、開発続行中です。エラーレポートは mn@michaelnielsen.org まで、日本語版に関する質問は muranushi@gmail.com までお送りください。その他の質問については、まずはFAQをごらんください。

Resources

Code repository

Mailing list for book announcements

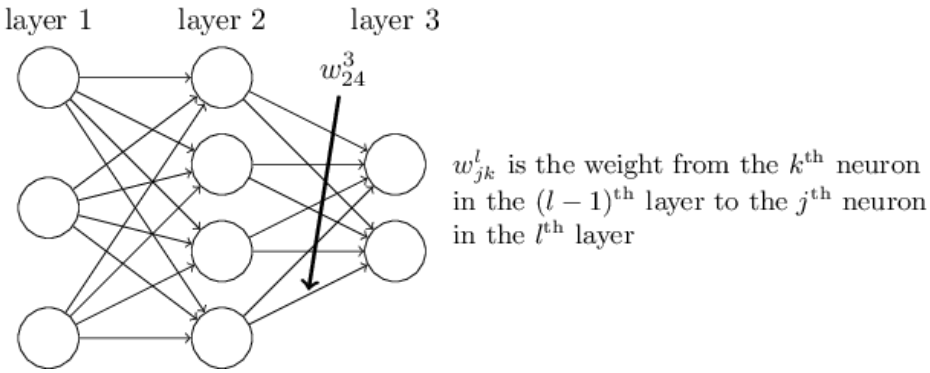
Michael Nielsen's project announcement mailing list

ウォーミングアップ：ニューラルネットワークの出力の行列を用いた高速な計算

著：Michael Nielsen / 2014年9月-12月
訳：「ニューラルネットワークと深層学習」翻訳プロジェクト

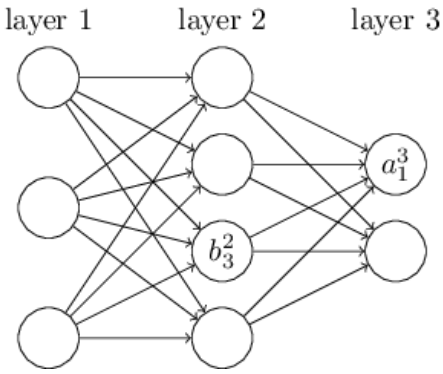
逆伝播を議論する前に、ニューラルネットワークの出力を高速に計算する行列を用いたアルゴリズムでウォーミングアップしましょう。私達は 前章の最後のあたり で既にこのアルゴリズムを簡単に見ています。しかしその時はざっと書いていたので、ここで立ち戻って詳しく説明しようと思います。特にこれまで説明して慣れた文脈で逆伝播で使用する記号に慣れるのに、このウォーミングアップは良い方法です。

ニューラルネットワーク中の重みを曖昧性なく指定する表記方法からまず始めましょう。w^l_{jk}で、$(l-1)$番目の層のk番目のニューロンから、l番目の層のj番目のニューロンへの接続に対する重みを表します。例えば下図は、2番目の層の4番目のニューロンから、3番目の層の2番目のニューロンへの接続の重みを表します。



この表記方法は最初は面倒で、使いこなすのにある程度の練習が必要かもしれません。しかし、少し頑張ればこの表記方法は簡単で自然だと感じるようになるはずです。この表記方法で若干曲者なのが、jとkの順番です。jを入力ニューロン、kを出力ニューロンとする方が理にかなっていると思うかもしれませんが、実際には逆にしています。奇妙なこの記述方法の理由は後程説明します。

ニューラルネットワークのバイアスと活性についても似た表記方法を導入します。具体的には、b^l_jでl番目の層のj番目のニューロンのバイアスを表します。また、a^l_jでl番目の層のj番目のニューロンの活性を表します。下図はこれらの表記方法の利用例です。



これらの表記を用いると、l番目の層のj番目のニューロンの活性a_j^lは、$(l-1)$番目の層の活性と以下の式で関係付けられます（式 (4) と前章の周辺の議論を比較してください）

$$a_j^l = \sigma\left(\sum_k w_{jk}^l a_k^{l-1} + b_j^l\right). \tag{23}$$

ここで、和は$(l-1)$番目の層の全てのニューロンkについて足しています。この式を行列で書き直すため、各層lに対し**重み行列**w^lを定義します。重み行列w^lの各要素はl番目の層のニューロンを終点とする接続の重みです。すなわち、j行目k列目の要素をw_{jk}^lとします。同様に、各層lに対し、**バイアスベクトル**b^lを定義します。おそらく想像できると思いますが、バイアスベクトルの要素はb_j^l達で、l番目の層の各ニューロンに対し$\mathbf{1}$つの行列要素が伴います。最後に、活性ベクトルa^lを活性a_j^l達で定義します。

(23) を行列形式に書き直すのに必要な最後の要素は、σなどの関数のベクトル化です。ベクトル化は既に前章で簡単に見ました。要点をまとめると、σのような関数をベクトルvの各要素に適用したいというのがアイデアです。このような各要素への関数適用には$\sigma(v)$という自然な表記を用います。つまり、$\sigma(v)$の各要素は$\sigma(v)_j = \sigma(v_j)$です。例えば$f(x) = x^2$とすると、次のようになります。

$$f\left(\begin{bmatrix}2\\3\end{bmatrix}\right) = \begin{bmatrix}f(2)\\f(3)\end{bmatrix} = \begin{bmatrix}4\\9\end{bmatrix}. \tag{24}$$

すなわち、ベクトル化したfはベクトルの各要素を$\mathbf{2}$乗します。

この表記方法を用いると、式 (23) は次のような美しくコンパクトなベクトル形式で書けます。

$$a^l = \sigma(w^l a^{l-1} + b^l). \tag{25}$$

この表現を用いると、ある層の活性とその前の層の活性との関係を俯瞰できます。我々が行っているのは活性に対し重み行列を掛け、バイアスベクトルを足し、最後にσ関数を適用するだけです。この見方はこれまで

*ところで、先ほどのw_{jk}^lという奇妙な表記を用いる動機はこの式に由来します。もし、jを入力ニューロンに用い、kを出力ニューロンに用いたとすると、式

のニューロン単位での見方よりも簡潔で、添字も少なくて済みます。議論の正確性を失う事なく添字地獄から抜け出せる方法と考えると良いでしょう。さらに、この表現方法は実用上も有用です。というのも、多くの行列ライブラリでは高速な行列掛算・ベクトル足し算・関数のベクトル化の実装が提供されているからです。実際、前章のコード では、ネットワークの挙動の計算にこの表現を暗に利用していました。

(25) は重み行列をそれの転置行列に置き換えなければなりません。些細な変更ですが、煩わしい上に「重み行列を掛ける」と簡単に言ったり(もしくは考えたり)できなくなってしまいます。

a^lの計算のために式 (25) を利用する時には、途中で$z^l \equiv w^l a^{l-1} + b^l$を計算しています。この値は後の議論で有用なので名前をつけておく価値があります。z^lをl番目の層に対する**重みつき入力**と呼ぶことにします。本章の以降の議論では重みつき入力z^lを頻繁に利用します。式 (25) をしばしば重み付き入力を用いて$a^l = \sigma(z^l)$とも書きます。z^lの要素は$z^l_j = \sum_k w^l_{jk} a^{l-1}_k + b^l_j$と書ける事にも注意してください。つまり、$z^l_j$は$l$番目の層の$j$番目のニューロンが持つ活性関数へ与える重みつき入力です。

コスト関数に必要な2つの仮定

逆伝播の目標はニューラルネットワーク中の任意の重みwまたはバイアスbに関するコスト関数Cの偏微分、すなわち$\partial C/\partial w$と$\partial C/\partial b$の計算です。逆伝播が機能するには、コスト関数の形について2つの仮定を置く必要があります。それらの仮定を述べる前に、コスト関数の例を念頭に置くのが良いでしょう。前章でも出てきた2乗コスト関数(参考:式(6))をここでも考えます。前章の記法では、2乗コスト関数は以下の様な形をしていました

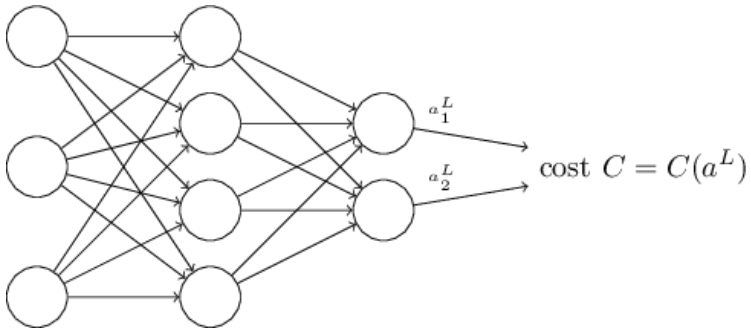
$$C = \frac{1}{2n} \sum_x \|y(x) - a^L(x)\|^2. \tag{26}$$

ここで、nは訓練例の総数、和は個々の訓練例xについて足しあわせたもの、$y = y(x)$は対応する目標の出力、Lはニューラルネットワークの層数、$a^L = a^L(x)$はxを入力した時のニューラルネットワークの出力のベクトルです。

では、逆伝播を適用するために、コスト関数Cに置く仮定はどのようなものでしょうか。1つ目の仮定はコスト関数は個々の訓練例xに対するコスト関数C_xの平均 $C = \frac{1}{n} \sum_x C_x$ で書かれているという事です。2乗コスト関数ではこの仮定が成立しています。それには1つの訓練例に対するコスト関数を$C_x = \frac{1}{2}\|y - a^L\|^2$とすれば良いです。この仮定はこの本で登場する他のコスト関数でも成立しています。

この仮定が必要となる理由は、逆伝播によって計算できるのは個々の訓練例に対する偏微分$\partial C_x/\partial w$、$\partial C_x/\partial b$だからです。コスト関数の偏微分$\partial C/\partial w$、$\partial C/\partial b$は全訓練例についての平均を取ることで得られます。この仮定を念頭に置き、私達は訓練例xを1つ固定していると仮定し、コストC_xを添字xを除いてCと書くことにします。最終的に除いたxは元に戻しますが、当面は記法が煩わしいので暗にxが書かれていると考えます。

コスト関数に課す2つ目の仮定は、コスト関数はニューラルネットワークの出力の関数で書かれているという仮定です。



例えば、2乗誤差関数はこの要求を満たしています、それは1つの訓練例xに対する誤差は以下のように書かれるためです

$$C = \frac{1}{2}\|y - a^L\|^2 = \frac{1}{2}\sum_j (y_j - a^L_j)^2. \tag{27}$$

もちろんこのコスト関数は目標とする出力yにも依存しています。コスト関数をyの関数とみなさない事を不思議に思うかもしれません。しかし、訓練例xを固定する事で、出力yも固定している事に注意してください。つまり、出力yは重みやバイアスをどのように変化させた所で変化させられる量ではなく、ニューラルネットが学習するものではありません。ですので、Cを出力の活性a^L単独の関数とみなし、yは関数を定義するための単なるパラメータとみなすのは意味のある問題設定です。

アダマール積 $s \odot t$

逆伝播アルゴリズムは、ベクトルの足し算やベクトルと行列の掛け算など、一般的な代数操作に基づいています。しかし、その中で1つあまり一般的ではない操作があります。sとtが同じ次元のベクトルとした時、$s \odot t$を2つのベクトルの**要素ごと**の積とします。つまり、$s \odot t$の要素は$(s \odot t)_j = s_j t_j$です。例えば、

$$\begin{bmatrix} 1 \\ 2 \end{bmatrix} \odot \begin{bmatrix} 3 \\ 4 \end{bmatrix} = \begin{bmatrix} 1 * 3 \\ 2 * 4 \end{bmatrix} = \begin{bmatrix} 3 \\ 8 \end{bmatrix} \tag{28}$$

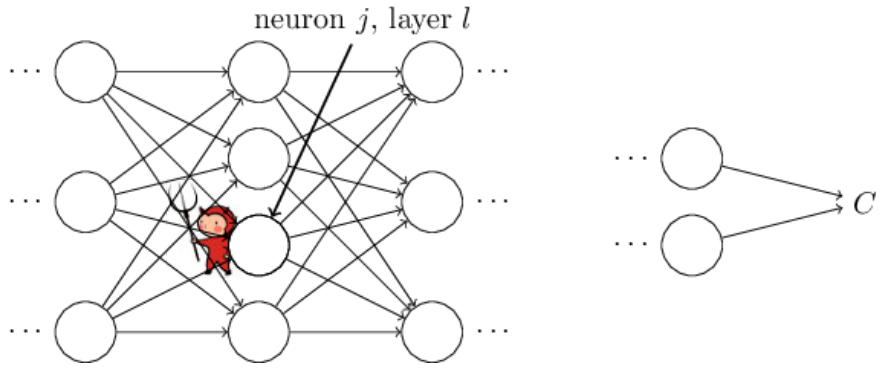
です。この種の要素ごとの積はしばしば**アダマール積**、もしくは**シューア積**と呼ばれます。私達はアダマール積と呼ぶことにします。よく出来

た行列ライブラリにはアダマール積の高速な実装が用意されており、逆伝播を実装する際に手軽に利用できます。

逆伝播の基礎となる4つの式

逆伝播は重みとバイアスの値を変えた時にコスト関数がどのように変化するかを把握する方法です。これは究極的には$\partial C/\partial w^l_{jk}$と$\partial C/\partial b^l_j$とを計算する事を意味します。これらの偏微分を計算する為にまずは中間的な値δ^l_jを導入します。この値はl番目の層のj番目のニューロンの**誤差**と呼びます。逆伝播の仕組みを見るとδ^l_jを計算手順とδ^l_jを$\partial C/\partial w^l_{jk}$や$\partial C/\partial b^l_j$と関連づける方法が得られます。

誤差の定義方法を理解する為にニューラルネットワークの中にいる悪魔を想像してみましょう。



悪魔はl番目の層のj番目のニューロンに座っているとします。ニューロンに入力が入ってきた時、悪魔はニューロンをいじって、重みつき入力に小さな変更Δz^l_jを加えます。従って、ニューロンは$\sigma(z^l_j)$の代わりに、$\sigma(z^l_j+\Delta z^l_j)$を出力します。この変化はニューラルネット中の後段の層に伝播し、最終的に全体のコスト関数の値は$\frac{\partial C}{\partial z^l_j}\Delta z^l_j$だけ変化します。

ここで、この悪魔は善良な悪魔で、コスト関数を改善する、つまりコストを小さくするようなΔz^l_jを探そうとするとします。$\frac{\partial C}{\partial z^l_j}$が大きな値（正でも負も良いです）であるとします。すると、$\frac{\partial C}{\partial z^l_j}$と逆の符号の$\Delta z^l_j$を選ぶことで、この悪魔はコストをかなり改善させられます。逆に、もし$\frac{\partial C}{\partial z^l_j}$が$0$に近いと悪魔は重みつき入力$z^l_j$を摂動させてもコストをそれほどは改善できません。悪魔が判断できる範囲においてはニューロンは既に最適に近い状態だと言えます*。つまり、ヒューリスティックには、$\frac{\partial C}{\partial z^l_j}$はニューラルネットワークの誤差を測定しているという意味を与える事ができます。

*もちろんこれが正しいのはΔz^l_jが小さい場合に限ってです。悪魔は微小な変化しか起こせないと仮定しています

この話を動機として、l番目の層のj番目のニューロンの誤差δ^l_jを以下のように定義します

$$\delta^l_j \equiv \frac{\partial C}{\partial z^l_j}. \tag{29}$$

. 慣習に沿って、δ^lでl番目の層の誤差からなるベクトルを表します。逆伝播により、各層でのδ^lを計算し、これらを真に興味のある$\partial C/\partial w^l_{jk}$や$\partial C/\partial b^l_j$と関連付けることができます。

悪魔はなぜ重みつき入力z^l_jを変えようとするのかを疑問に思うかもしれません。確かに、出力活性a^l_jを変化させ、その結果の$\frac{\partial C}{\partial a^l_j}$を誤差の指標として用いる方が自然かもしれません。実際そのようにしても、以下の議論は同じように進められます。しかし、やってみるとわかるのですが、誤差逆伝播の表示が数学的に若干複雑になってしまいます。ですので、我々は誤差の指標として$\delta^l_j = \frac{\partial C}{\partial z^l_j}$を用いることにします*。

*MNISTのような分類問題では、誤差(error)という言葉はしばしば誤分類の割合を意味します。例えばニューラルネットが96.0%の数字を正しく分類できたとしたら、"error"は4.0%です。もちろん、これはδベクトルとは全く異なる意味です。実際の文脈ではどちらの意味かで迷うことはないでしょう。

攻略計画 逆伝播は4つの基本的な式を基礎とします。これらを組み合わせると、誤差δ^lとコスト関数の勾配を計算ができます。以下でその4つの式を挙げていきますが、1点注意があります:これらの式の意味をすぐに消化できると期待しない方が良いでしょう。そのように期待するとがっかりするかもしれません。逆伝播は内容が豊富であり、これらの式は相当の時間と忍耐がかけて徐々に理解できていくものです。幸いなことに、ここで辛抱しておくと後々何度も報われることになります。この節の議論はスタート地点に過ぎませんが、逆伝播の式を深く理解する過程の中で役に立つもののはずです。

誤差逆伝播の式をより深く理解する方法の概略は以下の通りです。まず、これらの式の手短な証明を示します。この証明を見ればなぜこれらの式が正しいのかを理解しやすくなります。その後、これらの式を擬似コードで書き直し、その擬似コードをどのように実装できるかを実際のPythonのコードで示します。本章の最後の節では、誤差逆伝播の式の意味を直感的な図で示し、ゼロからスタートしてどのように誤差逆伝播を発見するかを見ていきます。その道中で、我々は何度も4つの基本的な式に立ち戻ります。理解が深まるにつれ、これらの式が快適で、美しく自然なものとさえ思えるようになるはずです。

出力層での誤差δ^Lに関する式: δ^Lの各要素は

$$\delta^L_j = \frac{\partial C}{\partial a^L_j} \sigma'(z^L_j). \tag{BP1}$$

です。これはとても自然な表式です。右辺の第1項の$\partial C/\partial a^L_j$はコストが$j$番目の出力活性の関数としてどの程度敏感に変化するかの度合いを測っています。例えば、Cが出力層の特定のニューロン(例えばj番目とします)にそれほど依存していなければ、我々の期待通りδ^L_jは小さくなりま

す。一方、右辺の第2項の$\sigma'(z^L_j)$は活性関数σがz^L_jの変化にどの程度敏感に反応するかの度合いを表しています。

ここで注目すべきなのは (BP1) 中の全ての項が簡単に計算できる事です。ニューラルネットワークの挙動を計算する間にz^L_jを計算でき、さらに若干のオーバーヘッドを加えれば$\sigma'(z^L_j)$も計算できます。従って、第2項は計算できます。第1項に関してですが、$\partial C/\partial a^L_j$の具体的な表式はもちろんコスト関数の形に依存します。しかし、コスト関数が既知ならば$\partial C/\partial a^L_j$を計算するのは難しくありません。例えば、2乗誤差コスト関数を用いた場合、$C = \frac{1}{2}\sum_j (y_j - a_j)^2$なので、$\partial C/\partial a^L_j = (a_j - y_j)$という簡単に計算できる式が得られます。

式(BP1) はδ^Lの各要素に対する表式です。この表式自体は悪くはないのですが、逆伝播で欲しい行列を用いた表式ではありません。この式を行列として書き直すのは容易で、以下の様に書けます

$$\delta^L = \nabla_a C \odot \sigma'(z^L). \tag{BP1a}$$

ここで、$\nabla_a C$は偏微分$\partial C/\partial a^L_j$を並べたベクトルです。$\nabla_a C$は出力活性に対する$C$の変化率とみなせます。(BP1a) と (BP1) は同値である事はすぐにわかります。ですので、以下では両者の式を参照するのに (BP1) を用いる事にします。例として、2乗誤差コスト関数の例では$\nabla_a C = (a^L - y)$です。従って行列形式の (BP1) は以下のようになります。

$$\delta^L = (a^L - y) \odot \sigma'(z^L). \tag{30}$$

見ての通り、この表式内の全ての項がベクトル形式の表式となっており、Numpyなどのライブラリで簡単に計算できます。

誤差δ^lの次層での誤差δ^{l+1}に関する表式: これは以下の通りです

$$\delta^l = ((w^{l+1})^T \delta^{l+1}) \odot \sigma'(z^l). \tag{BP2}$$

ここで、$(w^{l+1})^T$は$(l+1)$番目の層の重み行列w^{l+1}の転置です。この式は一見複雑ですが、各要素はきちんとした解釈を持ちます。$(l+1)$番目の層の誤差δ^{l+1}番目が既知だとします。重み行列の転置$(w^{l+1})^T$を掛ける操作は、直感的には誤差をネットワークとは**逆方向**に伝播させていると考える事ができます。従って、この値はl番目の層の出力の誤差を測る指標の一種とみなすことができます。転置行列を掛けた後、$\sigma'(z^l)$とのアダマール積を取っています。これによりl番目の層の活性関数を通してエラーを更に逆方向に伝播しています。その結果、l番目の層の重みつき入力についての誤差δ^lが得られます。

(BP2) を (BP1) と組み合わせる事で、ニューラルネットワークの任意の層lでの誤差δ^lを計算できます。まず、δ^Lを式 (BP1) で計算します。次に、式 (BP2) を適用してδ^{L-1}を計算します。その後、再び (BP2) を適用して、δ^{L-2}を計算します。以下これを繰り返してニューラルネットワークを逆向きに辿る事ができます。

任意のバイアスに関するコストの変化率の式: 具体的には以下の通りです

$$\frac{\partial C}{\partial b^l_j} = \delta^l_j. \tag{BP3}$$

すなわち、誤差δ^l_jはコスト関数の変化率$\partial C/\partial b^l_j$と**完全に同一**です。(BP1) と (BP2) からこの値の計算方法は既にわかっているので、この事実はは好都合です。(BP3) を簡潔に

$$\frac{\partial C}{\partial b} = \delta \tag{31}$$

と書くことができます。ここで、δの各成分は同じニューロンのバイアスbで評価した値と解釈します。

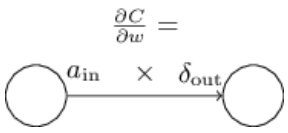
任意の重みについてのコストの変化率: 具体的には以下の通りです。

$$\frac{\partial C}{\partial w^l_{jk}} = a^{l-1}_k \delta^l_j. \tag{BP4}$$

この式を見ると、偏微分$\partial C/\partial w^l_{jk}$を計算方法が既知の$\delta^l$と$a^{l-1}$を用いて計算できることがわかります。この式はもう少し添字の軽い式で

$$\frac{\partial C}{\partial w} = a_{\rm in} \delta_{\rm out}, \tag{32}$$

と書き直せます。ここで、$a_{\rm in}$は重みwを持つ枝に対する入力ニューロンの活性で、$\delta_{\rm out}$は同じ枝に対する出力ニューロンの持つ誤差です。重みwとそれに接続する2つのニューロンだけに焦点を絞ると、この式は以下のように見ることができます:



式 (32) から$a_{\rm in}$が小さい($a_{\rm in} \approx 0$)時には、勾配$\partial C/\partial w$も小さくなる傾向があるという結論が得られます。このような状態を重みの**学習が遅い**と表現します。その意味は、勾配降下法を行っている間、値が大きく変化しないという事です。同じ事の言い換えですが、式 (BP4) の帰結の1つとして、活性の低いニューロンから入力を受けとる重みは学習が遅いとわかります。

(BP1) - (BP4) からわかる事は他にもあります。出力層から見てみましょう。(BP1) 内の$\sigma'(z^L_j)$の項に注目します。前章のシグモイド関数のグラフを思い出すと、$\sigma(z^L_j)$が0か1に近づくとき関数σはとても平坦になっていました。これは$\sigma'(z^L_j) \approx 0$の状態です。従って、出力ニューロンの活性が低かったり(≈ 0)、高かったり(≈ 1)すると、最終層の学習は遅い事がわかります。このような状況を、出力ニューロンは**飽和**し、重みの学習が終了している(もしくは重みの学習が遅い)と表現するのが一般的です。同様の事は出力ニューロンのバイアスに対しても成立します。

出力層より前の層でも似た考察ができます。特に (BP2) 内の$\sigma'(z^l)$の項に注目します。この式は、ニューロンが飽和状態だとδ^l_jは小さくなる傾向がある事を意味します。また、さらに飽和状態のニューロンに入力される重みの学習も遅くなることを意味します*。。

*$(w^{l+1})^T\delta^{l+1}$が十分大きく、$\sigma'(z^l_j)$が小さくてもその埋め合わせができるならば、この推論は成り立ちません。ここでは一般的な傾向について述べています。

まとめると、入力ニューロンが低活性状態であるか、出力ニューロンが飽和状態(低活性もしくは高活性状態)の時には、重みの学習が遅いという事がわかりました。

これらの知見はどれも極端に驚くべき事ではありません。しかし、これらの考察を通じてニューラルネットワークの学習過程に関するメンタルモデルの精緻にできます。さらに、以上の考察を逆向きに利用する事ができます。これら4つの基本方程式は任意の活性化関数について成立します (後述のように、証明に関数σの特別な性質を用いていないからです)。従って、これらの式を利用して好きな学習特性を持つ活性化関数を**設計**する事が可能です。アイデアを示すために例を挙げると、例えば(シグモイドではない)活性化関数σとしてσ'が常に正で、0に漸近しないものを選んだとします。すると、通常のシグモイド関数を用いたニューロンが飽和した際に起こってしまう学習の減速を防ぐ事が可能です。この本の後ろでは、この種の修正を活性化関数に対して施します。(BP1) - (BP4) の4つの式を覚えておくと、なぜこのような修正を行うのか、修正でどのような影響が起こるかを説明するのに役立ちます。

Summary: the equations of backpropagation

$$\delta^L = \nabla_a C \odot \sigma'(z^L) \tag{BP1}$$

$$\delta^l = ((w^{l+1})^T\delta^{l+1}) \odot \sigma'(z^l) \tag{BP2}$$

$$\frac{\partial C}{\partial b^l_j} = \delta^l_j \tag{BP3}$$

$$\frac{\partial C}{\partial w^l_{jk}} = a^{l-1}_k \delta^l_j \tag{BP4}$$

問題

- **誤差逆伝播の別の表示方法:** これまで、誤差逆伝播の式(特に (BP1) と (BP2))をアダマール積を用いて記述していました。アダマール積に慣れていない読者はこの表式にに戸惑ったかもしれません。これらの式を通常の行列の掛け算に基づいて表示する別の方法があります。読者によってはこのアプローチは教育的かもしれません。(1) (BP1) を以下の様に書き換えられる事を示してください

$$\delta^L = \Sigma'(z^L)\nabla_a C \tag{33}$$

ここで、$\Sigma'(z^L)$は$\sigma'(z^L_j)$を対角成分に持ち、非対角成分は0の正方行列です。この行列は$\nabla_a C$に通常の行列の掛け算で作用します。(2) (BP2) を以下の様に書き換えられる事を示してください

$$\delta^l = \Sigma'(z^l)(w^{l+1})^T \delta^{l+1}. \tag{34}$$

(3) (1)と(2)を組み合わせて、以下の式を示してください。

$$\delta^l = \Sigma'(z^l)(w^{l+1})^T \ldots \Sigma'(z^{L-1})(w^L)^T \Sigma'(z^L)\nabla_a C \tag{35}$$

行列の掛け算に慣れている読者にとっては (BP1) と (BP2) よりも、こちらの方が理解しやすいかもしれません。それでも (BP1) と (BP2) の表式を用いたのは、こちらの方が実装時の数値計算が速いからです。

4つの基本的な式の証明（任意）

それでは、(BP1)-(BP4)を証明していきます。これらはすべて多変数関数の微分の連鎖律の結論です。もし連鎖律に慣れていたら、読み進める前に自力での導出に挑戦してみるのを強くおすすめします。

まず、出力での誤差δ^Lの表式 (BP1) から証明しましょう。この式を示すのに、まず次の式を思い出します

$$\delta^L_j = \frac{\partial C}{\partial z^L_j}. \tag{36}$$

連鎖律を適用すると、この微分を出力活性に関する偏微分で書き直す事ができます

$$\delta^L_j = \sum_k \frac{\partial C}{\partial a^L_k}\frac{\partial a^L_k}{\partial z^L_j}. \tag{37}$$

ここで、和は出力層のすべてのニューロンkについて足し合わせます。もちろん、$k = j$の時には、k番目のニューロンの出力活性a^L_kは、j番目のニューロンの重み付き入力z^L_jにのみ依存します。従って、$k \neq j$の時には$\partial a^L_k / \partial z^L_j$の値は0です。結果として前述の式を以下のように簡略化できます

$$\delta^L_j = \frac{\partial C}{\partial a^L_j} \frac{\partial a^L_j}{\partial z^L_j}. \tag{38}$$

$a^L_j = \sigma(z^L_j)$であった事を思い出すと、第2項は$\sigma'(z^L_j)$と書けて、

$$\delta^L_j = \frac{\partial C}{\partial a^L_j} \sigma'(z^L_j) \tag{39}$$

となります。これを添字なしの形式で書くと (BP1) が得られます。

次に、誤差δ^lをその1つ後ろの層の誤差δ^{l+1}を用いて表す (BP2) を証明します。そのために、連鎖律を用いて$\delta^l_j = \partial C / \partial z^l_j$を$\delta^{l+1}_k = \partial C / \partial z^{l+1}_k$を用いて書き直します

$$\begin{aligned} \delta^l_j &= \frac{\partial C}{\partial z^l_j} && (40) \\ &= \sum_k \frac{\partial C}{\partial z^{l+1}_k} \frac{\partial z^{l+1}_k}{\partial z^l_j} && (41) \\ &= \sum_k \frac{\partial z^{l+1}_k}{\partial z^l_j} \delta^{l+1}_k. && (42) \end{aligned}$$

ここで、最後の行は2つの項を交換し、第2項をδ^{l+1}_kの定義で置き換えました。最後の行の第1項を評価するために、次の式に注意します

$$z^{l+1}_k = \sum_j w^{l+1}_{kj} a^l_j + b^{l+1}_k = \sum_j w^{l+1}_{kj} \sigma(z^l_j) + b^{l+1}_k. \tag{43}$$

この式を微分すると、次が得られます

$$\frac{\partial z^{l+1}_k}{\partial z^l_j} = w^{l+1}_{kj} \sigma'(z^l_j). \tag{44}$$

この式で (42) を置き換えると、次の式が得られます

$$\delta^l_j = \sum_k w^{l+1}_{kj} \delta^{l+1}_k \sigma'(z^l_j). \tag{45}$$

この式を添え字を用いずに書いたものが (BP2) そのものです。

証明したいあと2つの式は (BP3) と (BP4) です。これらの式もこれまでの2つの式と似た方法で連鎖律から導けます。証明は読者に

お任せします。

演習

- (BP3) と (BP4) を証明してください。

以上で逆伝播の4つの基本的な式の証明が完了しました。証明は一見複雑かもしれません。しかし、これらは連鎖律を慎重に適用した結果にしか過ぎません。もう少し詳しく言えば、逆伝播は多変数関数の微分で利用される連鎖律をシステマチックに適用する事で、コスト関数の勾配を計算する方法と見る事ができます。それが逆伝播の正体であり、残りは些細な部分です。

逆伝播アルゴリズム

逆伝播の式により、コスト関数の勾配の計算が可能になりました。その方法を具体的にアルゴリズムの形で書き下してみましょう。

1. 入力 x: 入力層に対応する活性a^1をセットする
2. フィードフォワード: 各$l = 2, 3, \ldots, L$に対し、$z^l = w^l a^{l-1} + b^l$ and $a^l = \sigma(z^l)$を計算する
3. 誤差δ^Lを出力: 誤差ベクトル$\delta^L = \nabla_a C \odot \sigma'(z^L)$を計算する
4. 誤差を逆伝播: 各$l = L-1, L-2, \ldots, 2$に対し、$\delta^l = ((w^{l+1})^T \delta^{l+1}) \odot \sigma'(z^l)$を計算する
5. 出力: コスト関数の勾配は $\frac{\partial C}{\partial w^l_{jk}} = a^{l-1}_k \delta^l_j$と $\frac{\partial C}{\partial b^l_j} = \delta^l_j$ で得られる

アルゴリズムを見ると、これがなぜ**逆**伝播と呼ばれるかがわかるでしょう。最終層から始まり、逆向きに誤差ベクトルδ^lを計算しています。ネットワークを逆向きに辿るのが奇妙に思われるかもしれません。しかし、逆伝播の証明を思い返すと、コストはニューラルネットワークの出力についての関数であるという事から、逆伝播の方向が決まっている事がわかります。前段の重みやバイアスによりコストがどのように変化するかを見るためには、連鎖律を繰り返し適用しなければならず、欲しい計算式を得るにはネットワークを逆方向に辿る必要があります。

演習

- **ニューロンの1つを差し替えた時の逆伝播:** フィードフォワードニューラルネットワークの特定の1つのニューロンを、$f(\sum_j w_j x_j + b)$を出力するものに変更したとします。ここで、fはシグモイド以外の適当な関数です。この場合、逆伝播アルゴリズムはどのように変更すればよいでしょうか。

- **線形ニューロンでの逆伝播:** ニューラルネットワーク内の非線形のσ関数を$\sigma(z) = z$に変更したとします。逆伝播アルゴリズムをこの変更にあうように書き直してください。

前述のように、逆伝播アルゴリズムは単一の訓練例に対するコスト関数$C = C_x$の勾配を計算します。しかし実際の実装では、逆伝播を、確率的勾配降下法などの多数の訓練例に対する勾配を計算する学習アルゴリズムと組み合わせるのが一般的です。以下のアルゴリズムでは、m個の訓練例からなるミニバッチに対して勾配降下法を適用して学習を行っています。

1. **訓練例のセットを入力**

2. **各訓練例xに対して:** 対応する活性$a^{x,1}$をセットし、以下のステップを行う:

   - **フィードフォワード:** $l = 2, 3, \ldots, L$に対し、$z^{x,l} = w^l a^{x,l-1} + b^l$と$a^{x,l} = \sigma(z^{x,l})$を計算する

   - **誤差$\delta^{x,L}$を出力:** ベクトル$\delta^{x,L} = \nabla_a C_x \odot \sigma'(z^{x,L})$を計算する

   - **誤差を逆伝播する:** $l = L-1, L-2, \ldots, 2$に対し、$\delta^{x,l} = ((w^{l+1})^T \delta^{x,l+1}) \odot \sigma'(z^{x,l})$を計算する

3. **勾配降下:** $l = L, L-1, \ldots, 2$に対し、重みを$w^l \rightarrow w^l - \frac{\eta}{m} \sum_x \delta^{x,l} (a^{x,l-1})^T$で更新し、バイアスを$b^l \rightarrow b^l - \frac{\eta}{m} \sum_x \delta^{x,l}$で更新する

もちろん、確率的勾配降下法を実装する際は、訓練例のミニバッチを作成する外部ループと複数回のエポックで訓練を繰り返す為の外部ループが必要です。簡単のためにそれらは記述しませんでした。

逆伝播の実装

逆伝播の理論を理解した事で、逆伝播の実装に利用した前章のコードを理解できる段階に達しました。この章を思い出すと、逆伝播の実装は`Network`クラスの`update_minibatch`メソッドと`backprop`メソッドに含

まれていました。これらのメソッドは前述のアルゴリズムをそのままコードに翻訳したものです。update_mini_batchメソッドは現在の訓練例のmini_batchについて勾配を計算し、Networkクラスの重みとバイアスを更新しています。

```
class Network():
...
    def update_mini_batch(self, mini_batch, eta):
        """ミニバッチ1つ分に逆伝播を用いた勾配降下法を適用し、
        ニューラルネットワークの重みとバイアスを更新する。
        "mini_batch"はタプル"(x, y)"のリストで"、
        eta"は学習率。"""
        nabla_b = [np.zeros(b.shape) for b in self.biases]
        nabla_w = [np.zeros(w.shape) for w in self.weights]
        for x, y in mini_batch:
            delta_nabla_b, delta_nabla_w = self.backprop(x, y)
            nabla_b = [nb+dnb for nb, dnb in zip(nabla_b, delta_nabla_b)]
            nabla_w = [nw+dnw for nw, dnw in zip(nabla_w, delta_nabla_w)]
        self.weights = [w-(eta/len(mini_batch))*nw
                        for w, nw in zip(self.weights, nabla_w)]
        self.biases = [b-(eta/len(mini_batch))*nb
                       for b, nb in zip(self.biases, nabla_b)]
```

ほとんどの作業はdelta_nabla_b, delta_nabla_w = self.backprop(x, y)の行で行われています。この行では、backpropメソッドを利用して偏微分$\partial C_x/\partial b^l_j$と$\partial C_x/\partial w^l_{jk}$を計算しています。backpropメソッドは前節のアルゴリズムに従って実装されています。層への添字の振り方を、前節の説明から若干の変更しています。この変更はPythonの特徴、具体的には負数の添字を用いてリストを後ろから数える方法を活用するために行っています(例えば、l[-3]はリストlの後ろから3番目の要素です)。次にbackpropメソッドのコードを示します。σ関数とそのベクトル化、σ関数の導関数とそのベクトル化、及びコスト関数の微分を計算するためのヘルパー関数も併せて載せています。これらのヘルパー関数を併せて見れば、自己完結した形でコードを理解できるはずです。もしどこかでつまづいたら 元のコードの説明(と全コード)を参照するのが良いでしょう。

```
class Network():
...
    def backprop(self, x, y):
        """コスト関数の勾配を表すタプル"(nabla_b, nabla_w)"を返却する。
        "self.biases" and "self.weights"と同様に、
        "nabla_b"と"nabla_w"はnumpyのアレイのリストで
        各要素は各層に対応する。"""
        nabla_b = [np.zeros(b.shape) for b in self.biases]
        nabla_w = [np.zeros(w.shape) for w in self.weights]
        # 順伝播
        activation = x
        activations = [x] # 層ごとに活性を格納するリスト
        zs = [] # 層ごとにzベクトルを格納するリスト
        for b, w in zip(self.biases, self.weights):
            z = np.dot(w, activation)+b
            zs.append(z)
```

```
            activation = sigmoid_vec(z)
            activations.append(activation)
        # 逆伝播
        delta = self.cost_derivative(activations[-1], y) * ¥
            sigmoid_prime_vec(zs[-1])
        nabla_b[-1] = delta
        nabla_w[-1] = np.dot(delta, activations[-2].transpose())
        # 下記のループ変数lは第2章での記法と使用方法が若干異なる。
        # l = 1は最終層を、l = 2は最後から2番目の層を意味する（以下同様）。
        # 本書内での方法から番号付けのルールを変更したのは、
        # Pythonのリストでの負の添字を有効活用するためである。
        for l in xrange(2, self.num_layers):
            z = zs[-l]
            spv = sigmoid_prime_vec(z)
            delta = np.dot(self.weights[-l+1].transpose(), delta) * spv
            nabla_b[-l] = delta
            nabla_w[-l] = np.dot(delta, activations[-l-1].transpose())
        return (nabla_b, nabla_w)
...
    def cost_derivative(self, output_activations, y):
        """出力活性に対する偏微分¥partial C_x / ¥partial a
        のベクトルを返却する。"""
        return (output_activations-y)

def sigmoid(z):
    """シグモイド関数"""
    return 1.0/(1.0+np.exp(-z))

sigmoid_vec = np.vectorize(sigmoid)

def sigmoid_prime(z):
    """シグモイド関数の導関数"""
    return sigmoid(z)*(1-sigmoid(z))

sigmoid_prime_vec = np.vectorize(sigmoid_prime)
```

問題

- ミニバッチによる逆伝播の行列を用いた導出: 我々の確率的勾配降下法の実装ではミニバッチ内の訓練例についてループしています。しかし、逆伝播のアルゴリズムを書き換えると、ミニバッチ内の全訓練例の勾配を同時に計算するように変更できます。単一の入力ベクトルxから始めるのではなく、各列がミニバッチ内のベクトルからなる行列$X = [x_1 x_2 \ldots x_m]$を用いるのが基本的なアイデアです。この行列に重み行列を掛け、バイアス項に対応する適当な行列を足し、各要素にシグモイド関数を適用する事で順伝播をします。逆伝播も似た方法で行います このアプローチによる逆伝播アルゴリズムの擬似コードを具体的に書き下してください。また、network.pyを行列を用いたアプローチに変更してください。このアプローチの利点は、最近の線形代数ライブラリをフルに有効活用でき、その結果ミニバッチ内をループする場合に比べて圧倒的に高速になる点です（例えば私のノートパソコンでは、前章で考えたMNISTの分類問

題で約2倍の高速化の効果が得られました)。実際きちんと作られた逆伝播のライブラリでは、この行列のアプローチかその変種を用いています。

逆伝播が速いアルゴリズムであるとはどういう意味か？

どういう意味で逆伝播は速いアルゴリズムか。これに答える為に、勾配を計算する別のアプローチを考えてみましょう。初期の時代のニューラルネットワーク研究を想像してみてください。おそらく1950年代か60年代だと思いますが、あなたは学習への勾配降下法の適用を考えている世界で最初の研究者です！ あなたの考えがうまくいくかを確かめるには、コスト関数の勾配を計算する方法が必要です。微積分学の知識を思い出して、勾配の計算に連鎖律が使うかを検討しています。しかし、少しごにょごにょと計算してみると、式は複雑そうなのでがっかりしてしまいます。そこで、別のアプローチを探します。コスト関数を重みのみの関数とみなし、$C = C(w)$と考えることにしました(バイアスについてはすぐ後で考えます)。重みを$w_1, w_2, \ldots$と番号付けし、特定の重みw_jについて$\partial C/\partial w_j$を計算します。すぐに思いつくのは近似

$$\frac{\partial C}{\partial w_j} \approx \frac{C(w+\epsilon e_j) - C(w)}{\epsilon} \tag{46}$$

を利用する方法です。ここで、$\epsilon > 0$は微小な正の数で、e_jはj方向の単位ベクトルです。言い換えれば、$\partial C/\partial w_j$を計算する為に2つの若干異なるw_jでコストCの値を計算し、式 (46) を適用します。同じアイデアでバイアスについての偏微分$\partial C/\partial b$にも計算できます。

このアプローチはよさそうに見えます。発想がシンプルな上、実装も数行のコードで実現できとても簡単です。連鎖律を用いて勾配を計算するアイデアよりもよっぽど有望なように思えます！

このアプローチは有望そうですが、残念ながらこのコードを実装してみるととてつもなく遅い事がわかります。なぜかを理解する為に、ニューラルネットワーク内に100万個の重みがあると想像してみてください。すると、各重みw_jに対して$\partial C/\partial w_j$を計算するには、$C(w+\epsilon e_j)$の計算が必要です。これには、勾配計算時に異なる値でのコスト関数計算が100万回必要で、各訓練例ごとに100万回の順伝播が必要な事を意味します。$C(w)$の計算も必要なので、結局ニューラルネットワーク内伝播回数は100万1回です。

逆伝播の賢い所は、たった1回の順伝播とそれに続く1回の逆伝播で**すべての**偏微分$\partial C/\partial w_j$を同時に計算できる点です。逆伝播の計算コストは大雑把には順伝播と同程度です *。従って、逆伝播の合計のコストはニューラルネットワーク全体への順伝播約2回分です。(46)に基づくアプローチで必要だった100万1回の順伝播と比較してみてください！ 逆伝播は一見 (46) に基づく方法よりも複雑ですが、実際にはずっと、ずっと高速なのです。

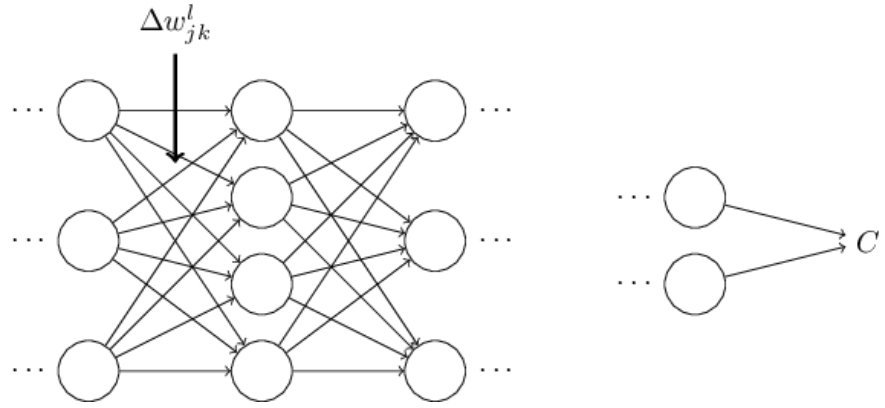
この見積りは妥当ですが、きちんと示すには若干の分析が必要です。フォワードパスの計算コストで支配的なのは重み行列の掛け算であるのに対し、バックワードパスで支配的なのは重み行列の転置の掛け算です。これらの操作は明らかに同程度の計算コストです。

この高速化は1986年に始めて真価がわかり、ニューラルネットワークが解ける問題の幅を大きく広げ、その結果多くの人が次々にニューラルネットワークに押しかけました。もちろん、逆伝播は万能薬ではありません。特にディープニューラルネットワーク、すなわち多くの隠れ層を持つネットワークの学習への逆伝播の適用においては、1980年代後半には既に壁にぶつかっていました。現代のコンピュータや新しい賢いアイデアにより、逆伝播を用いてディープニューラルネットワークを訓練する事が可能になった事をこの本では後述します。

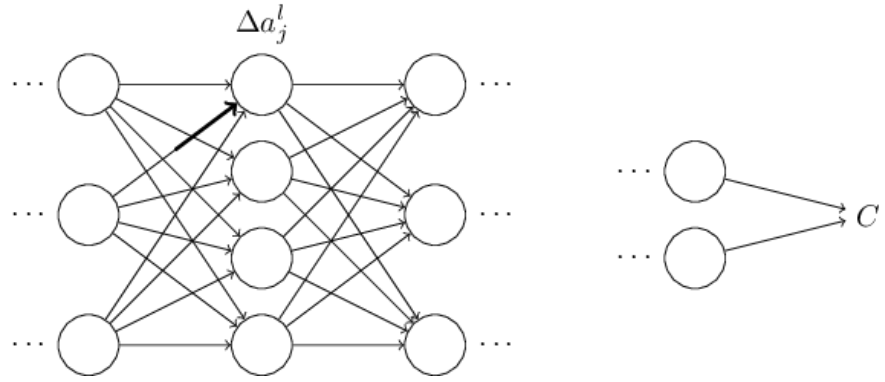
逆伝播：全体像

以前に説明したように、逆伝播には2つの謎があります。1つはアルゴリズムが本当にやっている事は何かです。出力から誤差が逆伝播していく様子を見てきました。もう一歩踏み込んで、ベクトルに行列を掛ける時に何が起こっているかについてのもっと直感的な理解を得られないでしょうか。2つ目の謎は、そもそもどうやって逆伝播を発見するかという点です。アルゴリズムの手順に従ったり、アルゴリズムの正しさを示すを証明を追う事はできます。しかし、その事と、問題を理解しアルゴリズムをまっさらな状態から発見するのはまた別の話です。逆伝播アルゴリズムの発見につながる妥当な論理づけは何かないでしょうか。本節ではこれらの謎に重点を置きます。

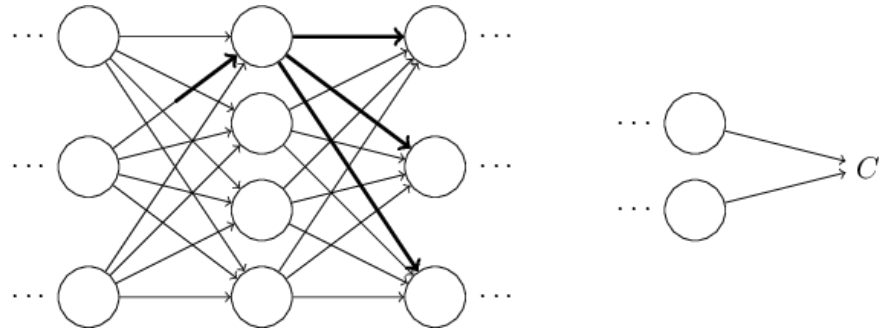
アルゴリズムの挙動に対する直感を養う為に、ニューラルネットワーク内の適当な重みw^l_{jk}に微小な変化Δw^l_{jk}を施してみましょう:



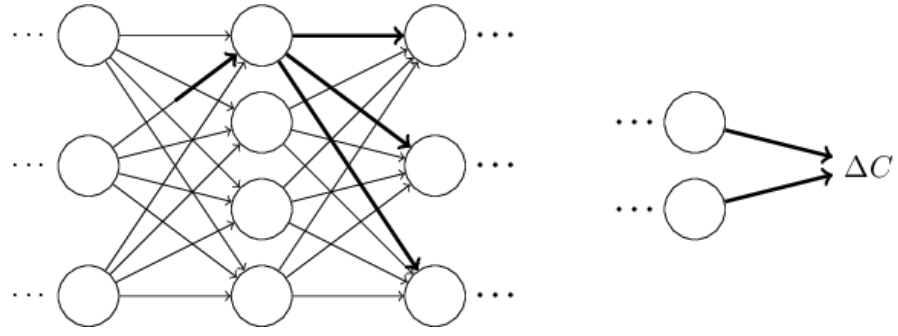
重みの変化により、対応するニューロンの出力活性が変化します:



この変化は引き続いて、次の層の**すべての**出力活性に変化を引き起こします:



これらの変化はさらに次の層の変化を引き起こします。これを繰り返して最終層、そしてコスト関数を変化させます:



コスト関数の変化ΔCは重みの変化Δw^l_{jk}と次式で関連付けられます

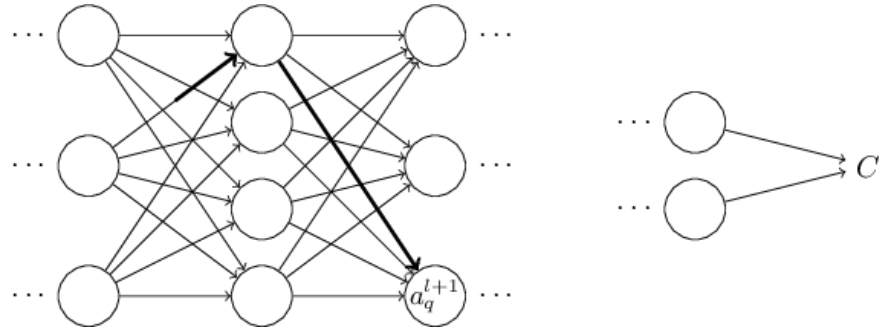
$$\Delta C \approx \frac{\partial C}{\partial w^l_{jk}} \Delta w^l_{jk}. \tag{47}$$

この式から、$\frac{\partial C}{\partial w^l_{jk}}$を計算するのに考えられるアプローチとして次の方法が示唆されます。すなわち、w^l_{jk}の微小な変化がニューラルネットワークを伝播し、その結果Cの微小な変化を引き起こす様子を丁寧に追跡するという方法です。もしそれができたら、伝播経路の途中にあるすべてのものを、簡単に計算できる変数で表現する事で、$\partial C/\partial w^l_{jk}$を計算できるはずです。

このアイデアを実行に移してみましょう。重みがΔw^l_{jk}だけ変化する事でl番目の層のj番目のニューロンの活性に微小な変化Δa^l_jが発生します。この変化は

$$\Delta a^l_j \approx \frac{\partial a^l_j}{\partial w^l_{jk}} \Delta w^l_{jk} \text{で与えられます。} \tag{48}$$

活性の変化Δa^l_jは次の層、すなわち$l+1$番目の層の**すべての**活性に変化を引き起こします。私達はこれらの活性の中の1つ、例えばa^{l+1}_qがどのような影響を受けるかのみに注目します。



Δa^l_jは次のような変化を引き起こします

$$\Delta a^{l+1}_q \approx \frac{\partial a^{l+1}_q}{\partial a^l_j} \Delta a^l_j. \tag{49}$$

式 (48) 内の表式をこれで置き換えると、

$$\Delta a^{l+1}_q \approx \frac{\partial a^{l+1}_q}{\partial a^l_j} \frac{\partial a^l_j}{\partial w^l_{jk}} \Delta w^l_{jk} \tag{50}$$

が得られます。もちろん今度はΔa^{l+1}_qが、次の層の活性に変化を引き起こします。実際には、w^l_{jk}からCまでのパスのうちの1つを考えると、このパスでは活性のそれぞれの変化が次の活性の変化を引き起こし、最終的に出力でのコストの変化を引き起こしています。もしこのパスが$a^l_j, a^{l+1}_q, \ldots, a^{L-1}_n, a^L_m$を通るとしたら、得られる表式は

$$\Delta C \approx \frac{\partial C}{\partial a^L_m} \frac{\partial a^L_m}{\partial a^{L-1}_n} \frac{\partial a^{L-1}_n}{\partial a^{L-2}_p} \ldots \frac{\partial a^{l+1}_q}{\partial a^l_j} \frac{\partial a^l_j}{\partial w^l_{jk}} \Delta w^l_{jk} \tag{51}$$

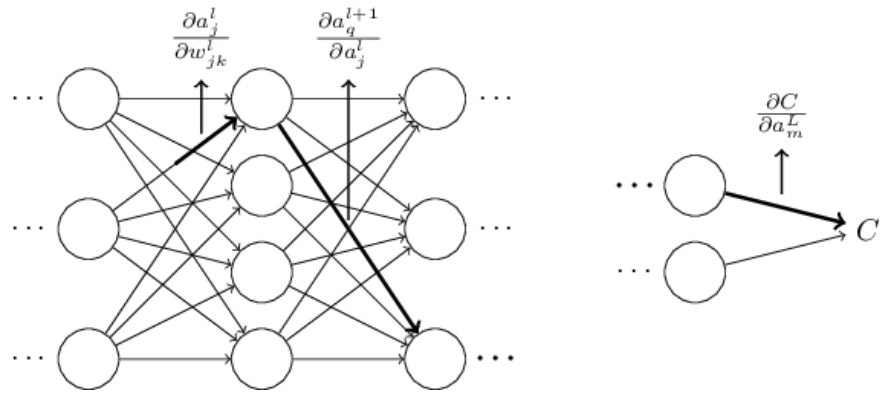
となります。すなわち、ニューロンを通過するごとに$\partial a/\partial a$の形の項が追加され、最後に$\partial C/\partial a^L_m$の項が付け加わります。この値は$C$の変化のうち、特定のパス内にある活性の変化に由来するものです。w^l_{jk}の変化を伝播しコストに影響を与えるパスは他にもたくさんあり、この式はその中の1つしか考慮していません。Cの変化の合計を計算するには、最初の重みと最後のコストの間で取りうる全てのパスについて和を取れば良いです。すなわち、

$$\Delta C \approx \sum_{mnp\ldots q} \frac{\partial C}{\partial a^L_m} \frac{\partial a^L_m}{\partial a^{L-1}_n} \frac{\partial a^{L-1}_n}{\partial a^{L-2}_p} \ldots \frac{\partial a^{l+1}_q}{\partial a^l_j} \frac{\partial a^l_j}{\partial w^l_{jk}} \Delta w^l_{jk}, \quad (52)$$

ここで、和はパスを通る中間ニューロンの選び方として考えられる全体について足し合わせます。(47) と比較すると、

$$\frac{\partial C}{\partial w^l_{jk}} = \sum_{mnp\ldots q} \frac{\partial C}{\partial a^L_m} \frac{\partial a^L_m}{\partial a^{L-1}_n} \frac{\partial a^{L-1}_n}{\partial a^{L-2}_p} \ldots \frac{\partial a^{l+1}_q}{\partial a^l_j} \frac{\partial a^l_j}{\partial w^l_{jk}} \quad (53)$$

とわかります。式(53) は一見すると複雑そうに見えます。しかし、これには直感的な良い解釈があります。私達は今、ニューラルネットワーク内の重みに関するCの変化率を計算しています。ニューラルネットワーク内の2つのニューロンを繋ぐ全ての枝に対して、変化率の因子が付随している事がこの式から分かります。その因子は一方のニューロンの活性に関する、もう一端のニューロンの活性の偏微分です。ただし、先頭の重みから第1層目のニューロンに接続している枝には始点にニューロンが接続していないですが、この枝に対する変化率の因子は$\partial a^l_j/\partial w^l_{jk}$です。パスに対する変化率の因子は、単純にパス内に含まれる変化率の因子を全て掛けたものとします。そして、$\partial C/\partial w^l_{jk}$に対する変化率の合計は最初の重みから最後のコストへ向かう全てのパスについての変化率の因子を足しあわせたものです。下図では1つのパスについてこの手順を図示しています。



これまでの議論は、ニューラルネットワーク内の重みを摂動させた時に何が起こっているかを発見的に考察する方法でした。この方向で

議論をさらに進める方法を簡単に紹介します。まず、式 (53) 内の偏微分はすべて具体的な表式を与えます。これは若干の計算をするだけで難しくはありません。これを行うと、添字について和を取る操作を行列操作に書き直す事ができるようになります。退屈で忍耐が必要な作業かも知れませんが、賢い洞察は必要ありません。その後できるだけ式を簡単にしていくと、なんと最終的に得られる式は逆伝播アルゴリズムそのものです！ つまり、逆伝播アルゴリズムは全パスの変化率の因子を総和を計算する方法とみなすことができるのです。少し別の表現をすると、逆伝播アルゴリズムは重み(とバイアス)に与えた小さな摂動がニューラルネットワークを伝播しながら出力に到達し、コストに影響を及ぼす様子を追跡する為の賢い方法だと言えます。

ここでは上の議論には立ち入りません。議論の詳細を全て追うのは非常にややこしく、相当の注意が必要です。もし挑戦する意欲があれば、試してみるとよいでしょう。もしそうでなくても、以上の議論で誤差逆伝播が達成しようしている事について何かの洞察が得られる事を期待します。

もう1つの謎、すなわち、まっさらの状態から誤差逆伝播を発見する方法についてはどうでしょうか。確かに今私が概説したアプローチに従えば、誤差逆伝播の証明は発見できます。しかし、残念ながらその証明はこの章の前の方で挙げた証明よりも若干長くて複雑です。では、どのようにすればこのもっと短い(けれど不思議な)証明を発見できるでしょうか。長い証明の詳細をすべて書きだしてみると、幾つかの明らかな簡略化が目につくはずです。それらの簡略化を行うと証明を短くできます、それをまた書き出してみます。すると再び明らかな簡略化が飛び出しますので、同じようにその簡略化を行います。これを数回繰り返すと、本章の前の方で挙げた、短いけれども、若干わかりにくい証明が得られます*。証明がわかりにくいのは、それを構成する際に道標となるようなものが除かれてしまった為です。私を信用してもらう必要があるのですが、本章で挙げた短い証明の起源には全くもって謎はないのです。本章で挙げた(短い)証明は、この章で紹介した(長い)証明を頑張って簡略化して得られたものです。

*1箇所賢い操作が必要な箇所があります。式 (53) において、中間変数はa_q^{l+1}のような活性です。賢いアイデアというのは、中間変数をz_q^{l+1}のような重みつき入力に取り替えるというものです。このアイデアを採用せず、活性a_q^{l+1}を使い続けると、最終的に得られる証明は若干複雑になります。

第3章

ニューラルネットワークの学習の改善

最初にゴルフを始めようとするとき、まずは基本スイングの練習にほとんどの時間を使うのが普通です。それ以外のスイングの練習をするのは、少しずつしかできません。チップやドローやフェードを身につけるのは、基本スイングの上に、修正しながら組み立てるものです。同様に、私達はいままで逆伝搬法に集中してきました。それが私達にとっての「基本スイング」であり、ニューラルネットワークにおけるほとんどの仕事を理解するための基本だったからです。この章では、純粋な逆伝搬の実装を改善しネットワークの学習のしかたを改善するいくつかのテクニックを説明します。

この章で説明するテクニックは以下のとおりです。クロスエントロピーと呼ばれるより良いコスト関数、ネットワークを学習データによらず汎化するのに役立つ「正規化」法と呼ばれる4つの手法（L1正規化およびL2正規化、ドロップアウト、そして人工的な学習データの伸張）、ネットワークの中の重みを初期化するより良い方法、そして、ネットワークに対して良いハイパーパラメータを選択するためのいくつかの発見的方法です。また、それ以外のいくつかのテクニックについても、あまり細部にこだわらず概観します。これらの議論は完全にお互いに独立なので、必要に応じて読み飛ばすこともできます。また、多くのテクニックを動くコードとして実装もします。それらの実装を使うと、第1章で説明した手書き文字の分類問題の結果が改善されます。

もちろん、ここではニューラルネットワークのために開発されたとても多くの手法のうちほんの少しを紹介するだけです。ここでの哲学は、利用できる手法が沢山ありすぎる場合の最も良い入門は、少しの手法を深く学習することだということです。それらの重要な手法はそのまま役に立つというだけではなく、ニューラルネットワークを使うときに起こる問題に対する理解を深めてくれます。その結果、必要に応じて他のテクニックを即座に使えるようになるでしょう。

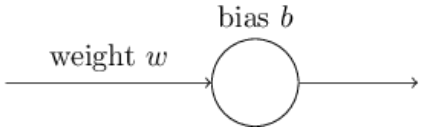
クロスエントロピーコスト関数

ほとんどの人にとって、間違えることは嫌なことです。私はピアノを習い始めてすぐに聴衆の前で演奏することがありました。そのとき緊張していたので、1オクターブ低く演奏を始めてしまいました。私は混乱してしまい誰かが間違いを指摘するまで気づきませんでした。とても恥ずかしい思いをしました。私達は、はっきり間違えているときには、嫌な思いをしつつも速

著:Michael Nielsen / 2014年9月-12月
訳:「ニューラルネットワークと深層学習」翻訳プロジェクト

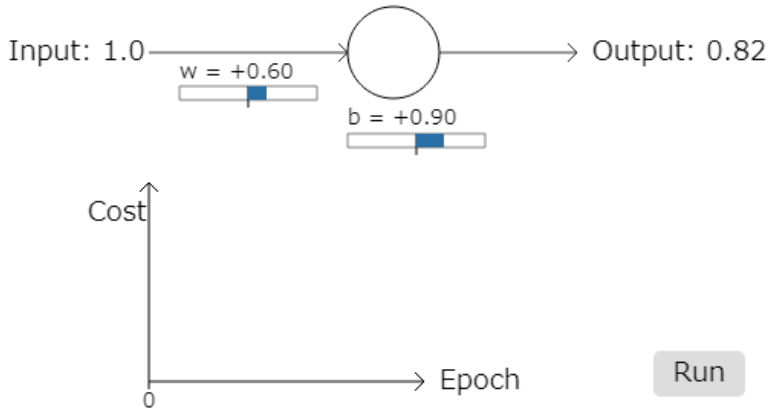
く学ぶことができます。次に私がピアノを聴衆の前で弾くときには正しいオクターブで弾いたということは、いうまでもないでしょう。一方で、間違いがはっきりしないときはゆっくりとしか学べません。

理想的にはニューラルネットワークは間違いから学んでほしいと、我々は願っていて期待もしています。そのようなことは実際に起こるでしょうか。この問いに答えるために簡単な例を見てみましょう。この例はちょうど一個の入力を持つニューロンです。

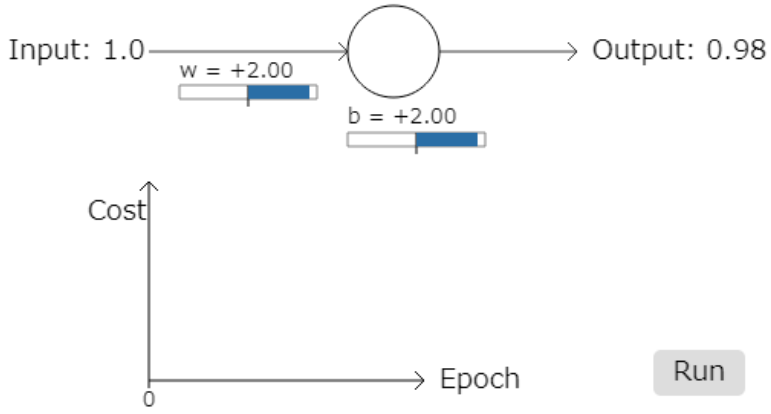


バカバカしいほど簡単なことをするためにこのニューロンを学習させます。入力1に対して出力0を返すようにします。もちろんこれは適当な重みとバイアスをすぐに手計算でき、学習アルゴリズムを使わないですむような自明な作業です。しかし、最急降下法を使って重みとバイアスを計算することは理解に役立ちます。ですから、ニューロンがどのように学ぶかを見てみましょう。

話をはっきりするために、初期の重みを0.6、バイアスを0.9としましょう。この設定は学習の前の設定として適当に選んだもので、とにかく特別な値を採用したわけではありません。ニューロンからの出力の初期値は0.82なので、ニューロンが望まれる出力の0.0を出すようになるようになるには、それなりの時間がかかりそうです。出力が0.0にとても近くなるまでにどのようにニューロンが学習していくかを見るには、右下にある「Run」ボタンを押してみてください。これは録画された動画ではないことに注意してください。ブラウザは実際に勾配を計算して、その勾配は重みとバイアスを更新するのに使われて、結果が表示されています。学習係数は $\eta = 0.15$ であり、これの値は、なにが起こっているかを見るには十分に遅く、しかし数秒でそれなりの学習をさせるには十分に速い値です。コスト関数は、第1章で紹介した二次コスト関数です。コスト関数の厳密な式は後で復習するので、今とくに定義を深く追う必要はありません。このアニメーションは「Run」を押せば何度でも実行できます。



これでわかるように、ニューロンは急速に重みとバイアスを学習していき、コストを下げていきます。結果としてニューロンからの出力は0.09になります。これは望まれる出力0.0とは違いますが、良い値です。しかしながら、最初の重みとバイアスとして2.0という値を選んだとしてみましょう。この場合にニューロンがどのように出力が0になるように学習していくかを見てみましょう。今度も「Run」を押してください。



この例は同じ学習係数($\eta = 0.15$)を使っているのにもかかわらず、学習がずっとゆっくり始まることがわかります。実施、最初の150エポックくらいでは重みやバイアスはほとんど変わりません。その後に学習が始まり、最初の例と同じように出力は急速に0.0に近づきます。

この振る舞いは人間の学習と比べると奇異に見えます。この節の最初に言ったように、我々はひどく間違った時に速く学習することが多いです。ですが、我々の人工的なニューロンは、ひどく間違っている時のほうが、少しだけ間違えていた時と比べて学習がとても困難であったように見えます。さらにいうと、このような振る舞いはこの単純なモデルだけに起こることではなく、もっと一般的なネットワークでも起こることがわかっています。なぜ学習がそんなに遅いのでしょう？そして、このように遅くなることを防ぐ手段はあるのでしょうか？

問題の原点を理解するために我々のニューロンが、重みとバイアスをコスト関数の偏微分$\partial C/\partial w$と$\partial C/\partial b$によって決まる値で更新されるとしましょう。「学習が遅い」ということは、その偏微分が小さいということと同じことです。問題は、なぜこれが小さいのかということを理解することです。そのことを理解するために偏微分を計算してみましょう。我々は二次導関数を使っていたことを思い出しましょう。これは、式(6)から、次の式で与えられます。

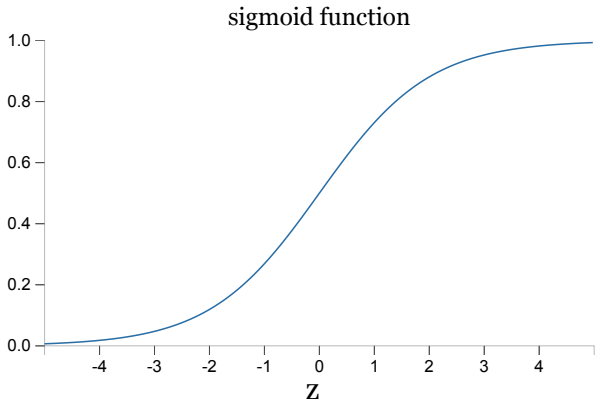
$$C = \frac{(y-a)^2}{2}, \tag{54}$$

ここで、aは学習の入力$x = 1$に対するニューロンの出力であり、$y = 0$は対応する期待される出力です。このことを重みとバイアスの言葉でもっとはっきり記述するために、$z = wx + b$のとき$a = \sigma(z)$と表せることを思い出しましょう。合成関数微分の公式を使い、重みとバイアスで微分すると以下の式を得ます。

$$\frac{\partial C}{\partial w} = (a-y)\sigma'(z)x = a\sigma'(z) \tag{55}$$

$$\frac{\partial C}{\partial b} = (a-y)\sigma'(z) = a\sigma'(z), \tag{56}$$

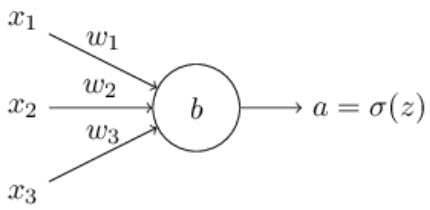
ここで、$x = 1$と$y = 0$の代入を使いました。これらの表現の振る舞いを理解するためには、右辺に出てくる$\sigma'(z)$の項に注目しましょう。関数σのグラフの形状をみてみましょう。



ニューロンが1に近づくと曲線がとても平らになり、そのため$\sigma'(z)$がとても小さくなることがわかります。そして、式(55)と(56)により$\partial C/\partial w$と$\partial C/\partial b$がとても小さくなります。これが学習が遅くなる原因です。さらにいうと、いままで扱ってきた単純な例に限らずとも、もっと一般的なニューラルネットワークでも学習が遅くなるのは根本的には同じ理由によるものです。

クロスエントロピーコスト関数の導入

学習が遅くなる問題はどう扱えばよいでしょう。二次コスト関数のかわりに、クロスエントロピーと呼ばれる他のコスト関数を使うことで解くことができることがわかります。クロスエントロピーを理解するには、我々の非常に簡単な例からちょっとだけ離れる必要があります。その代わり、ニューロンをいつくかの入力変数$x_1, x_2, \ldots$で学習させることとし、それらのそれぞれに対応する重みを$w_1, w_2, \ldots$をとし、バイアスをbとします。



ニューロンの出力はもちろん$a = \sigma(z)$となり、ここで$z = \sum_j w_j x_j + b$は重み付きの入力の和です。このニューロンのクロスエントロピーを次の式で定義します。

$$C = -\frac{1}{n} \sum_x \left[y \ln a + (1-y) \ln(1-a) \right], \tag{57}$$

ここでnは学習データのデータ数で、和の計算はすべての学習の入力xについてで、yは対応する望まれる出力です。

この式(57)が、学習が遅くなる問題を解決するということは明らかではありません。それどころか、これをコスト関数と呼んでいいかすら明らかではないでしょう。学習が遅くなる問題に注目する前に、どういう意味でクロスエントロピーがコスト関数として解釈できるかを見てみましょう。

二つの性質によりクロスエントロピーはコスト関数であると解釈できます。一つにはそれは非負、つまり$C > 0$であることです。このことを知るには、式(57)のすべての項が正であることを確認すればいいです。これは、(a)両方のlogの引数が0から1の範囲なのでその値は負になり、(b)前にマイナス符号が付いているからです。

二つ目には、すべての学習の入力xについてニューロンの出力が望まれる出力、つまり$y = y(x)$に近いのなら、クロスエントロピーはゼロに近づくということです。このことを見るために、例えば同じ入力xに対し$y(x) = 0$かつ$a \approx 0$と仮定しましょう。これはニューロンがその入力に対して良い仕事をした場合です。$y(x) = 0$なので、式(57)の最初の項が消え、2項目は$-\ln(1-a) \approx 0$となります。$y(x) = 1$かつ$a \approx 1$の場合も同様な解析ができます。このことにより、実際の出力が望まれる出力に近いとき、コストへの寄与は小さいことがわかります。

まとめると、クロスエントロピーは正で、すべての学習の入力xと望まれる出力yに関して、ニューロンがより良くなるとクロスエントロピーは0に近づ

*このことそ証明するには、望まれる出力$y(x)$が0か1であることを仮定しなければなりません。これは、例えば分類問題や真偽値関数を計算しているときについては通常成り立ちます。この仮定をおかない時になにが起こるかを理解するには、この節の最後の演習を参照してください。

きます。この両方の性質は、私達がコスト関数に対して直感的に期待する性質です。実際、これらの性質は二次コスト関数でも満たされています。これはクロスエントロピーにとって良いニュースです。しかし、クロスエントロピーは二次コスト関数と違って、学習が遅くなる問題がないという良い性質があります。このことを見るために、クロスエントロピーの重みについての偏微分を計算してみましょう。式(57)に$a = \sigma(z)$を代入して、合成関数の公式を**2**度適用すると以下の式を得ます:

$$\frac{\partial C}{\partial w_j} = -\frac{1}{n} \sum_x \left(\frac{y}{\sigma(z)} - \frac{(1-y)}{1-\sigma(z)} \right) \frac{\partial \sigma}{\partial w_j} \tag{58}$$

$$= -\frac{1}{n} \sum_x \left(\frac{y}{\sigma(z)} - \frac{(1-y)}{1-\sigma(z)} \right) \sigma'(z) x_j. \tag{59}$$

すべてを共通の分母でまとめて簡単にすると次の式を得ます:

$$\frac{\partial C}{\partial w_j} = \frac{1}{n} \sum_x \frac{\sigma'(z) x_j}{\sigma(z)(1-\sigma(z))} (\sigma(z) - y). \tag{60}$$

シグモイド関数の定義を使うと$\sigma(z) = 1/(1+e^{-z})$であり、すこしの代数的計算をすると$\sigma'(z) = \sigma(z)(1-\sigma(z))$であることがわかります。このことは演習で確認しますが、とりあえずはそれはわかっているものとして受け入れましょう。ちょうど上の式で、$\sigma'(z) = \sigma(z)(1-\sigma(z))$が消え、単純になり次のようになります:

$$\frac{\partial C}{\partial w_j} = \frac{1}{n} \sum_x x_j (\sigma(z) - y). \tag{61}$$

これは美しい表現です。これを見ると、重みが学習する度合いは$\sigma(z) - y$つまり出力における誤りによってコントロールされてることがわかります。つまり誤りが大きくなればなるほどニューロンの学習は速くなります。これはちょうど我々が直感的に望んでいたことです。特に、二次コスト関数の同じような式、つまり式(55) で$\sigma'(z)$が学習の速度低下の原因となっていたがそれを防いだのです。クロスエントロピーを使うと、$\sigma'(z)$の項が消え、そこのことを心配する必要がなくなるのです。この項の消滅はクロスエントロピーで起こる特別な奇跡です。実際には奇跡ではありません。あとでわかるように、クロスエントロピーはこの性質を持つように特別に選ばれたのです。

同じようにバイアスについての偏微分も計算します。また詳細を追うことをしませんが、次の式が確認できます。

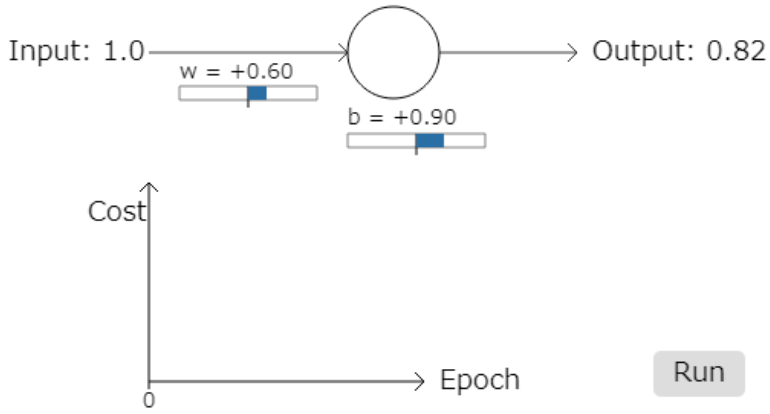
$$\frac{\partial C}{\partial b} = \frac{1}{n} \sum_x (\sigma(z) - y). \tag{62}$$

繰り返しますが、これにより二次コストの同じような式、つまり式(56) で $\sigma'(z)$が学習速度低下の原因になるのを防いでいます。

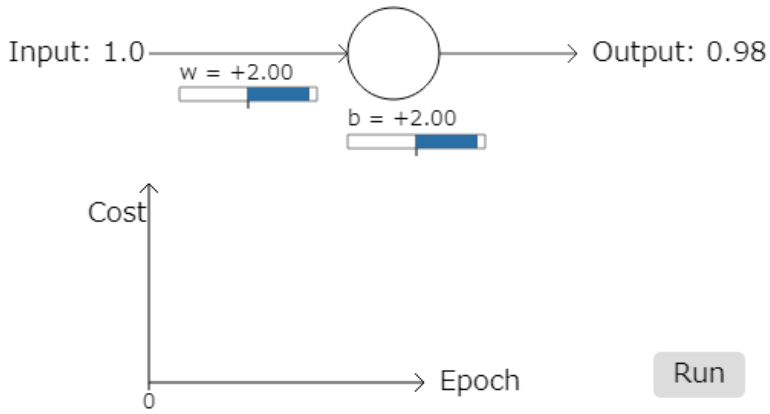
演習

- $\sigma'(z) = \sigma(z)(1-\sigma(z))$であることを確認しなさい

前に遊んでみたおもちゃのような例にもどり、二次コスト関数の代わりにクロスエントロピーを使うとどうなるか見てみましょう。方向転換して、まずは二次コスト関数がうまく行くケースとして、重みが0.6、バイアスが0.9のときから始めましょう。



驚くまでもなく、この例では前と同じようにニューロンは完璧に学習します。では今度は、以前にニューロンが停滞したケース(link, for comparison)、つまり重みとバイアスが両方とも2.0から始めるケースを見てみましょう:



うまくいきました！今度はニューロンは望んだ通り早く学習します。近づいて見ると、コスト曲線の坂は、二次コスト関数の対応する初期値の平らな部分と比べると、もっと急になっていることがわかります。その傾きはクロスエントロピーがうまくやってくれていて、ニューロンが最も速く学習して

ほしい時に停滞してしまうこと、つまりニューロンが間違った起動のしかたをすることを防いでくれます。

例で使われた学習率とはなにかについていってませんでした。以前、二次コスト関数では$\eta = 0.15$を使っていました。同じ例でも同じ学習率を使うべきでしょうか？ところが、コスト関数を変更してしまうと、学習率が「同じ」とはどういうことかを正確には言えません。リンゴとオレンジを比べているようなものです。両方のコスト関数に対して、なにが起こっているかわかるような学習率を、単純に私は経験的に見つけていたというだけです。

そんなことではグラフに意味がないではないかと、あなたは反対するかもしれません。学習率の選択法が決まっていないのに、ニューロンの学習する速さに誰が興味を持つだろうか！？そのような反対意見は重要な点を見逃しています。グラフで重要な点は学習の絶対的な速度についてではないのです。学習の速度がどのように変わっていくかが重要なのです。取り立てて言うと、二次コスト関数は、後に正しい出力値に近づいたときと比べて、ニューロンがあきらかに間違っている状態の時に**遅い**のです。一方で、クロスエントロピーは、あきらかに間違っている状態の時にも速いです。これらの話は学習率がどのように設定されたかに依存しません。

私達は、一つのニューロンのときのクロスエントロピーを調べてきました。しかし、多層の多くのニューロンの場合に汎化するのは簡単です。とくに、$y = y_1, y_2, \ldots$がニューロンの望まれる出力だと仮定しましょう。つまり、ニューロンの最終層の出力です。また、$a^L_1, a^L_2, \ldots$がニューロンの実際の出力だとしましょう。ここで、クロスエントロピーを次の式で定義します。

$$C = -\frac{1}{n} \sum_x \sum_j \left[y_j \ln a^L_j + (1-y_j) \ln (1-a^L_j) \right]. \tag{63}$$

これは以前の表現、式(57)と比べて、すべてのニューロンについての和$\sum_j$をとったことをのぞいて同じです。微分を明示的に計算することはしませんが、式(63)を使えば多くのニューロンのネットワークで学習の速度低下を防げることはすばらしいことでしょう。もし興味があれば、後述する問題で微分を計算してみることはできます。

二次コスト関数の代わりにクロスエントロピーを使うべきなのはいつでしょうか？ところが、出力ニューロンがシグモイドニューロンであるときには、クロスエントロピーはほとんどいつもよりよい選択なのです。なぜかを理解するためには、ニューロンを設定するときには、通常重みとバイアスを同じ種類の乱数で初期化するということを考慮しなければいけません。そのような初期値は、学習の入力により決定的に悪くなる可能性があります。

つまり、出力ニューロンが0であるべき場合に1付近で止まってしまったり、その逆もあり得るということです。二次コスト関数を使っていたとしたら、学習は遅くなります。重みは他の入力から学ぶことを続けるので、学習が完全に停止することはありませんが、それは明らかに私達が望んでいないことです。

演習

- クロスエントロピーについて気をつけなければいけないのは、最初はyとaのそれぞれの役割を覚えるのが難しいところです。正しい式が$-[y \ln a + (1-y) \ln (1-a)]$であったか、$-[a \ln y + (1-a) \ln (1-y)]$であったかという点はよく混乱します。これらの後者の式の方で$y = 0$または$y = 1$だった場合なにが起こるでしょう？このことは前者の式に悪影響があるでしょうか？理由を考えてください。

- この節の最初の一つのニューロンでの考察で、もし学習データの入力で$\sigma(z) \approx y$ならばクロスエントロピーは小さくなるという議論をしました。この議論はyは0または1であるという仮定の元にでした。この仮定は分類問題では通常は正しいですが、他の問題（例えば回帰問題）ではyはときに0から1の間の値をとることがあります。すべての学習の入力について、クロスエントロピーは$\sigma(z) = y$で最小になることを示してください。これが成り立つときクロスエントロピーは次の値をとります：

$$C = -\frac{1}{n} \sum_x [y \ln y + (1-y) \ln (1-y)]. \tag{64}$$

値$-[y \ln y + (1-y) \ln (1-y)]$は バイナリエントロピー と呼ばれることもあります。

問題

- **多層複数ニューロンのネットワーク** 前章で導入した記法を使って、出力層で二次コスト関数の重みについての微分は次の式になることを示しなさい：

$$\frac{\partial C}{\partial w^L_{jk}} = \frac{1}{n} \sum_x a^{L-1}_k (a^L_j - y_j) \sigma'(z^L_j). \tag{65}$$

ここで$\sigma'(z^L_j)$の項は、出力ニューロンが悪い値で停滞した時に学習の速度低下を引き起こします。クロスエントロピーについて、一つの学習データの例xについての出力誤差δ^Lの重みは次の式で表されることを示しなさい：

$$\delta^L = a^L - y. \tag{66}$$

この式を使って、出力層の重みについての偏微分は次の式で与えられることを示しなさい：

$$\frac{\partial C}{\partial w^L_{jk}} = \frac{1}{n} \sum_x a^{L-1}_k (a^L_j - y_j). \tag{67}$$

$\sigma'(z^L_j)$の項が消え、ニューロンが一つでなく多層複数ニューロンの場合でも、クロスエントロピーは学習の速度低下を避けることがわかりました。この解析の簡単な変形でバイアスについても同じことがいえます。もしそれが明らかではないなら、同様の解析を続けてやるべきです。

- 出力に線形ニューロンがあるときの二次コストの使用 多層複数ニューロンのネットワークがあるとします。最終層のニューロンはすべて**線形ニューロン**であると仮定します。線形ニューロンとは、シグモイド関数が適用されず、出力が単純に$a^L_j = z^L_j$であることを意味します。二次コスト関数を使えば、一つの学習データの例xに対する出力誤差δ^Lは次の式で与えられることを示しなさい：

$$\delta^L = a^L - y. \tag{68}$$

前の問題と同様に、この式を使うと、出力層での重みとバイアスについての偏微分は次の式で与えられることを示しなさい：

$$\frac{\partial C}{\partial w^L_{jk}} = \frac{1}{n} \sum_x a^{L-1}_k (a^L_j - y_j) \tag{69}$$

$$\frac{\partial C}{\partial b^L_j} = \frac{1}{n} \sum_x (a^L_j - y_j). \tag{70}$$

このことにより、もし出力ニューロンが線形ニューロンならば、二次コスト関数は学習の速度低下について何の問題も産まないことがわかります。このケースでは、二次コスト関数はそれどころか使うべき適切なコスト関数です。

クロスエントロピーを使ったMNISTの分類

最急降下法とバックプロパゲーションを使って学習プログラムの一部として、クロスエントロピーを実装するのは簡単です。そのことはこの章のあとで、MNISTの手書き数字の分類に関する前述のプログラムnetwork.pyの改善版を開発することで示します。新しいプログラムはnetwork2.pyという名前で、これはだたクロスエントロピーを取り込んだだけではなく、この章で使ったいくつかのテクニックを取り込んでいます*。ここで、私達の新しいプログラムがどのくらいうまくMNISTの数字を分類するかを見てみまし

*コードはGitHub上から手に入れることができます。

よう。第1章と同様に、30個の隠れニューロンを持つネットワークを使い、ミニバッチサイズとして10を使います。学習率を$\eta = 0.5$*とし、30エポック学習させます。network2.pyのインターフェースはnetwork.pyと少し違いますが、何が起こっているかはわかると思います。ところで、network2.pyのインターフェースに関するドキュメントは、Pythonのシェルから help(network2.Network.SGD)と入力することで見ることができます。

```
>>> import mnist_loader
>>> training_data, validation_data, test_data = ¥
... mnist_loader.load_data_wrapper()
>>> import network2
>>> net = network2.Network([784, 30, 10], cost=network2.CrossEntropyCost)
>>> net.large_weight_initializer()
>>> net.SGD(training_data, 30, 10, 0.5, evaluation_data=test_data,
... monitor_evaluation_accuracy=True)
```

ここで注意ですがnet.large_weight_initializer()コマンドは第1章で示したのと同じように重みとバイアスを初期化します。このコマンドを実行しなければいけないのは、この章で後ほど、ネットワークの重みのデフォルト初期値を変更するからです。上記のコマンド列を実行した結果は95.49%の的中率になります。これは1章で二次コストによって得られた的中率95.49%にとても近いです。

100個の隠れニューロンを使い、クロスエントロピーを使いそれ以外のパラメータは同じにしたケースを見てみましょう。この場合、的中率は96.82%になります。これは二次コストを使って的中率96.59であった1章の結果からすると大きな改善です。これは小さな変化に見えるかもしれませんが、誤り率は.41%から3.18%に下がっています。これは元の誤り14個につき1個を消したということです。これはとてもありがたい改善です。

クロスエントロピーコストは二次コストと比べて同等かより良い結果をもたらすと言いたくなるかもしれません。しかし、これらの結果はクロスエントロピーがより良い選択であることを証明したと結論付けるわけにはいかないのです。なぜなら、学習率やミニバッチサイズのようなハイパーパラメータの選定にまだ十分な努力をしていないからです。この改善に本当に説得力を持たせるには、そのようなハイパーパラメータを徹底的に最適化しなければいけません。とはいってもここまでの結果は、クロスエントロピーは二次コストよりよいのではという前述の議論を後押し補強するものです。

ところでこれは、この章を通して見られる一般的なパターンであり、さらにはこの本の残りを通しても見られるパターンです。我々は新しいテクニックを展開し、それを試し、そして「改善した」結果を得るということを行います。もちろんそのような改善はよいことです。しかし、その改善の解釈はいつも問題があります。他のハイパーパラメータの最適化に莫大な努力をして改善が見られるときに、初めてそれらのテクニックは真の説得力を持

*1章では二次コスト関数を使い、学習率$\eta = 3.0$としました。前の議論により、コスト関数が違うときに「同じ」学習率を使うというのはどういうことか、正確に言うことはできません。両方のコスト関数について、私は実験をして、与えられたハイパーパラメータに対してほぼ最適なパフォーマンスを出す学習率を見つけました。

クロスエントロピーと二次コストの学習率については、非常に大雑把な一般的な発見的手法が存在します。前に見たように、二次コストの勾配項の中には、$\sigma' = \sigma(1-\sigma)$という余計な項があります。この値を$\sigma$について平均をとってみると、$\int_0^1 d\sigma \sigma(1-\sigma) = 1/6$となります。これにより、(非常に大雑把にいうと)二次コストは同じ学習率について、平均すると6倍遅く学習するということがわかります。このことは、妥当な出発点は二次コストの学習率を6で割ることだということを示唆しています。もちろん、この議論は厳密からは程遠いですし、あまり真剣にとらえるべきではないです。とはいえ、それは時には役に立つ出発点となります。

ちます。それは大変な量の仕事で、多くの計算資源を必要とするので、通常はそのような網羅的な調査は行いません。そのかわり、いままでやってきたように非形式的なテストを行います。一方で、そのようなテストは絶対的な証明としては不十分だということを忘れず、議論が崩壊しそうな兆候には警戒し続けてください。

ここまでクロスエントロピーについて長い間議論してきました。MNISTの結果にほんの少しの改善をするだけなのに、なぜそんなに労力をつかうのでしょう? この章の後で他のテクニック、特に正規化などを見ますが、それらはもっと大きな改善をします。ならばどうしてクロスエントロピーにそんなに力をいれるのでしょう? 理由の一つは、クロスエントロピーは広く使われているコスト関数でなので、よく理解する価値があるからです。しかしもっと大事な理由は、ニューロンの飽和がニューラルネットで重要な問題で、その問題はこの本を通して繰り返し触れることだからです。そのため、私はクロスエントロピーについて長く議論してきましたが、それはニューロンの飽和を理解し、どのように処理されるべきかを理解するために良い実験室であったのでした。

クロスエントロピーは何を意味するのか? それはどこから来たのか?

クロスエントロピーについての私達の議論は、代数的な分析と実用的な実装に集中してきました。それは役に立ちますが、もっと大きな思想的な問題が解かれていません。例えば、クロスエントロピーは何を意味するのか? クロスエントロピーを直感的に考える方法があるだろうか? どうやったらクロスエントロピーのようなものを最初に思いつくのか? といったことです。

これらの疑問のうち最後のもの:どうやったらクロスエントロピーのようなものを最初に思いつくのか?から始めましょう。前に述べたように学習の速度低下をすでに発見していて、その原因は式(55) and (56)の$\sigma'(z)$の項であるということまで分かっているとしましょう。これらの式を少しの間見ていると、どのようなコスト関数を選べば$\sigma'(z)$の項を消すことができるかを考えるようになるかもしれません。その場合、一つの学習データのサンプルxに対してコスト$C = C_x$は次を満たさなければいけません。

$$\frac{\partial C}{\partial w_j} = x_j(a-y) \tag{71}$$

$$\frac{\partial C}{\partial b} = (a-y). \tag{72}$$

もし、これらの式が成り立つようなコスト関数が選べたとすると、初期誤りが大きければニューロンはより速く学習するという直感を簡単に実現する

ことになります。またそのことは、学習の速度低下の問題も解決します。実際、この式から始めて単純に数学的な考えを進めると、クロスエントロピーの式を導くことができます。このことを見るため、まずは合成関数の微分の法則を考えます。

$$\frac{\partial C}{\partial b} = \frac{\partial C}{\partial a}\sigma'(z). \tag{73}$$

$\sigma'(z) = \sigma(z)(1-\sigma(z)) = a(1-a)$ということを使うと、2つ目の式は次のようになります。

$$\frac{\partial C}{\partial b} = \frac{\partial C}{\partial a}a(1-a). \tag{74}$$

式(72)と比べて、次を得ます。

$$\frac{\partial C}{\partial a} = \frac{a-y}{a(1-a)}. \tag{75}$$

この式をaについて積分すると次のような積分定数を持つ式を得ます。

$$C = -[y\ln a + (1-y)\ln(1-a)] + \text{constant}, \tag{76}$$

これが、学習データの一つサンプルxのコストに対する寄与です。コスト関数全体を得るには、学習サンプルの全てについて和をとり、次のようになります。

$$C = -\frac{1}{n}\sum_x [y\ln a + (1-y)\ln(1-a)] + \text{constant}, \tag{77}$$

ここでの定数はそれぞれの学習データのサンプルに対する定数の平均値です。式(71)と(72)により、クロスエントロピーの式を定数項を除いて唯一に定めます。クロスエントロピーは薄い空気の中から奇跡的に生まれるものではないのです。むしろ単純かつ自然に発見されるべきものなのです。

クロスエントロピーの直感的な意味はなんでしょうか? どうやったらそのようなことを考えられるのでしょう? このことを深く説明するのは、行きたいところよりさらに遠くに連れて行かれてしまうかもしれません。しかしクロスエントロピーは情報理論から来ているのですが、それを解釈する標準的方法があるということに触れておくことは価値があるでしょう。大雑把にいうと、クロスエントロピーとは驚きの尺度です。特に私達のニューロンは関数 $x \rightarrow y = y(x)$を計算しようとしています。しかしそのかわり関数 $x \rightarrow a = a(x)$を計算しようとしています。ここでaはニューロンが計算したyが1である確率だとして、同じように$1-a$は計算されたyが0である確率だとしましょう。そうすると、クロスエントロピーは真のyの値を学習するときに平均的にどのくらい「驚き」を得るかを示します。出力が期待したどおり

だとあまり驚かないし、期待していないものだと強く驚きます。もちろん私は「驚き」を正確に説明していないので、これは中身のないおしゃべりに見えるかもしれません。しかし実際に、驚きが何を意味するかを説明する、正確な情報理論的手法があるのです。これについて、良い、短い、自己完結したオンラインで入手可能な議論を私は見たことがないです。でももしこれについて掘り下げたいのなら、ウィキペディアは簡単なまとめがあり、これにより正しい方向に向かうことができるでしょう。情報理論についての本「Cover and Thomas」の5章に書かれているKraft不等式についての説明をあたってみれば、さらに詳細を埋めることができるでしょう。

問題

- 学習に二次コストを使ったネットワークで出力ニューロンが飽和したときに学習の速度低下が起こることを時間をかけて議論してきました。学習を抑制する他の因子は、式(61)のx_jの項の存在です。この項により、入力x_jがゼロに近ければ対応する重みw_jの学習は遅くなります。賢いコスト関数を選んでも、このx_jの項を消すのは不可能であることを説明してください。

過適合と正規化

ノーベル賞受賞者の物理学者エンリコ・フェルミはあるとき、とある物理の未解決問題の解としてあるグループが提案した数学的モデルについての意見を求められました。そのモデルは実験と非常に良い一致をみせていましたが フェルミは懐疑的でした。フェルミは、そのモデルには自由に設定できるパラメータがいくつあるか、と尋ねました。答えは4つ、ということでした。そこでフェルミはこう答えたそうです。:「私の友人ジョン・フォン・ノイマンが言っていたよ。私ならパラメータが4つあれば象だってフィッティングできる、5つあれば象の鼻を振れる、とね。」

*このエピソードは、フリーマン・ダイソン のチャーミングな文章から引用したものです。ダイソンはまさにフェルミが批判したモデルの提案者の一人でした。4パラメータの象は ここ にあります。

この話のポイントは、もちろん、モデルの自由パラメータの数が多ければ、驚くほど多くの現象を説明できてしまう、という点です。たとえ手元のデータと良く一致したとしても、そのようなモデルが良いモデルだとは一概には言えません。もしかすると、そのモデルは、与えられたデータの背後にある現象になんら本質的な洞察を与えることなく、パラメータの多さにまかせてとりあえず フィットできてしまうだけなのかもしれません。もしこれが起こっているなら、このモデルは既存のデータに対してはよくあてはまりますが、新しい状況への汎化に失敗するでしょう。モデルの真価は、そのモデルが経験したことないような状況での予言力で測られるのです。

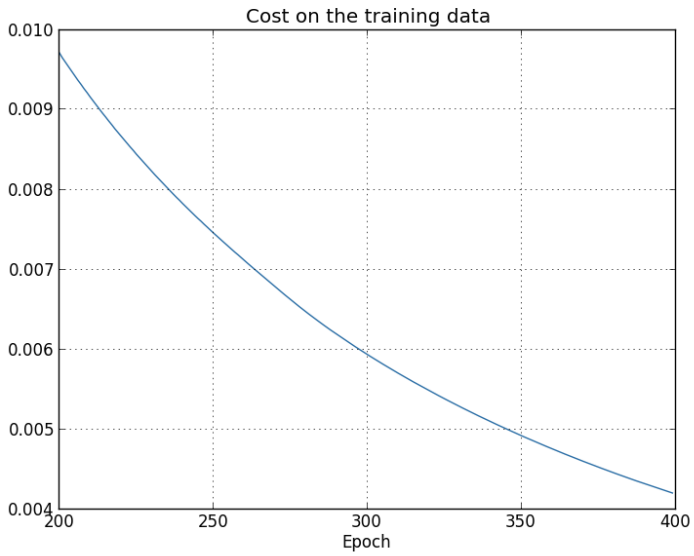
フェルミとフォンノイマンは、4つしかパラメータのないモデルに対してすでに懐疑的でした。30個の隠れニューロンを持つ私たちのMNIST分類ネットワークには、なんと約24,000個ものパラメータがあります！ うわっ、私のパラメータ、多すぎ？100個の隠れニューロンがあるネットワークなら、パラメータの数は8万個近くにもなり、そして最新の深層学習ニューラルネットにはときに何百万、何億という数のパラメータが含まれます。こんなものを信じて、大丈夫でしょうか？

この問題の具体例として、私たちのニューラルネットワークが新しい状況に適応できない状態というのを 実際に作ってみましょう。まず、使うのは隠れニューロン30個、パラメータが23,860個のモデルです。ただし、MNISTの50,000個の画像を全部使わず、先頭1,000個の画像だけを使います。データ集合を制限することで、汎化失敗現象が見やすくなります。訓練方法は以前と同様、学習率$\eta = 0.5$、ミニバッチサイズ10を採用します。ただし、訓練例が少ないぶん、訓練期間は400エポックと、以前より長くします。コスト関数の変化のようすを見るためにnetwork2を使いましょう:

```
>>> import mnist_loader
>>> training_data, validation_data, test_data = ¥
... mnist_loader.load_data_wrapper()
>>> import network2
>>> net = network2.Network([784, 30, 10], cost=network2.CrossEntropyCost)
>>> net.large_weight_initializer()
>>> net.SGD(training_data[:1000], 400, 10, 0.5, evaluation_data=test_data,
... monitor_evaluation_accuracy=True, monitor_training_cost=True)
```

この結果を用いて、学習の進行に対するコスト関数の変化をプロットしてみます*：

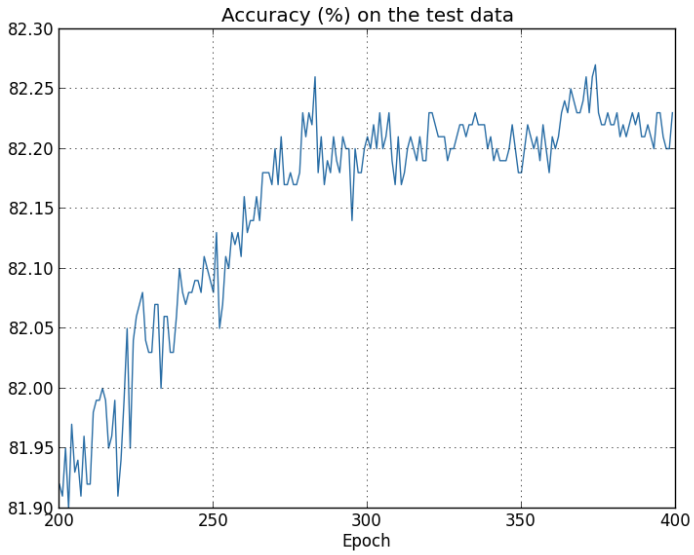
*このグラフと、次の4つのグラフは overfitting.py から生成しています。



期待通り、コストはなめらかな減少を示しています。学習後期のふるまいを拡大して観察したいので、上図ではエポック200 - 399だけを表示して

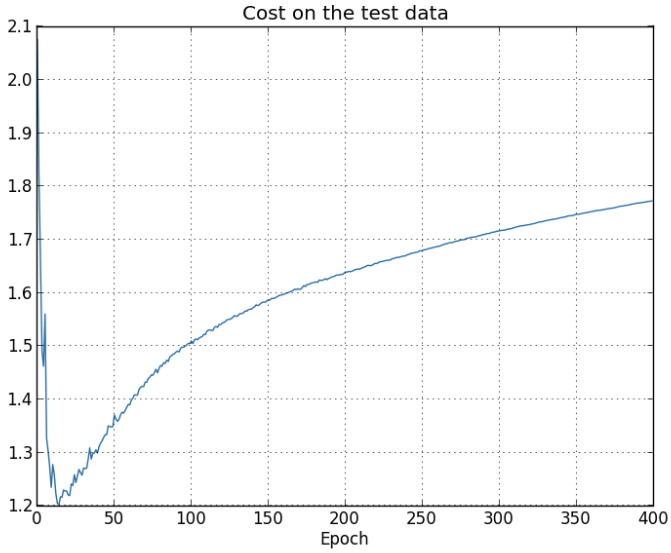
います。この学習後期で起こっている面白い現象を、これからみていきます。

こんどは、試験データの分類精度がどう変化しているか見てみましょう:



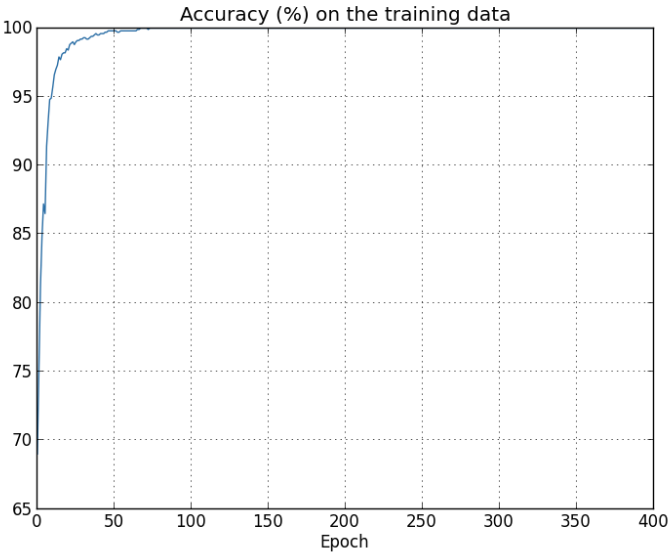
この図も、かなり拡大して見せています。図の範囲に入る以前の、最初の200エポックで 精度は82パーセント直前まで上昇しています。その後、学習はだんだん鈍化し、ついに280エポック付近で分類精度はほとんど改善がみられなくなります。以降は、280エポックにて達成した精度のまわりに、小さな統計的ゆらぎが見られるだけになります。このグラフと、訓練データのコスト関数がなめらかに減少しつづける以前のグラフを見比べてください。コスト関数を見るかぎり、モデルの性能はいっけん良くなり続けているようです。しかし試験データの分類精度をみると、コスト関数でみられた「改善」は幻であることがわかります。エポック280以降のニューラルネットワークが学習した知識は、フェルミがディスっていたモデルと同じく、試験データに汎化できない知識だったのです。したがって、有用な学習ではなかった、といえます。このようなとき、エポック280以降のニューラルネットワークは **過適合**している、**過学習**している、などと言います。

もしかして、訓練データの**コスト関数**と、試験データの **分類精度**という、異なるものを比較したのがいけなかったのかもしれません。それでは、訓練データのコスト関数と試験データのコスト関数同士を較べたらどうでしょうか？ 逆に、訓練データと試験データの分類精度を比べたらどうでしょうか？実は、どの方法で比較しても、同じような過学習の兆候が見られます。ただし、細部には確かに違いがあります。例えば、試験データのコスト関数を見てみましょう:



図から、試験データのコスト関数はエポック15あたりまでは改善していくが、その後は実は 悪化しはじめていたことがわかります。いっぽうで訓練データのコスト関数は改善しつづけていますから、これもまた過適合の兆候であるといえます。もっとも、ここで1つの疑問がわいてきます。学習は どの時点から過適合に陥ったと見做すべきでしょうか？ エポック15からでしょうか、それともエポック280？ 実用的な観点からいえば、私たちの本来の目的は試験データの分類精度を向上させることであって、試験データのコスト関数は分類精度の間接的な指標にすぎません。ですから、エポック280をもって 学習が過適合に陥った時点と見做すのが最も合理的といえます。

訓練データの分類精度にも、過学習の兆候が現れています:



精度が100パーセントまでひたすら向上しています。つまり、私たちのニューラルネットワークは 1,000件の訓練画像をすべて正しく分類しているのです！いっぽう、試験データの分類精度は せいぜい82.27パーセント

程度が最大です。つまり、私たちのニューラルネットワークはもはや 数字の認識一般を学習しているのではなく、訓練データ画像に特有の癖を学習してしまっているのです。数字とは何かを理解し試験データにも汎化できるような理解を得ようとせずに、ただ訓練データを丸暗記してしまっている、と言ってもよいでしょう。

過適合はニューラルネットワークの持つ大問題です。特に、現代のニューラルネットワークはしばしば極めて多数の重みやバイアスをパラメータとして持つため、過適合の問題が顕著になります。ニューラルネットワークを効率的に学習させるためには、過適合の進行を検知し、過学習を避ける手立てが必要です。そして、過適合の影響を軽減するための手法が望まれます。

過適合を検知する安直な方法は、上述したアプローチ、つまり、ニューラルネットワークを訓練しながら試験データに対する精度の変化を追跡することです。もし訓練によって試験データに対する精度が向上しなくなったら、その時点で訓練を止めるべきです。もちろん厳密に言えば、これを過適合の兆候であると断言することはできません。試験データと訓練データに対する精度の改善が同時に止まっているのかもしれません。それでも、この戦略は過適合を防ぐ手段としては有効です。

実際に、私たちはこの戦略の変種を採用します。MNISTのデータをロードする時に、私たちは3つのデータセットをロードしていることを思い出しましょう:

```
>>> import mnist_loader
>>> training_data, validation_data, test_data = ¥
... mnist_loader.load_data_wrapper()
```

これまでのところ、私たちは training_data と test_data のみを用いて、validation_data を無視してきました。validation_data は数字の画像を 10,000 枚含んでいます。これらの画像は、MNISTの訓練データに含まれる 50,000 枚の画像や試験データに含まれる 10,000 枚の画像とは異なるものです。過適合を防ぐために、私たちは test_data の代わりに validation_data を用いて、先程述べた戦略を実行することにします。つまり、各エポックが終了する毎に、validation_data に対する分類精度を計算します。validation_data に対する分類精度が飽和したところで、ニューラルネットワークの訓練を終了します。この戦略を**早期打ち切り**と呼びます。もちろん、実際問題としては精度が飽和したことを即座に検知する術はありません。その代わりに、精度が飽和したと確信が持てる時点まで、訓練を続けることにします*。

過適合を防ぐために、なぜ test_data ではなく validation_data を用いるのでしょうか？ 実は、過適合を防ぐために採用したこの戦略は、より一般的

*いつ止めるべきか決めるには幾つかの判断が必要となります。先ほど示したグラフでは、私はエポック 280を精度が飽和した地点であると定めました。この判断が悲観的過ぎて、実際にはまだ学習の余地が残っている可能性もあります。時たま、ニューラルネットワークの訓練中にしばらく学習が停滞して、その

な戦略の一部分なのです。より一般的な戦略とは、訓練を続けるエポック数、学習率、最適なニューラルネットワークのアーキテクチャ等の ハイパーパラメータを評価・比較するために、validation_data を用いるというものです。そうして得られた評価を元に、良いハイパーパラメータを見つけて定めます。実際、これまで触れてきませんでしたが、この本でこれまでに採用してきたハイパーパラメータの一部は、私自身がこの戦略に基づいてたどり着いたものです。(これに関する詳細は後ほど)

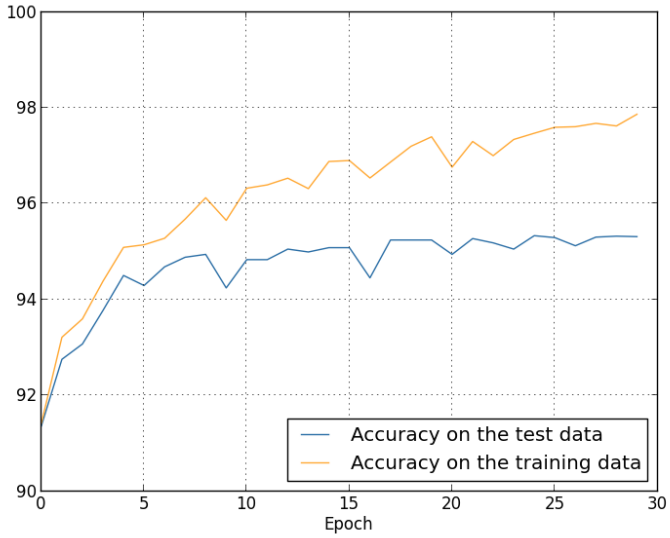
後再び改善する場合もあるのです。もし仮に、エポック400を過ぎてからさらなる学習が起こったとしても驚きはしません。ただし、そのような改善はあったとしても小さなものでしょう。このように早期打ち切りの判断に絶対は無く、多少楽観的な戦略を取ることも可能なのです。

もちろん、この説明では全く元の質問に答えたことになっていません。過適合を防ぐために test_data ではなく validation_data を使うのはなぜか、という質問を、より一般的な質問、つまり、良いハイパーパラメータを見つけるために test_data ではなく validation_data を使うのはなぜか、というものに 置き換えただけなのです。この質問への回答を理解するために思い出してほしいのは、私たちが良いハイパーパラメータを探すときには、普通、様々な異なる値のハイパーパラメータを試しては評価を行うだろうということです。その際、ハイパーパラメータの評価を test_data で行ったとすると、ハイパーパラメータが test_data に対して過適合してしまう恐れがあります。つまり、最終的なハイパーパラメータは test_data の持つ特有の癖を学習してしまい、ニューラルネットワークが学習した成果を他のデータセットに対して汎化することができない、ということが起こりうるのです。ハイパーパラメータの決定に validation_data を用いることで、このような事態を防ぎます。そうして望ましいハイパーパラメータが得られたところで、test_data を用いた 最終的な精度評価を行います。このような手順を踏むことで、test_data に対する 評価が一般的なデータに対してニューラルネットワークが発揮するパフォーマンスの正当な評価であると、自信を持って結論できるのです。言い方を変えると、検証データ (validation_data) は 良いハイパーパラメータを学習するために使われるある種の訓練データ (training_data) であると 言えます。このようなアプローチは、validation_data を training_data から 除外する ("hold out") ことから、**ホールドアウト法**と呼ばれます。

実際問題として、ニューラルネットワークのパフォーマンスを test_data によって 評価してしまった後になってから、当初の課題に対して異なるアプローチを試してみたくなる かもしれません。その時には、ニューラルネットワークの異なるアーキテクチャを試みて、新たに良いハイパーパラメータを探しなおすことになるでしょう。このようなことをしてしまうと、結局 test_data に対して過適合してしまう危険は無いのでしょうか？ 私たちが結果の一般性に自信を持つためには、潜在的には無限に遡ってデータ・セットを用意する 必要があるのでしょうか？この疑問に真剣に取り組むのは、深く難しい問題です。しかし、私たちの実用上の目的に対して必要なことではないので、ここでこの疑問について深追いすることはしま

せん。むしろ、training_data、validation_data、そして test_data に基づく基本的なホールドアウト法を用いて、積極的に前進したいと思います。

ここまでで、訓練画像を1,000枚に限った時に起こる過適合を見てきました。では、訓練データが 含む50,000枚の画像全てを用いた時に何が起こるでしょうか？画像の枚数以外のパラメータはそのままに（30個の隠れニューロン、学習率 0.5、ミニ・バッチの大きさは 10）、50,000枚の画像全てを用いて30エポックに渡り訓練してみます。ここに示しているのは、訓練データと試験データに対する分類精度のグラフです。ここでは、以前のグラフと今回の結果との直接的な比較を行うために、検証データではなく試験データを用いています。



明らかに、たった1,000個の訓練例を用いた時と比べると、試験データと訓練データに対する 精度がずっと近いままでいることがわかります。訓練データに対する最大の分類精度は97.86パーセントですが、これは試験データに対する精度 95.33パーセントと1.53パーセントしか違いません。1,000枚の訓練例に 限った時に生じた17.73パーセントのギャップを思い出してください！過適合は進行しているにしろ、大幅に軽減されています。私たちのニューラルネットワークは、訓練データから試験データに対して、ずっとよく汎化できています。一般的に言って、過適合を軽減する最良の方法の一つは、より多くの訓練データを用意することです。十分な訓練データがあれば、とても大きなニューラルネットワークでさえ 過適合を起こすことはないでしょう。ただし残念ながら、訓練データはしばしば高価だったり入手困難だったりするため、大量の訓練データを手に入れることは常に現実的な選択肢とは限りません。

正規化

訓練データを増やすことは、過適合を軽減する一つのやり方ですが、他の方法は無いのでしょうか？ 一つの可能なアプローチは、ニューラルネットワークのサイズを小さくすることです。しかし、大きなニューラルネットワークには小さなニューラルネットワークよりも強力な潜在能力が ありますから、これは積極的に採用するような選択肢ではありません。

幸いにも、ニューラルネットワークや訓練データの大きさを変えなくても、過適合を軽減する方法があります。それらの手法は、**正規化**法として知られています。この節では、**重み減衰**、あるいは **L2正規化**として知られる手法を説明します。L2正規化は最もよく用いられる正規化手法の一つです。L2正規化では、コスト関数に**正規化項**と呼ばれる余分な項を付け足します。これが、正規化されたクロスエントロピーです：

$$C = -\frac{1}{n} \sum_{xj} \left[y_j \ln a_j^L + (1-y_j) \ln (1-a_j^L) \right] + \frac{\lambda}{2n} \sum_w w^2. \quad (78)$$

第1項はこれまで通りのクロスエントロピーの表式です。しかし今回はさらに、第2項としてニューラルネットワークの 持つ全ての重みの2乗和が足されています。この余分の項は全体に $\lambda/2n$ の因子が掛かっています。$\lambda > 0$ は**正規化パラメータ**と呼ばれる実数で、n はいつも通り訓練データの大きさです。この λ をどう選ぶかについては後ほど議論します。また、正規化項はバイアスを**含まない**ことにも 注意しておいてください。これについても後ほど説明します。

もちろん、クロスエントロピー以外のコスト関数を正規化することも可能です。例えば、2乗コスト関数をL2正規化すると以下のようになります：

$$C = \frac{1}{2n} \sum_x \|y-a^L\|^2 + \frac{\lambda}{2n} \sum_w w^2. \quad (79)$$

どちらの場合にも、正規化されたコスト関数を次のように書けます：

$$C = C_0 + \frac{\lambda}{2n} \sum_w w^2. \quad (80)$$

ここで、C_0 は元の正規化されていないコスト関数です。

直感的には、正規化することによりニューラルネットワークがより小さな重みを好むようになります。大きな重みが許されるのは、そうすることがコスト関数の第1項を余程大きく改善する場合だけです。言い換えると、正規化とは重みを小さくすることと元のコスト関数を小さくすることの間でバランスを取る方法であると 見ることもできます。このバランスを取る上で、2つの要素の相対的な重要性を決定するのが λ の値です：λ が小さい時は元のコスト関数を最小化することを好み、λ が大きい時には より小さな重みを好むのです。

さて、実際のところ、そのようなバランスを取ることがなぜ過適合を軽減する助けになるのか、その理由は誰の目にもすぐさま明らかなものではありません。しかし、正規化が実際に過適合を軽減することは分かっています。その理由については次の節で触れることにして、まずは正規化によって過適合が軽減されている例を 調べてみることにしましょう。

そのような例を作るにはまず、どうすれば正規化されたニューラルネットワークに確率的勾配降下法を 適用できるのかを明らかにする必要があります。特に、2種類の偏微分、$\partial C/\partial w$ と $\partial C/\partial b$ をニューラルネットワークが持つ 全ての重みとバイアスに対して計算する方法を知る必要があります。式 (80) の偏微分を取ると、次の表式が得られます:

$$\frac{\partial C}{\partial w} = \frac{\partial C_0}{\partial w} + \frac{\lambda}{n} w \tag{81}$$

$$\frac{\partial C}{\partial b} = \frac{\partial C_0}{\partial b}. \tag{82}$$

偏微分項、$\partial C_0/\partial w$ と $\partial C_0/\partial b$ は、前章で説明した逆伝播法を使って計算できます。したがって、正規化されたコスト関数の勾配は簡単に計算できることがわかります。これまで通りに逆伝播法を使って、その後に重みによる偏微分に対して $\frac{\lambda}{n} w$ を 加えれば良いのです。バイアスに関する偏微分はこれまでと何も変わりません。つまり、バイアスに対する勾配降下法の学習規則は通常の規則そのものなのです:

$$b \rightarrow b - \eta \frac{\partial C_0}{\partial b}. \tag{83}$$

重みに関する学習規則は次のように変わります:

$$w \rightarrow w - \eta \frac{\partial C_0}{\partial w} - \frac{\eta \lambda}{n} w \tag{84}$$

$$= \left(1 - \frac{\eta \lambda}{n}\right) w - \eta \frac{\partial C_0}{\partial w}. \tag{85}$$

これは通常の勾配降下法による学習規則とほぼ同じです。唯一の違いは、まず最初に重み w に因子 $1 - \frac{\eta \lambda}{n}$ を掛けてリスケールしている点です。この因子は重みを小さくするので、**重み減衰**とも呼ばれます。一見、重みがゼロになるまでどこまでも小さくなってしまうかのように見えます。しかし、それは正しくありません。なぜなら、他の項が正規化する前のコスト関数を小さくするために、重みを大きくする方向に働くかもしれないからです。

これで勾配降下法のやり方は分かりました。では確率的勾配降下法は？実際のところ、正規化されていない確率的勾配降下法と同じことをやります。つまり、$\partial C_0/\partial w$ を見積もるのに、m個の訓練例を含むミニバッチに

関して 平均を取れば良いのです。そうして、次のような確率的勾配降下法の正規化された学習規則が得られます (c.f. 式 (20)):

$$w \rightarrow \left(1-\frac{\eta\lambda}{n}\right) w - \frac{\eta}{m}\sum_x \frac{\partial C_x}{\partial w}. \tag{86}$$

ここで、和はミニバッチに含まれる訓練例 x について取り、C_x は各訓練例の (正規化されていない)コストを表します。これは、$1-\frac{\eta\lambda}{n}$ という重み減衰因子を除いて、確率的勾配降下法のこれまで使ってきた規則と全く同じです。最後に念のため、バイアスの学習に関する正規化された学習規則を述べておきましょう。これはもちろん、正規化されていない規則と全く同じです (c.f. 式 (21)):

$$b \rightarrow b - \frac{\eta}{m}\sum_x \frac{\partial C_x}{\partial b}. \tag{87}$$

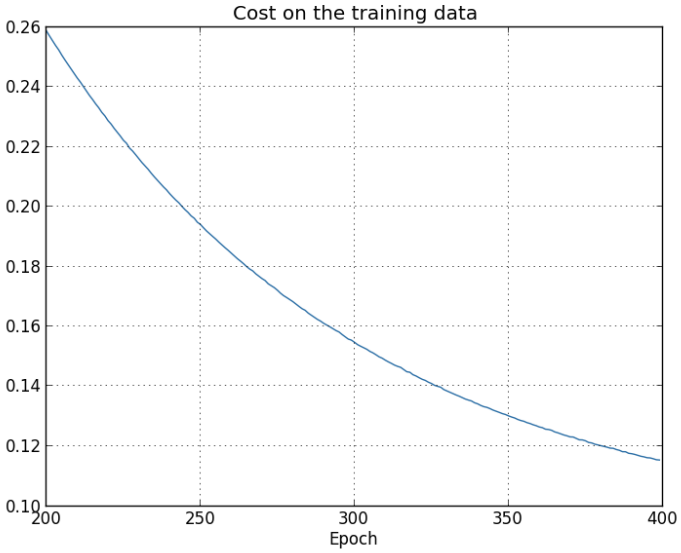
ここで、和はミニバッチに含まれる訓練例 x について取ります。

では、正規化がどのくらいニューラルネットワークのパフォーマンスを変えるのか確かめてみましょう。ここでは、隠れニューロンが30個、ミニバッチの大きさが 10、学習率が 0.5、そして クロスエントロピーをコスト関数として採用したニューラルネットワークを使います。今回はさらに、正規化パラメータとして $\lambda = 0.1$ を使用します。コードの中で正規化パラメータを表す変数名として lmbda を用いている点には気をつけてください。というのも、Pythonでは lambda が正規化パラメータとは関係の無い意味を持つ予約語だからです。また、ここでは再び validation_data ではなく test_data を用いました。既に議論したように、厳密に言えば validation_data を使うべきです。今回 test_data を使うのは、そうすることで正規化する前の結果とより直接的な比較が可能になるためです。このコードを validation_data を使うように書き換えるのは容易ですし、実際にそうしても同様の結果が得られることがわかるでしょう。

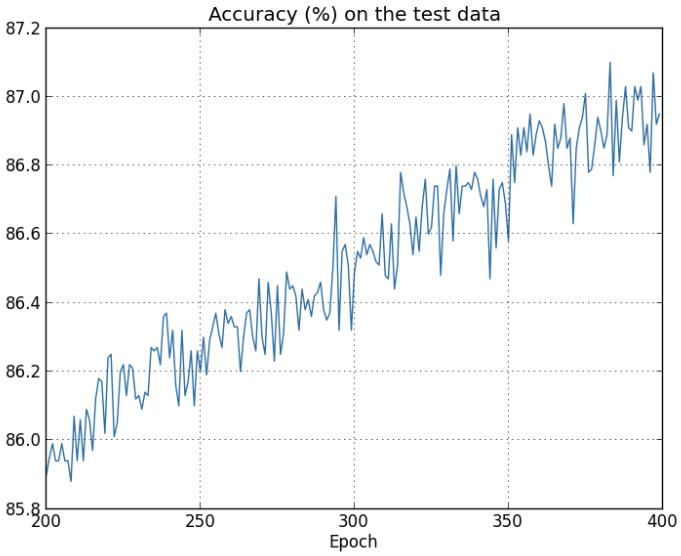
```
>>> import mnist_loader
>>> training_data, validation_data, test_data = ¥
... mnist_loader.load_data_wrapper()
>>> import network2
>>> net = network2.Network([784, 30, 10], cost=network2.CrossEntropyCost)
>>> net.large_weight_initializer()
>>> net.SGD(training_data[:1000], 400, 10, 0.5,
... evaluation_data=test_data, lmbda = 0.1,
... monitor_evaluation_cost=True, monitor_evaluation_accuracy=True,
... monitor_training_cost=True, monitor_training_accuracy=True)
```

訓練データに対するコストは時間とともに減衰し続けることが分かるでしょう。この点は正規化していない以前の結果と変わりません*:

*これとこの後2つのグラフはプログラム overfitting.py で生成しました。



しかし今回は、test_data に対する精度も400エポックが終わるまで向上し続けています:



明らかに、正規化することで過適合が抑制されています。そしてそれ以上に、精度もかなり良くなっています。ピーク時の分類精度が正規化無しで 82.27パーセントだったのに対し、正規化した後は87.1パーセントに向上しています。実は、400エポックを超えて訓練し続けることで、目に見える改善がほぼ間違いなく得られるでしょう。経験的には、正規化することで ニューラルネットワークはより良く汎化するようになり、過適合の効果もかなり軽減されるようです。

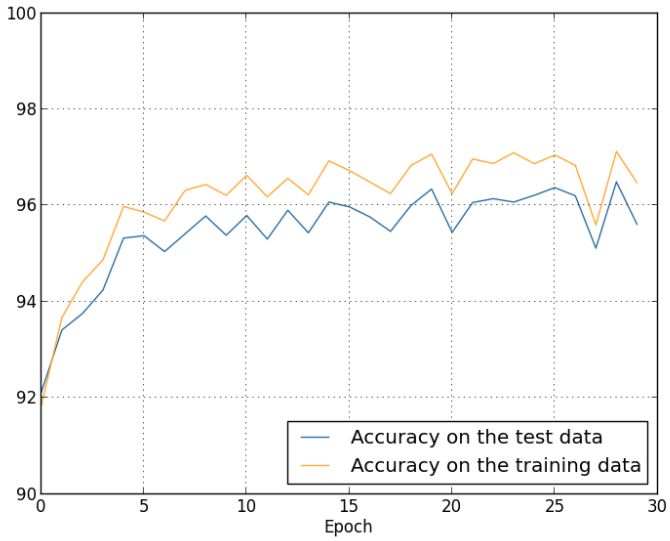
では、訓練画像を1,000枚に制限するのをやめて、50,000枚全ての訓練データを使うことにすると 何が起こるでしょうか。もちろん既に見たとおり、50,000枚全ての画像を使えば過適合はそれほど 問題になりません。その状況で、果たして正規化にご利益があるのでしょうか？ ハイパーパラメータは前回と同じ、30エポック、学習率 0.5、ミニバッチの大きさは

10、を使うことにします。しかし、正規化パラメータは変更する必要があります。なぜなら、訓練データの大きさを $n = 1,000$ から $n = 50,000$ に変更したため、そのことが重み減衰因子 $1 - \frac{\eta\lambda}{n}$ を変えているためです。もし $\lambda = 0.1$ を使い続けたならば、重み減衰はずっと抑えられて、正規化の効果もずっと小さなものになるでしょう。訓練データの大きさの違いを埋め合わせるため、$\lambda = 5.0$ を使うことにします。

さあ、今一度重みを初期化してから、ニューラルネットワークを訓練しましょう:

```
>>> net.large_weight_initializer()
>>> net.SGD(training_data, 30, 10, 0.5,
... evaluation_data=test_data, lmbda = 5.0,
... monitor_evaluation_accuracy=True, monitor_training_accuracy=True)
```

その結果はこちら:



良い知らせがあります。それもたくさん。第一に、試験データに対する分類精度が、正規化する前の 95.49 パーセントから 96.49 パーセントに上昇しています。これは大きな改善です。第二に、訓練データと試験データに対する結果の差が、正規化する前と比べてずっと小さくなっていることが見て取れます。その差は今や 1 パーセントにも 満たない小さなものです。これでもはっきりとした差は残っていますが、過適合を改善するという意味ではかなりの進歩です。

最後に、隠れニューロンを100個にして正規化パラメータを $\lambda = 5.0$ のままにした時、試験データの分類精度がどうなるか見てみましょう。ここでは、過適合に関する詳細な分析は行いません。ただのお楽しみとして、クロスエントロピーコスト関数とL2正規化という2つの新しい手法が どれだけの精度をもたらすのか見てみるのが、ここでの目的です。

```
>>> net = network2.Network([784, 100, 10], cost=network2.CrossEntropyCost)
>>> net.large_weight_initializer()
>>> net.SGD(training_data, 30, 10, 0.5, lmbda=5.0,
... evaluation_data=validation_data,
... monitor_evaluation_accuracy=True)
```

最終的に得られる分類精度は、検証データに対して 97.92 パーセントです。これは隠れニューロンが30個だった時と比べると大きな飛躍です。実は、もう少しだけチューニングして、$\eta = 0.1$ と $\lambda = 5.0$ にセットして60エポック走らせると、検証データに対する分類精度が 98 パーセントの壁を超えて、98.04 パーセントという結果を実現します。たった152行のコードにしては悪くない仕事ですね！

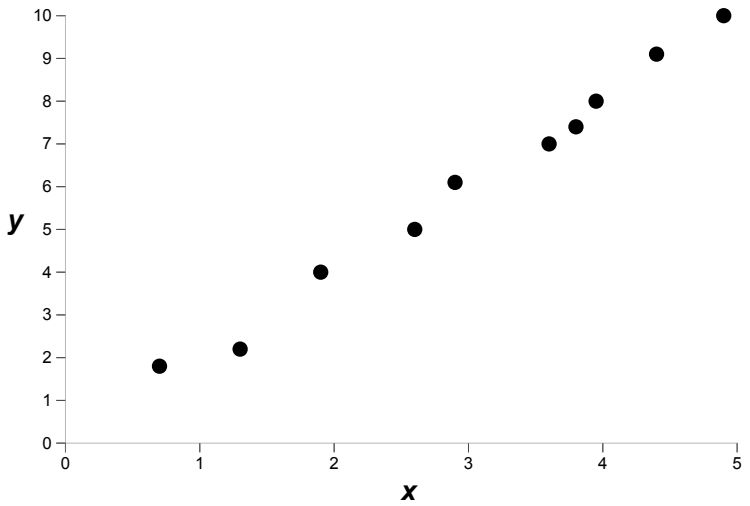
ここまで私は、正規化を過適合を防ぎ分類精度を向上させるための手法として説明してきました。実は、正規化のご利益はそれだけではありません。ここで作成したMNISTを分類するニューラルネットワークを、重みの初期値をランダムに変えながら繰り返し走らせてみましょう。実際にやってみて私が気づいたのは、正規化しないと時折「引っかかる」ことです。つまり、コスト関数の極小に捕らえられているようなのです。結果として、異なる初期値から始めると時々かなり異なる結果に行き着くことになります。対照的に、正規化した場合にはより簡単に再現可能な結果が得られます。

なぜこのようなことが起こるのでしょうか？ヒューリスティックには、もしコスト関数が正規化されていないと、重みベクトルの長さは大きくなりがちです。時間が経つにつれ、実際に重みベクトルはかなり大きくなりえます。重みベクトルが長い時、勾配降下法はその動径方向に向かいやすく、重みベクトルの向きを変える方向にはわずかにしか動きません。このため、一旦重みベクトルが長くなってしまうと、重みベクトルが同じ方向を向いて 動かなくなってしまう可能性があります。この現象が原因で、私たちが使っている学習アルゴリズムでは重み空間を十分に探索することができず、結果的にコスト関数の良い最小値を見つけるのが困難になっているのではないかと、私は考えています。

なぜ正規化で過適合が軽減されるのか？

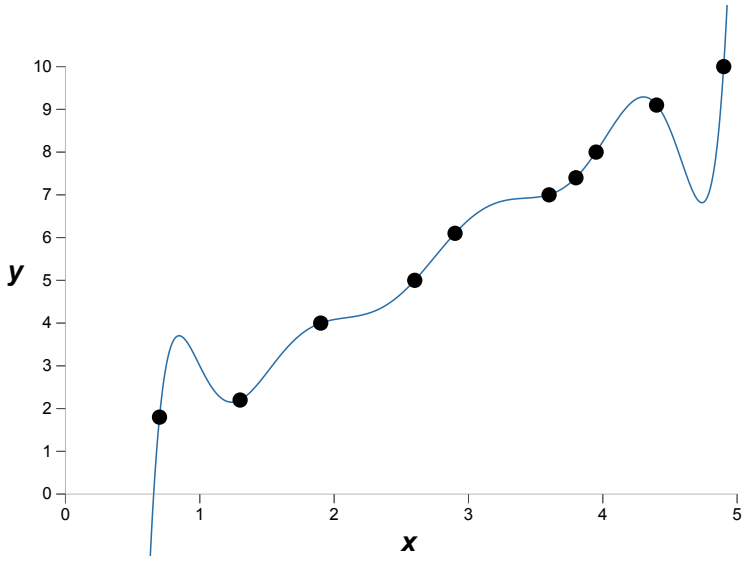
実際に正規化を実装してみることで、確かに正規化によって過適合が軽減されることを見てきました。この結果には勇気づけられますが、残念ながら、なぜ正規化が過適合を軽減してくれるのか明白とは言えません。この点についてしばしばなされる説明は次のようなものです。「重みがより小さいということは、ある意味で、複雑さがより小さいことを意味し、それゆえ、より単純で強力にデータを説明してくれるのだ。」ただ、この説明はずいぶん簡潔で、曖昧で誤魔化されているようにも思えます。ですか

ら、この説明を批判的に検証してみましょう。そのために、次のような単純なデータセットを用意して、これを説明するモデルの構築してみましょう:

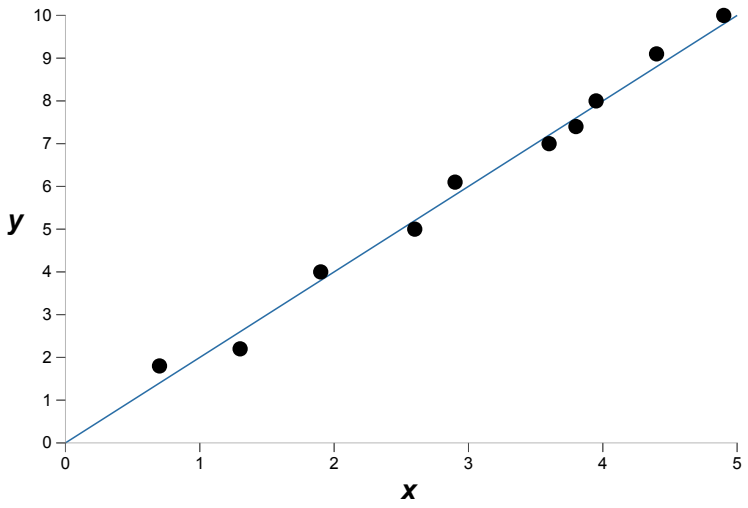


私たちはここで、暗にある現実の現象を調べています。そして、x と y は現実のデータを表しています。ここでの目標は、x の関数として y を予言するモデルの構築です。そのためにニューラルネットワークを用いることも可能ではありますが、今はもっと単純なことを考えます:y を x の多項式としてモデル化してみます。ニューラルネットワークの代わりに多項式を利用するのは、多項式は何が起こっているのか特に見やすいためです。そこで、まずは多項式で起こっていることを理解してから、それをニューラルネットワークに翻訳しようと思います。今、上のグラフには10個のデータ点があるので、これらの点を全て通過する9次多項式 $y = a_0 x^9 + a_1 x^8 + \ldots + a_9$ がただ一つに定まります。これがその多項式のグラフです*:

*ここでは多項式の係数を明示しませんが、例えば Numpy に含まれる `polyfit` 等の ルーチンを使って簡単に係数を計算できます。もし興味があれば、多項式の正確な形はグラフのソースコードで 見ることが出来ます。グラフを生成するプログラムの14行目から定義されている関数 `p(x)` です。



この多項式はデータ点を正確にフィッティングしています。しかし、これらのデータ点は $y = 2x$ という直線を使っても良くフィッティングできます:



この2つのうち、どちらがより良いモデルであると言えるでしょうか？ どちらがよりもっともらしく、真実に近いのでしょうか？ そして、同じ現実の現象の他の例に対して、どちらのモデルがより良く汎化できるでしょうか？

これらは難しい質問です。実は、どの質問に対しても、確証を持って答えることは出来ません。そうするには、背後にある現実の現象に関する情報がもっと必要です。しかし、2つの可能性を考えてみましょう：(1) 実は9次多項式が本当に現実の現象を記述するモデルになっていて、したがって他の例に対しても完璧に汎化できる可能性、そして (2) 正しいモデルは $y = 2x$ なのだが、例えば測定の誤差などに起因するノイズが乗っているため、モデルがデータ点を完全にはフィットしていない可能性、です。

どちらが正解だと言い切る根拠はありません（あるいは、どちらでもない、第3の可能性もありうるかもしれません）。論理的には、どちらでもありうるのです。にも関わらず、両者の違いは決して些細なものではありません。確かに、データ点の与えられている領域では2つのモデルの予言には小さな違いしかありません。しかし、もしも上のグラフで示されているよりももっとずっと大きな x に対して y を予言したいとしたら、どうでしょうか。その時には、両者の予言の間にはとてつもない違いが生じるでしょう。なぜならば、9次多項式では x^9 の項が支配的になりますが、線形モデルは、なんというか、線形のままですから。

一つの立場は、どうしても複雑なモデルを採用しなければならない特段の事情がない限り、科学においてはより単純な説明を採用するのが望ましいという考え方です。多くのデータ点を説明できる簡単なモデルが見つかった時には、「エウレカ！」と叫びたい 衝動に駆られるでしょう。なんといっても、単純な説明とデータが単なる偶然の一致を見せるとは、考えにくいように思われます。むしろ、そのモデルは現象の背後にある何らかの真理を言い当てている と信じたくなるでしょう。今考えている例では、

$y = 2x + (ノイズ)$ というモデルは $y = a_0 x^9 + a_1 x^8 + \ldots$ よりずっとシンプルに見えます。そんな単純さが偶然に現れたのだとすると驚きです。だからこそ、私たちは $y = 2x + (ノイズ)$ が背後にある何らかの真理を表していると期待するのです。この立場に基づくと、9次多項式モデルは局所的なノイズの効果を学習しているに過ぎません。だから、9次多項式モデルは与えられたデータ点を完璧に説明はするけれども、他のデータ点に対して汎化はできないでしょう。そして、ノイズ付き線形モデルが より強力な予言能力を持つでしょう。

この立場がニューラルネットワークにおいて何を意味するのか見てみましょう。正規化されたニューラルネットワークで期待されるように、ニューラルネットワークの大部分では小さな重みを持つと仮定しましょう。重みが小さいということは、ここそこでランダムな入力を変化させても ニューラルネットワークの振る舞いが大きくは変わらないことを意味します。そのため、正規化されたニューラルネットワークでは、データに含まれる局所的なノイズの効果を学習しづらくなっています。その代わり、正規化されたニューラルネットワークは訓練データの中で繰り返し見られる データの特徴に反応するのです。対照的に、大きな重みを持つニューラルネットワークは、入力の小さな変化に敏感に反応してその振る舞いを大きく変えてしまいます。そのため、正規化されていないニューラルネットワークは、大きな重みを使って、訓練データのノイズに関する情報をたくさん含んだ複雑なモデルを学習してしまうのです。要するに、正規化されたニューラルネットワークは訓練データに頻繁に現れるパターンに基づいた 比較的シンプルなモデルを構築します。そして、訓練データが持つノイズの特異性を学ぶことに対して 耐性を持つのです。このため、ニューラルネットワークがノイズではなく現象そのものに対する真の学習をして、それをより良く汎化できるのではないかと、希望が持てます。

ここまで説明したところで、よりシンプルなモデルが好ましいという考え方に不安を感じるかもしれません。この考えはしばしば「オッカムの剃刀」と呼ばれ、何か一般的な科学の原理であるかのように扱われます。しかし、もちろん、これは一般的な科学の原理ではありません。複雑なモデルよりも単純なモデルを好むべき、ア・プリオリな論理的理由は無いのです。実際、より複雑な説明が正しいという場合もあるのです。

結果的には複雑な説明の方が正しかったという例を2つ紹介しましょう。1940年台に物理学者のマルツェル・シャインは新粒子を発見したと主張しました。彼が働いていたゼネラル・エレクトリック社はすっかり盛り上がって、その発見を広く宣伝しました。しかし、物理学者のハンス・ベーテは懐疑的でした。ベーテはシャインの元を訪れ、シャインの新粒子の飛跡が残るプレートを調べました。シャインは次々にプレートを見せました

が、ベーテはそれぞれのプレートにデータとして採用すべきでない理由を見出しました。最後に、シャインは問題なさそうに見えるプレートを差し出しました。ベーテは、それが統計的なまぐれ当たりに過ぎないのではないかと言いました。シャイン:「そうかもしれないが、その確率はあなたの公式によると5分の1しかありません。」ベーテ:「しかし、もう既に5つのプレートを見ました。」そこでシャインは次のように言いました:「しかし、あなたは一つ一つのプレート、一つ一つの写真に対して異なる理論で批判をしました。ところが、それらが新粒子であるというたった一つの仮説は、全てのプレートを説明します。」ベーテは次のように答えました:「あなたと私のたった一つの違いは、あなたの説明は間違っているが、私の説明は全てが正しいということです。あなたのただ一つの説明は間違いですが、私の複数の説明は正しいのです。」後の研究により、ベーテが正しかったことが確かめられました*。

*この話は、物理学者のリチャード・ファインマンが歴史家のチャールズ・ワイナーとの インタビュー の中で紹介したものです。

2つ目の例として水星の軌道にまつわる話を紹介します。天文学者のユルバン・ルヴェリエは1859年に水星の軌道がニュートンの重力理論による 予測と一致しないことを発見しました。そのズレは本当に、本当に僅かなもので、当時よく見られた説明は、つまるところ、ニュートンの重力理論はだいたい正しく、しかし僅かな修正が必要であるというものでした。1916年に、アインシュタインはそのズレが彼の一般相対性理論を使うととてもよく 説明できることを示しました。ところが、一般相対性理論はニュートンの重力理論と 根本的に異なっており、ずっと複雑な数学に基づいていました。このようにとても複雑な一般相対性理論ですが、今日ではアインシュタインの説明が 広く受け入れられています。そして、ニュートンの理論は修正されたものも含めて間違っていると 考えられています。これは一つには、アインシュタインの理論がニュートンの理論では 説明が困難な他の多くの現象を説明できるためです。そしてさらに、ニュートンの重力では全くもって予言できない現象を正確に予言できるのです。しかし、これらの印象的な性質は、初期の頃から明らかであったわけではありません。もし単純さのみに基づいて判断していたら、おそらくいずれかの修正されたニュートン理論が もっと魅力的に映ったことでしょう。

これらの逸話から3つの教訓が得られます。第1に、2つのモデルのどちらが真に「より単純」かを決めるのは、非常に微妙な問題であること。第2に、もしそのような判断を下すことができたとしても、単純さが正しさの指標になりうるかどうかは注意深く検討する必要があること。第3に、モデルを測る本当の尺度は単純さではなく、未知の領域で新しい現象をどれほど良く予言できるかという点であることです。

これらの注意を頭の片隅においておいて、経験的な事実として、正規化したニューラルネットワークは通常、正規化されていないニューラルネット

ワークよりも よく汎化するのだと言っておきます。なので、この本の残りの部分では頻繁に 正規化を利用します。私が上でいくつかの逸話を紹介したのは、正規化によってニューラルネットワークの汎化が助けられることについて、完全に納得できる理論的な説明が存在しないことを伝えるためなのです。実は、研究者は新しい正規化の手法を試しては、それらを比較してどちらがより良いか調べ、その理由を理解しようと試みています。ですから、正規化は今の段階ではある種の その場しのぎであるとみなすこともできます。正規化は仕事の助けになりますが、正規化によって実際に何が起こっているのかについて、完全に満足の行く系統的な理解は無く、不完全なヒューリスティックと経験則しかないのです。

ここには科学の核心に関わる、さらに深い論点があります。それは、私たちが経験的な理解をどう汎化するのかという問いです。正規化は上手く計算するための魔法のようなもので、それは確かにニューラルネットワークが汎化するのを助けますが、どのように汎化しているのかという原理に関する理解も、何が最良のアプローチなのかという答えも教えてはくれないのです*。

このことが一際悩ましく思えるのは、私たち人間が日常的に驚くほどうまく汎化するからです。象の画像をほんの数枚見ただけで、子供はすぐに他の象を認識し始めるでしょう。もちろん、時には間違えもするでしょうし、サイを象と混同するかもしれません。それでも多くの場合に非常に正確でしょう。つまり、ここに一つのシステム、すなわち人間の脳があって、それは莫大な数のパラメータを持ちます。そしてそれは、たった数枚の訓練画像を見せられるだけで、このシステムは他の画像に対して汎化しはじめるのです。私たちの脳はある意味で、非常に上手く正規化されているのです！ どうすれば私たちにもニューラルネットワークで同じことができるでしょうか？ 現時点では分かりません。向こう数年のうちに、人工的なニューラルネットワークを 正規化するより強力な手法が開発されて、究極的にはわずかな訓練データから良く汎化できるようになると期待します。

実は、これまで実装してきたニューラルネットワークは、既に**ア・プリオリ**に期待されるよりも良く汎化しています。100個の隠れニューロンを持つニューラルネットワークには、80,000個近いパラメータがあります。訓練データには50,000枚の画像しかありません。いわば、80,000次多項式で50,000個のデータ点を フィットしようとしているようなものです。本来なら、このニューラルネットワークはひどく 過適合を起こしても当然なわけです。にも関わらず、これまで見てきたように非常に良く汎化するのです。なぜこんなことが起こるのでしょうか？これはよく理解されているわけ

*これらの論点は、スコットランド人の哲学者、デイビッド・ヒュームによって "An Enquiry Concerning Human Understanding" (1748) の中で議論された、帰納法の問題に 遡ります。帰納法の問題は、現代の機械学習においては、デイビッド・ウォルパートとウィリアム・マクレディーによるノーフリーランチ定理として現れます。(リンク)

ではありません。「多層ニューラルネットワークにおける勾配降下学習のダイナミクスには『自己正規化』効果がある」という仮説はあります*。

*出典は Gradient-Based Learning Applied to Document Recognition, by Yann LeCun, Léon Bottou, Yoshua Bengio, and Patrick Haffner (1998) です。

この節の終わりに、ここまで説明せずに来た詳細に戻ってきましょう: なぜL2正規化でバイアスを制限し**ない**のでしょうか？ もちろん、バイアスを正規化するように正規化の手順を修正するのは簡単です。経験的には、そうしても結果が大きく変わらないので、バイアスを正規化するかどうかは単なる慣例であるということも出来ます。しかし、次の事実には言及しておく価値があるでしょう。それは、バイアスが大きくなっても、重みが大きくなった時のように 入力に対するニューロンの感受性が高まるわけではない、という事実です。だから、大きなバイアスのために訓練データのノイズを学習してしまうのではないかと 心配する必要は無いわけです。同時に、大きなバイアスを許すことで、ニューラルネットワークはより柔軟に振る舞えるようになります。特に、大きなバイアスを許すことでニューロンの出力が容易に飽和できるようになります。時にこの性質が望ましい場合があります。これらの理由から、普通はバイアス項を正規化に含めません。

その他の正規化手法

正規化の手法はL2正規化以外にもたくさんあります。実は、余りにたくさんありすぎて、その全てを要約することはできそうもありません。この節では、3つの手法、L1正規化、ドロップアウト、そして人工的な学習データの伸張、に的を絞って手短に紹介します。ここではL2正規化についてしてきたような詳細な説明はしません。その代わり、ここでの目的は主要なアイデアに親しんでもらうことと、正規化手法の多様性を認識してもらうことです。

L1正規化: このアプローチでは、正規化する前のコスト関数に重みの絶対値の和を足します:

$$C = C_0 + \frac{\lambda}{n} \sum_w |w|. \tag{88}$$

直感的には、大きな重みを不利にし、ニューラルネットワークが小さな重みを好むように仕向けるという意味で、L2正規化と似ています。もちろん、L1正規化項はL2正規化項と同じではないので、両者が全く同じ振る舞いをするわけでもありません。L1正規化して訓練したニューラルネットワークの振る舞いがL2正規化の場合とどう違うのか見てみましょう。

そのために、まずはコスト関数の偏微分を調べましょう。式(88) を微分して、次の表式が得られます:

$$\frac{\partial C}{\partial w} = \frac{\partial C_0}{\partial w} + \frac{\lambda}{n}\,\mathrm{sgn}(w). \tag{89}$$

ここで、$\mathrm{sgn}(w)$ は w の符号、つまり、w が正なら $+1$ で負なら -1 です。この表式を使うと、L1正規化を使った確率的勾配降下法を実行するように逆伝播法を修正できます。そうやって求めたL1正規化の下での更新規則は、

$$w \rightarrow w' = w - \frac{\eta\lambda}{n}\mathrm{sgn}(w) - \eta\frac{\partial C_0}{\partial w} \tag{90}$$

です。ここでいつも通り、$\partial C_0/\partial w$ をミニバッチ平均によって評価することができます。これをL2正規化を用いた更新規則と比べてみましょう: (c.f. 式 (86)),

$$w \rightarrow w' = w\left(1 - \frac{\eta\lambda}{n}\right) - \eta\frac{\partial C_0}{\partial w}. \tag{91}$$

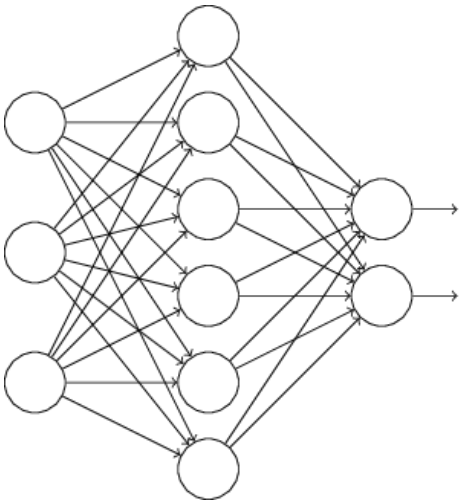
どちらの表式でも、正規化の効果は重みを縮小することだとわかります。この点は、どちらの正規化も大きな重みを不利にするという直感と整合します。しかし、縮小の仕方が異なります。L1正規化は、重みを0の方向に一定の大きさだけ縮小します。L2正規化では、重みは w に比例する量だけ縮小します。そのため、特定の重みが大きな絶対値 $|w|$ を取っている時に、L1正規化はL2正規化と比べるとほんのわずかにしか w の値を変化させません。反対に、ある重みが小さな絶対値 $|w|$ を持っている時には、L2正規化よりもずっと大きく w の値を変化させます。全体としては、L1正規化をした後のニューラルネットワークでは、一部の比較的少数の重要なリンクに重みに集中し、他の重みは0の方に追いやられることになります。

ここまでの議論で、偏微分 $\partial C/\partial w$ が $w = 0$ で定義されないことについて 敢えて触れずに来ました。関数 $|w|$ は $w = 0$ に尖った「角」があるので、この点で微分不可能なのです。とはいえ、実はこのことは問題ありません。もし $w = 0$ なら、その点で通常の（正規化していない）規則を使って確率的勾配降下法を行えば良いのです。この方法なら問題無いでしょう。というのも、直感的には正規化の効果が重みを縮小することなら、$w = 0$ では明らかにこれ以上縮小のしようが無いからです。より正確には、式 (89) と (90) の中で、$\mathrm{sgn}(0) = 0$ という定義を採用します。そうすることで、L1正規化して確率的勾配降下法を実行するコンパクトな規則が出来上がります。

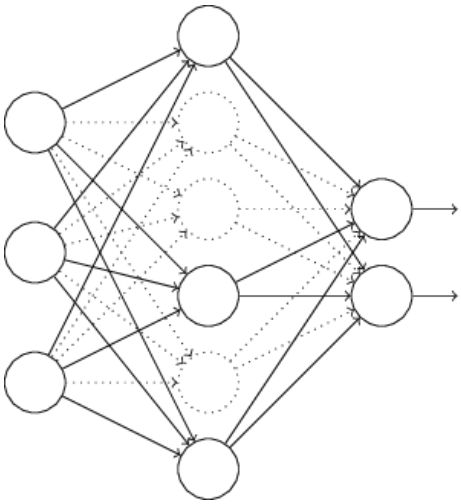
ドロップアウト: ドロップアウトは、これまで紹介してきたものとは全く違う正規化の手法です。L1正規化やL2正規化と違って、ドロップアウトでは元のコスト関数をそのまま使います。その代わり、ドロップアウトではニューラルネットワークそのものを修正します。まずはドロップアウトの基本的な

仕組みを説明してから、なぜドロップアウトで上手くいくのか、そしてどのような結果をもたらすのかを説明します。

あるニューラルネットワークを訓練しようとしていると仮定しましょう：



特に、訓練入力 x とそれに対応する正解の出力 y を知っているとします。通常、ニューラルネットワークを訓練するためにまず x を順伝播させて、次に勾配を求めるために逆伝播を行います。ドロップアウトではこのプロセスを修正します。まず、ランダムに（そして一時的に）ニューラルネットワークの隠れニューロンを半分削除します。このとき入力・出力ニューロンは削除せずにそのままにしておきます。すると、下に示したようなニューラルネットワークが得られます。ここで、抜け落ちた（ドロップアウトした）ニューロン、つまり一時的に削除されたニューロンは、点線で残してあることに注意してください：



その後、入力 x を修正されたニューラルネットワークで順伝播させ、次にその結果を再び修正後のニューラルネットワークで逆伝搬させます。ミニバッチに含まれる全ての例についてこれを実行した後、削除されずに残っている重みとバイアスを更新します。そしてこの過程を繰り返します。つまり、まずドロップアウトしたニューロンを戻し、削除するニューロンを再

びランダムに選び直し、新しいミニバッチを用いて勾配を評価し、ニューラルネットワークの重みとバイアスを更新します。

これを繰り返すことで、ニューラルネットワークは重みとバイアスを学習します。もちろん、ここで得られた重みやバイアスは、隠れニューロンの半分がドロップアウトしているという条件の元で学習した結果です。実際にニューラルネットワークの全体を動かすと、2倍のニューロンが有効になります。この点を補うために、隠れニューロンから出て行く重みを半分にします。

このドロップアウトの手続きは奇妙でアドホックに思えるかもしれません。なぜこれが正規化を助けるのでしょうか？何が起こっているのか説明するために、一旦ドロップアウトについて考えるのをやめて、ドロップアウト無しに 通常の方法でニューラルネットワークを訓練することを想像してみてください。特に、複数のニューラルネットワークを同じ訓練データで訓練する 状況を考えてみましょう。もちろん、それぞれのニューラルネットワークは 全く同じものではありませんから、結果として訓練した後のニューラルネットワークから得られる出力も ニューラルネットワークごと異なるかもしれません。そうなった時には、何らかの平均化や多数決のような手法を用いることで どの出力を採用するか決定できるでしょう。例えば、5つのニューラルネットワークを 訓練して、そのうち3つが数字の画像を "3" に分類したとすると、その時は おそらく実際に "3" なのでしょう。残り2つのニューラルネットワークは 単に間違えてしまっただけだと思われます。この種の平均化の手法は、しばしば過適合を軽減する強力な（しかし負担も大きい）手法です。異なるニューラルネットワークは異なる過適合のしかたをすることがあるので、平均化がこの種の過適合を防ぐ助けになるのです。

これとドロップアウトがどう関係するというのでしょうか？直感的には、異なるニューロンの組み合わせをドロップアウトしてから訓練するのは、異なるニューラルネットワークを訓練しているようなものです。だから、ドロップアウトの手続きは、大量の異なるニューラルネットワークの結果を 平均化しているようなものであると見なせます。異なるニューラルネットワークは 違った過適合のしかたをするでしょうから、上手くいけば全体としては過適合が軽減されるでしょう。

これと関連するヒューリスティックな説明は、この手法を用いた最初期の論文で与えられています：「この手法はニューロン間の複雑な相互適合を軽減します。というのも、訓練に際してニューロンは特定の他のニューロンを頼りにすることができないからです。したがって、あるニューロンは、ランダムに選ばれた他のニューロンと一緒にされても役に立つよう

*ImageNet Classification with Deep Convolutional Neural Networks, by Alex Krizhevsky, Ilya Sutskever, and Geoffrey Hinton (2012).

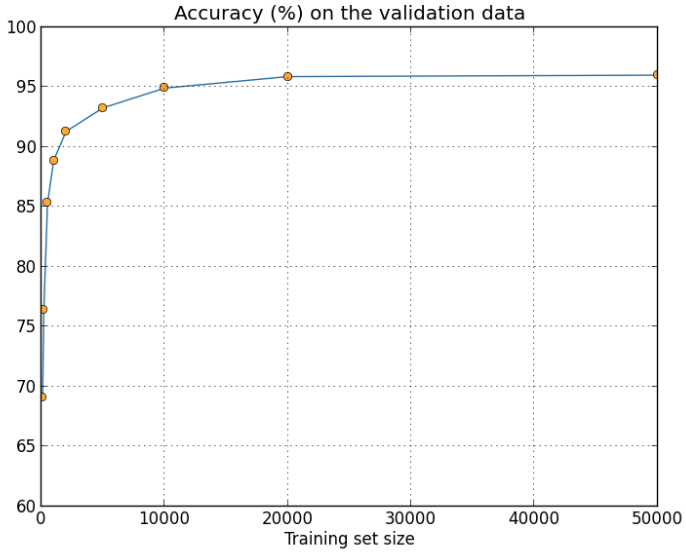
な、データが持つより強固な特徴を学ぶように強制されるのです。」ニューラルネットワークのことを何らかの予言をするモデルだと見なすなら、ドロップアウトは入力される情報の欠落に対して強いモデルを作る方法だと言うことができるでしょう。この点において、何となくL1正規化やL2正規化と似ているとも言えます。これらの手法は、重みを小さくすることで、ニューロン間の接続を失うことに対して ニューラルネットワークを強くしていると言えるからです。

もちろん、ドロップアウトの実力を測る真の尺度は、それがニューラルネットワークの パフォーマンス改善に大きな成功を収めてきたという点です。この手法を導入した原論文* では、様々な異なるタスクにこの手法を適用しています。私たちにとっては、彼らがドロップアウトをMNISTの分類問題に適用していることが 特に興味深いところです。彼らはシンプルなフィードフォワードニューラルネットワークを使って、私たちがやってきたのと同じようなことをしています。論文には、その時点で達成されていた試験データに対する最高の分類精度が 98.4 パーセントであると 書いてあります。彼らはドロップアウトと修正されたL2正規化を組み合わせて、それを 98.7 パーセントに改善しました。同様に印象的な結果が画像や音声認識、自然言語処理 などたくさんの他のタスクに関しても得られています。ドロップアウトは過適合の問題が深刻になる巨大で多層のニューラルネットワークを訓練するときに、特に有用になります。

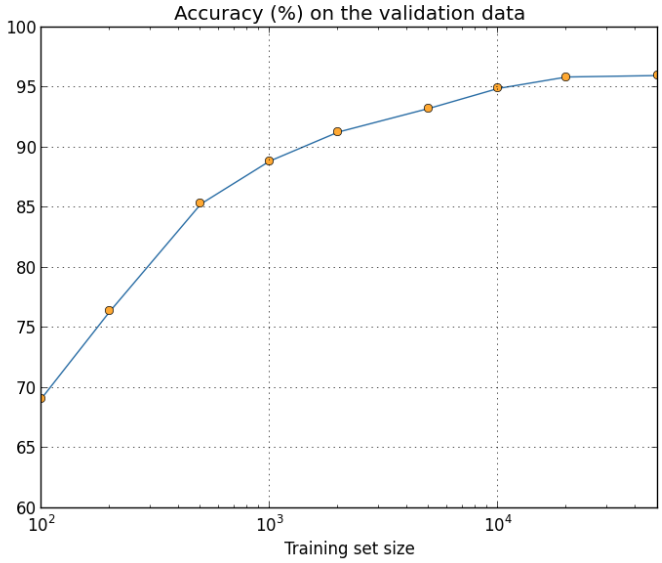
*Improving neural networks by preventing co-adaptation of feature detectors by Geoffrey Hinton, Nitish Srivastava, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov (2012). この論文では、このドロップアウトに関する短い紹介文ではあえて触れずに来た 多くの繊細な点についても議論しています。

人工的な学習データの伸張: 前に確かめたように、訓練画像が1,000枚しか無いと MNISTの分類精度は80パーセント中盤にまで減少してしまいます。このこと自 体は驚くべきことではありません。 訓練データが少ない分、ニューラルネットワークが学ぶ手書き文字のバリエーションも少なくなるからです。 では、30個の隠れニューロンを持つニューラルネットワークを異なるサイズの訓練データセットで 訓練したら、分類精度はどのように変化するでしょうか。ここでは、ミニバッチの大きさを10、学習率を $\eta = 0.5$、正規化パラメータを $\lambda = 5.0$ とし、コスト関数にはクロスエントロピーを 使うことにします。訓練データセットの全体を使うときには訓練を30エポック回します。 訓練データを小さくするのに反比例するように、エポック数を大きくすることにします。 重み減衰因子が訓練データセットの大きさに依らず一定になるように、全訓練データを用いるときには $\lambda = 5.0$ とし、小さい訓練データセットを使うときには そのサイズに比例して λ を小さくすることにします*。

*これと次のグラフはプログラム more_data.py が生成したものです。



見て分かるように、より多くの訓練データを使うことで分類精度はかなり改善されます。おそらく、訓練データをさらに増やすことでこの改善はまだまだ続くでしょう。上のグラフを見るとほとんど飽和しているようにも見えますが、横軸を訓練データサイズの対数にしてプロットし直すと、次のグラフが得られます:



こうしてみると、終わりに向けてまだグラフが上昇しているのが明らかになります。もしもっと莫大な数の訓練データを使えたら、例えば、100万も10億も手書きの数字を集められたら、このとても小さなニューラルネットワークでもかなり良いパフォーマンスを発揮するでしょう。

訓練データをもっと集めようというのは素晴らしいアイデアです。ただ残念なことに、それはとても負担が大きく、現実的にはいつでも実行できることではありません。しかし、訓練データを集めるのと同じくらい効果的な別の方法があります。例えば、MNISTの訓練画像から「5」の画像を一枚取り出してみましょう。

そして、この画像を少しだけ、例えば15度回転させてみます：

私たちが見れば、まだこの画像が同じ数字を表していると分かります。それでも、ピクセル単位で見ると、MNISTに含まれるどの画像ともかなり異なります。この画像を訓練データに加えれば、これからニューラルネットワークは数字の分類についてより多く学ぶだろうことは想像に難くありません。さらに、訓練データに加えて良いのはこの1枚だけではありません。**全てのMNIST訓練画像について*たくさんの*小さな回転を施して、元の訓練データを拡張することができます。そしてこの拡張された訓練データを使ってニューラルネットワークのパフォーマンスを改善できるのです。

このアイデアは非常に強力で広く使われています。ある論文*から幾つかの結果を紹介しましょう。この論文では、訓練データを拡張する方法の幾つかの変種をMNISTに適用しています。彼らが考えたニューラルネットワークの構造は、私たちが使ってきたものと似た フィードフォワードニューラルネットワークで、800個の隠れニューロンを持ち コスト関数にはクロスエントロピーを使っています。通常のMNIST訓練データでニューラルネットワークを訓練した時には、彼らは試験データに対して98.4%の分類精度を達成しました。しかし、次に訓練データを 上で説明したような回転に加えて、平行移動と歪みで拡張しました。拡張された訓練データでニューラルネットワークを訓練すると、98.9% の分類精度を達成しました。彼らはさらに、彼らが「弾性歪み」と呼んだ変形を使って実験しました。これは 手の筋肉で起こるランダムな振動を模倣する、特別な変形です。訓練データの拡張に弾性歪みを使うことで、さらに改善して99.3%という分類精度を達成しました。彼らが行ったことは、実際の手書き文字に見られるバラつきに触れさせることで ニューラルネットワークに実質的により多くの手書き文字を経験させていたと言えます。

*Best Practices for Convolutional Neural Networks Applied to Visual Document Analysis, by Patrice Simard, Dave Steinkraus, and John Platt (2003).

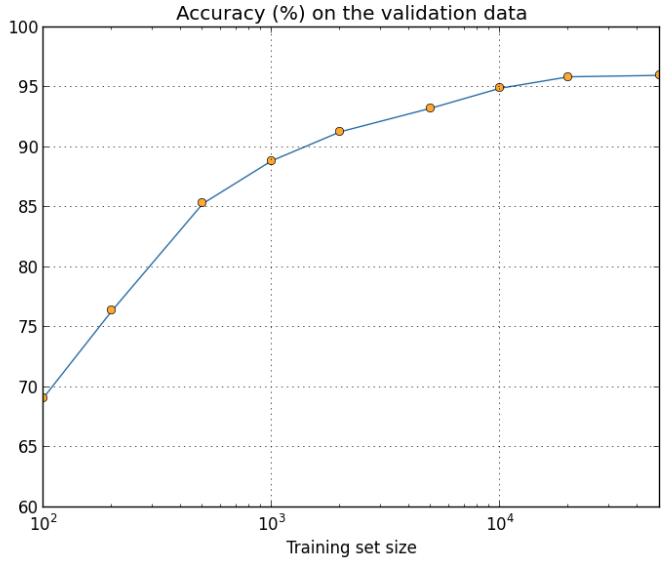
このアイデアの仲間は、手書き文字認識に限らず多くの機械学習のタスクで パフォーマンスを改善するために使うことができます。基本的な原理は、現実的なデータに見られる バラつきを反映するような操作を訓練データに行い、訓練データを拡張するということです。これをどう実行するか考えるのは難しいことではありません。例えば、音声認識を行うニュー

ラルネットワークを作っているとしましょう。私たち人間は、背後に雑音があっても会話を認識できます。ですから、雑音を加えてデータを拡張できます。また、人間は会話の速度が早くても遅くても、会話を認識できます。したがってこれも訓練データを拡張する一つの方法です。これらの手法はいつも使われるわけではありません。例えば、訓練データにノイズを加えて拡張する代わりに、まずノイズ軽減フィルターを通して綺麗にした データをニューラルネットワークに入力することにした方が効率的だとしても不思議ではありません。それでも、訓練データを拡張するという選択肢を頭の片隅においておき、それを実行する機会を伺うのは、良い心がけです。

演習

- 上で議論したように、訓練画像を少し回転させるのはMNISTの訓練データを拡張する一つの方法です。この時、任意の回転角を許すとどのような問題が起こるでしょうか？

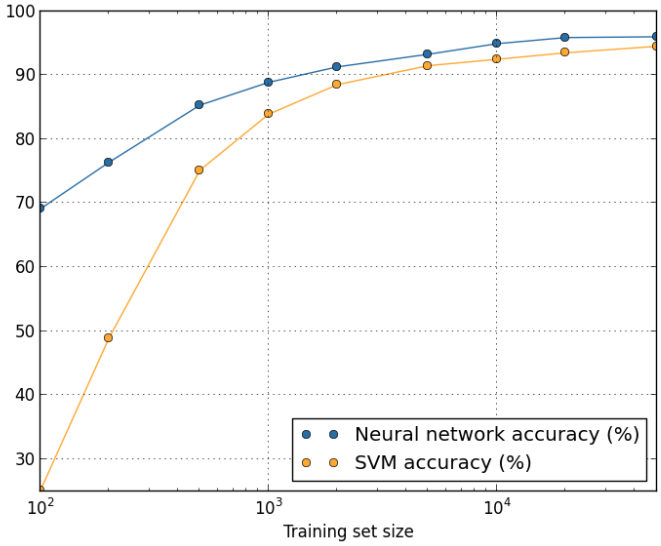
ビッグデータと分類精度比較に関する余談: ニューラルネットワークの分類精度が訓練データセットの大きさによってどう変化するのか もう一度見てみましょう:



ここで、ニューラルネットワークの代わりに、何か他の機械学習手法で数字を分類することを 考えてみましょう。例えば、Chapter 1で手短に紹介した サポートベクターマシン（SVM）を使ってみましょう。SVMに馴染みがなくても心配は不要です。ここでの議論にはその詳細を理解している必要はありません。SVMを自分で実装する代わりに、scikit-learn ライブラリで提供されている SVMをここでは利用します。SVMのパフォーマンスが訓練データセットの大きさに関して どう変化するのかを示したのが下の

図です。比較のために、ニューラルネットワークの結果も 合わせてプロットしました*:

*このグラフは、(これまでのいくつかのグラフと同様に)プログラム more_data.py で生成したものです。



たぶんこのグラフを見てまず皆さんが驚くのは、ニューラルネットワークがどんな大きさの訓練データセットについてもSVMを大きく上回る結果を上げていることでしょう。これは素晴らしいことではありますが、過大評価してもいけません。というのも、私たちはニューラルネットワークのパフォーマンス改善のために様々な チューニングを施してきましたが、ここで使ったSVMについてはscikit-learnで提供されている 設定をそのまま使ったものだからです。このグラフに関するもっと微妙で、しかし興味深い事実は、50,000枚の訓練画像を使って訓練したSVMが94.48パーセントの分類精度を達成しており、これは5,000枚の訓練画像で訓練されたニューラルネットワークの分類精度93.24パーセントよりも 高いパフォーマンスを発揮しているという点です。言い換えるなら、時として、より多くの訓練データを使うことで、使用している機械学習アルゴリズムの差は 埋められるということです。

時にはより一層面白いことが起こります。ある問題をアルゴリズムAとアルゴリズムBという 2つの機械学習アルゴリズムを使って解こうとしているとしましょう。時々 、ある訓練データセットではアルゴリズムAが勝ち、異なる訓練データセットではアルゴリズムBが 勝つ、ということが起こります。上のグラフではそのようなことは起きていません。もしそうなら、2つのグラフが交差するはずです。しかし実際に、そのようなことは起こりうるのです*。したがって、「アルゴリズムAはアルゴリズムBよりも優れていますか？」と質問された時、正しい返事のし方は「使っているデータセットは何ですか？」になります。 データセットが指定されない限り、パフォーマンスの比較はできないのです。

*次の論文の中でその顕著な例が与えられています: Scaling to very very large corpora for natural language disambiguation, by Michele Banko and Eric Brill (2001).

これらの注意は、開発を行う時にも研究論文を読む時にも心に留めておくべきです。多くの論文では、標準的なベンチマークとなるデータセットに対するパフォーマンスを改善する 新しい手法やコツを探すことに注力しています。研究論文ではしばしば、「我々の素晴らしい手法は、標準的なベンチマークX に対してYパーセントの改善をもたらしました」という形式を取る主張を目にします。このような主張はしばしばそれ自体が本当に興味深いものですが、使用した特定の訓練データセットにおいてのみ適用できるものだと理解しなければなりません。もしこの世界とは別に、ベンチマークのデータセットを作成した人がより多くの研究資金を獲得できる 世界があったら何が起こっているでしょうか？研究者たちはより多くの訓練データを集めるために より多くのお金を掛けるかもしれません。すると、「素晴らしい手法」がもたらした「改善」が、より大きな訓練データセットでは消滅してしまう、なんてことも、全くもってありうることなのです。言い換えると、元々 主張した「改善」は、歴史の偶然に過ぎないのかもしれません。ここから学ぶべき教訓は、特に実際の応用においては、より良いアルゴリズム**と**より良い 訓練データがどちらも必要なのです。より良いアルゴリズムを追求するのは構いませんが、より多くの訓練データを集めることでより簡単に目標を達成できる可能性を見落として、アルゴリズム探しに注力する愚は避けなければなりません。

問題

- **(研究課題)** 機械学習アルゴリズムは訓練データセットを 大きくする極限でどう振る舞うのでしょうか？どのようなアルゴリズムに対しても、漸近的パフォーマンスの概念を大きな訓練データの極限で定義してみるのは自然なことです。この問題に対する手っ取り早い方針としては、上に示したようなグラフを適当な近似曲線で フィッティングして、それを無限大まで外挿してみることです。このようなアプローチに 対しては、近似曲線の選び方によって異なる漸近的パフォーマンスにたどり着いてしまう という批判があるでしょう。特定の種類の近似曲線を使うことに対して、原理的な 正当化は可能ですか？もしそうなら、いくつかの異なる機械学習アルゴリズムに対して、漸近的パフォーマンスの比較を行ってください。

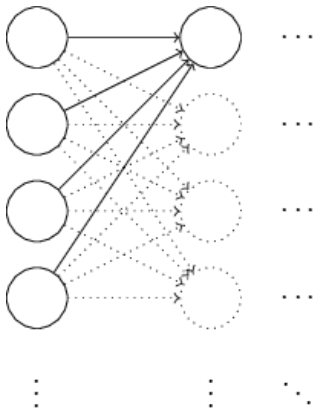
まとめ： ここまで過適合と正規化について集中的に考察してきました。これについては一旦ここでお終いにして次のトピックに移りますが、これらの問題については再び戻ってくることになります。何度も述べたように、コンピュータの性能が向上し、より大きなニューラルネットワークの 訓練が可能になるにつれ、ニューラルネットワークにおける過適合の問題も重要性を増しています。結果として、過適合を軽減するより強力な正規化手

法の開発が急務となっており、現在も極めて活発な研究分野をなしています。

重みの初期化

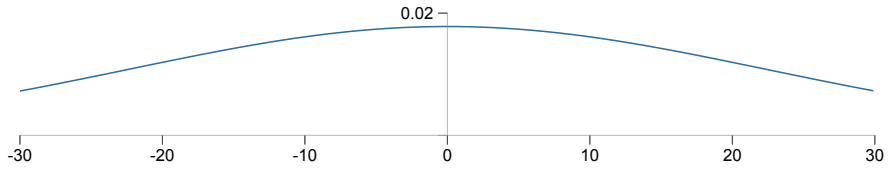
ニューラルネットワークを学習させる前に、重みとバイアスの初期値を選ぶ必要があります。これまでの例では、Chapter 1で 手短に議論した処方に従って選んできました。念のためもう一度書いておくと、その処方とは、全ての重みとバイアスを平均 0、標準偏差 1 に規格化した独立なガウス分布に従って 選ぶものでした。これまではこの方法で上手くいきましたが、そのように選ぶべき根拠も 今のところ明らかではありません。重みとバイアスの初期値をより上手く選ぶことで、ニューラルネットワークの学習を加速することはできないでしょうか？

実は、結論から言ってしまうと、規格化されたガウス分布を使うよりもずっと良い やり方があるのです。その理由を見るために、非常にたくさん、例えば $1,000$個の 入力ニューロンを持つニューラルネットワークを考えてみましょう。そして、第1の隠れ層につながる重みを全て規格化されたガウス分布で初期化するとします。しばらくは入力ニューロンと隠れ層の第1のニューロンを結ぶ重みに集中し、ニューラルネットワークの残りの部分を無視することにします：



議論をさらに単純化するために、ある特定の訓練入力 x を使って ニューラルネットワークを訓練することを考えます。その x は、半分の入力ニューロンがオン、すなわち 1 の値を取り、残り半分のニューロンがオフ、すなわち 0 の値を取るものとします。以下の議論はより一般的な状況で成り立ちますが、この特別な例から状況が大まかに理解できると思います。隠れニューロンへの入力となる重み付き総和 $z = \sum_j w_j x_j + b$ を考えましょう。 半分の x_j はゼロなので、総和のうち 500 項は消えます。したがって、z は 500 個の重みと 1 個のバイアスを合わせて、全部で 501 個の規格化されたガウス分布変数の和になります。したがって、z 自身も平均がゼロで標準偏差が $\sqrt{501} \approx 22.4$ の ガウス分布に従うランダム変数で

す。つまり、z はかなり広がったガウス分布に従っていて、その分布は全く鋭くありません:



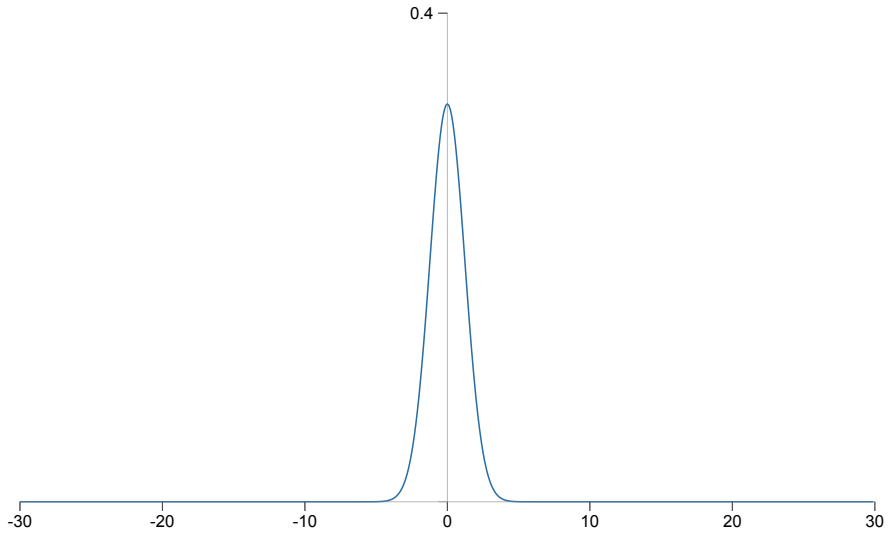
特に、$|z|$ が非常に大きく $z \gg 1$ か $z \ll -1$ が高い確率で成り立つことは、このグラフから明らかです。その場合、隠れニューロンの出力 $\sigma(z)$ は 1 か 0 に非常に近い値を取ります。つまり、今考えている隠れニューロンは 飽和してしまうのです。そしてそうなってしまうと、重みを少し変化させても 隠れニューロンの活性はほんの僅かにしか変化しません。そしてこの非常に僅かな活性の変化は、ニューラルネットワークの残りの部分に対してほとんど影響せず、最終的にコスト関数に ごく小さな変化しかもたらさないでしょう。結果として、これらの重みの学習に勾配降下法を使っても、学習は非常にゆっくりとしか進行しないでしょう*。この問題は、出力ニューロンが誤った値に飽和していると学習が非常に遅くなるという、この章の前半で議論した問題と似ています。その際は、コスト関数を上手く選ぶことで その問題を解決しました。良いコスト関数は出力ニューロンの飽和の問題を解決しましたが、残念ながら、同じ方法では隠れニューロンの飽和の問題は全く改善されません。

*この点についてはより詳細にChapter 2で議論しました。その際、逆伝播の式を 用いて、飽和したニューロンへ入力する重みの学習は遅くなることを示しました。

ここまで、第1の隠れ層に入力する重みについて議論してきました。もちろん、同様の議論は 他の隠れ層に対しても成り立ちます:隠れ層の重みを規格化されたガウス分布を使って 初期化した場合、隠れニューロンの活性はしばしば 0 か 1 に非常に近く、学習は非常にゆっくりとしか進みません。

この種の飽和と学習の減速を避けるために、重みとバイアスのもっと上手い初期化の方法は あるのでしょうか？今、n_{in} 個の入力を持つニューロンを考えます。その入力に掛かる重みを平均 0 で標準偏差 $1/\sqrt{n_{\text{in}}}$ のガウス分布で 初期化しましょう。つまり、ガウス分布を押しつぶしてニューロンが飽和しづらくなるようにします。バイアスについては後述する理由により、変わらず平均 0 で標準偏差 1 のガウス分布で初期化します。そのように選ぶことで、重み付き総和 $z = \sum_j w_j x_j + b$ は再び平均 0 のガウス分布に 従うランダム変数となりますが、以前よりもずっと鋭い分布になります。前に考えたように、500 個の入力がゼロで 500 個の入力が 1 だとしましょう。このとき、簡単な計算により(下の演習参照)、z は平均 0 で標準偏差 $\sqrt{3/2} = 1.22\ldots$ のガウス分布に従うことが分かります。この分布は前よりもずっと鋭いピークを持っています。実は、下のグラフではかなり控えめに ピークが描かれています。というのも、ピークの全体を

描くために、このグラフでは前のグラフから 縦軸のスケールを変えているからです。



このようなニューロンはずっと飽和しづらく、学習の減速はずっと起こりづらいでしょう。

Exercise

- 上の段落に出てきた $z = \sum_j w_j x_j + b$ の標準偏差が $\sqrt{3/2}$ であることを 示してください。ヒント:(a) 独立なランダム変数の和の分散は、個々のランダム変数の分散の 和に等しくなります。(b) 分散は標準偏差の2乗です。

上述のように、バイアスについては以前と同様、平均が 0 で標準偏差が 1 のガウス分布で 初期化します。こうしてもニューロンの過剰に飽和させることは無いので大きな問題になりません。実は、飽和の問題さえ避ければ、バイアスをどのように初期化しても大勢に影響はないのです。全てのバイアスを 0 に初期化して、あとは勾配降下法で学習するに任せる人さえ居ます。どのように初期化しても大して変わりないので、私たちは以前と同じ方法でバイアスを初期化することにします。

では、MNISTの手書き文字分類のタスクを使って、重みの初期化のやり方による結果の違いを見てみましょう。以前と同様に、30 個の隠れニューロン、大きさ 10 のミニバッチ、正規化パラメータ $\lambda = 5.0$ と、クロスエントロピーコスト関数を用いることにします。結果をグラフ上で少し見やすくするため、学習率は $\eta = 0.5$ から 0.1 に少し減らすことにします。まず、以前のやり方で重みを初期化してからニューラルネットワークを訓練してみましょう:

```
>>> import mnist_loader
>>> training_data, validation_data, test_data = ¥
... mnist_loader.load_data_wrapper()
```

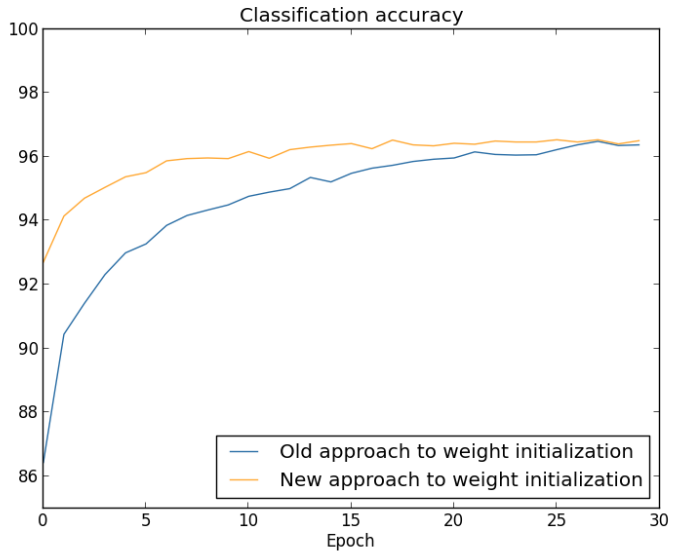
```
>>> import network2
>>> net = network2.Network([784, 30, 10], cost=network2.CrossEntropyCost)
>>> net.large_weight_initializer()
>>> net.SGD(training_data, 30, 10, 0.1, lmbda = 5.0,
... evaluation_data=validation_data,
... monitor_evaluation_accuracy=True)
```

次に、新しい方法で重みを初期化してみましょう。実はこちらのほうが簡単です。というのも、network2 のデフォルトでは、新しい方法で重みを初期化するように なっているからです。したがって、上の net.large_weight_initializer() を 省略すればよいです:

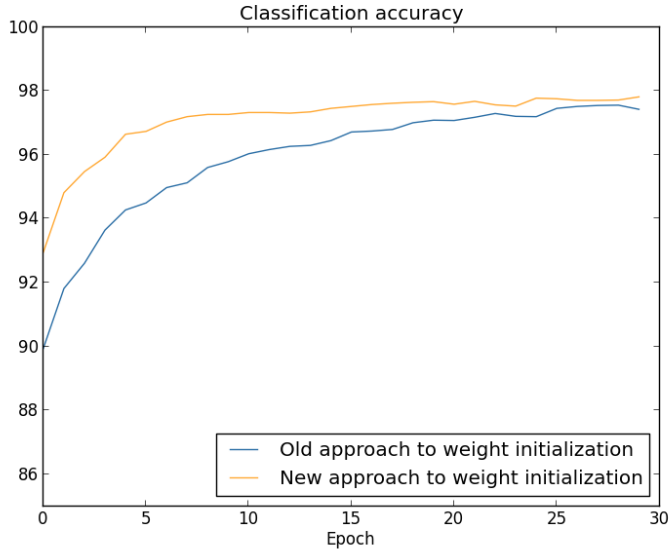
```
>>> net = network2.Network([784, 30, 10], cost=network2.CrossEntropyCost)
>>> net.SGD(training_data, 30, 10, 0.1, lmbda = 5.0,
... evaluation_data=validation_data,
... monitor_evaluation_accuracy=True)
```

結果をプロットすると*、次のグラフが得られます:

*これと次のグラフを作成するために使ったプログラムは、weight_initialization.py です。



いずれの場合も、最終的な分類精度は96パーセント強といったところです。30エポックで両者はほぼ同じ分類精度に到達します。しかし、新しい方法では、ずっとずっと早くその分類精度に到達します。最初のエポックが終わったところで、古いやり方では分類精度が87パーセント弱ですが、新しい方法では93パーセントに 達しようとしています。重みの初期化方法を変えることで、古いやり方よりずっと良い領域から出発できて、結果の改善もかなり素早く進行するのです。同じ現象は 100 個の隠れニューロンを使っても起こります:



この場合には、2つの曲線が一致するには至っていません。しかし、私が自分で 実験してみた結果、あと2, 3エポックだけ訓練を続けると、分類精度はほとんど 同じになります。したがって、これらの実験の結果を見ると、重みの初期化方法を改善しても、学習がスピードアップするだけで、最終的なニューラルネットワークのパフォーマンスを 改善することは無いかのように思えます。しかし、Chapter 4では、重みの初期分布の標準偏差を $1/\sqrt{n_{\rm in}}$ にすることで、ニューラルネットワークの長時間における振る舞いが顕著に改善される例を扱います。重みの初期化方法は学習速度のみならず、時にニューラルネットワークのパフォーマンスをも左右することになるのです。

重みの初期分布の標準偏差を $1/\sqrt{n_{\rm in}}$ にすることで、ニューラルネットワークの学習が改善されます。重みを初期化する他の手法も 提案されていますが、多くは同じ基本的なアイデアに基づいています。私たちの目的のためにはこれまでに紹介した方法で十分なので、ここでは他の手法は取り上げません。関心がある読者は、Yoshua Bengioによる2012年の論文*の14, 15ページにある議論と、そこで引用されている文献を見てみることを勧めます。

*Practical Recommendations for Gradient-Based Training of Deep Architectures, by Yoshua Bengio (2012).

問題

- **正規化と改善された重み初期化法の関係** L2正規化することで、時々自動的に新しい方法で重みを初期化することと似た 効果が得られることがあります。重みの初期化に古い方法を使ったとしましょう。ヒューリスティックな議論で次のことを示してください: (1) λ が小さすぎなければ、訓練のはじめの数エポックでは重み減衰が支配的である。(2) $\eta\lambda \ll n$ とすると、重みは $\exp(-\eta\lambda/m)$ の因子で減衰する。(3) λ が大きすぎないとすると、重みの多さが $1/\sqrt{n}$ になった

辺りで、重み減衰が落ち着いてくる。ここで、n はニューラルネットワークの全ての重みの数とします。この節でグラフに示した例では、これら全ての条件が満たされていることを議論してください。

手書き文字認識再訪：コード

この章で議論してきたアイデアを実装しましょう。新しいプログラム network2.py を開発します。これはChapter 1で作成した network.py を修正したバージョンです。もし network.py の内容を覚えていなければ、ここで軽く復習しておくと後の助けになるでしょう。たった74行のコードですし、簡単に理解できます。

network.py と同様に、network2.py での中心選手も Network クラスです。このクラスはニューラルネットワークを表しています。Network のインスタンスを初期化するには、各層のニューロン数 sizes とコスト関数を指定する cost を指定します。コスト関数にはデフォルトでクロスエントロピー が指定されています：

```
class Network():

    def __init__(self, sizes, cost=CrossEntropyCost):
        self.num_layers = len(sizes)
        self.sizes = sizes
        self.default_weight_initializer()
        self.cost=cost
```

__init__ メソッド冒頭の2行は network.py と同じですし、見れば何をしているのか分かるでしょう。しかし続く2行は新しく、その挙動を詳細に理解する必要があります。

まずは default_weight_initializer を見てみましょう。このメソッドでは改善された手法で重みを初期化します。あるニューロンに入力する重みの数を n_{in} とすると、新しい手法ではそのニューロンに入力する重みを平均 0 で標準偏差 $1/\sqrt{n_{\text{in}}}$ の ガウス分布で初期化します。また、バイアスについては平均 0 で標準偏差が 1 のガウス分布で 初期化します。そのコードは以下の通りです：

```
    def default_weight_initializer(self):
        self.biases = [np.random.randn(y, 1) for y in self.sizes[1:]]
        self.weights = [np.random.randn(y, x)/np.sqrt(x)
                        for x, y in zip(self.sizes[:-1], self.sizes[1:])]
```

このコードを理解するために、まず np が線形代数計算を行う Numpy ライブラリーである ことを思い出しましょう。Numpy はプログラムの冒頭で import しています。第1層は入力層なので、この層ではバイアスを初期化していません。これは network.py で行ったことと同じです。

補足として、プログラムには large_weight_initializer メソッドも含まれています。このメソッドは、重みとバイアスをChapter 1で使った古い方法、つまり平均 0 で標準偏差 1 の ガウス分布で初期化します。コードは、default_weight_initializer とほんの僅かに違うだけです:

```
    def large_weight_initializer(self):
        self.biases = [np.random.randn(y, 1) for y in self.sizes[1:]]
        self.weights = [np.random.randn(y, x)
                        for x, y in zip(self.sizes[:-1], self.sizes[1:])]
```

large_weight_initializer メソッドを入れたのは、主にこの章の結果とChapter 1の 結果を簡単に比較できるようにするためです。実際問題として古い方法を使うべき状況はほとんど思いつきません！

Network クラスの __init__ メソッドに関する第2の変更点は、cost 属性を初期化する点です。この動作を理解するために、クロスエントロピーコスト関数を表すクラスを見てみましょう*:

```
class CrossEntropyCost:

    @staticmethod
    def fn(a, y):
        return np.nan_to_num(np.sum(-y*np.log(a)-(1-y)*np.log(1-a)))

    @staticmethod
    def delta(z, a, y):
        return (a-y)
```

*もしPythonの静的メソッドに馴染みがなければ、@staticmethod デコレータは 気にせず、fn や delta を普通のメソッドだと考えて差し支えありません。関心がある読者のために書いておくと、@staticmethod を付けることで、そのメソッドはそれが属しているブジェクトに依存していないことを表しています。fn や deltaメソッドが第1引数に self を受け取っていないのは そのためです。

中身を詳しく見ていきましょう。まず気づいてほしいのは、クロスエントロピー自体は 数学的にいえば「関数」ですが、Pythonのクラスとして実装されている点です。この実装を選んだのは、コスト関数がニューラルネットワークで2つの異なる役割を持つからです。その1つ目は、出力活性 a が正解の出力 y にどのくらい似ているかの指標の役割です。この役割は CrossEntropyCost.fn メソッドに実装されています。(脇道に逸れますが、CrossEntropyCost.fn の中で呼ばれている np.nan_to_num は ゼロに非常に近い数値の対数をNumpyで正しく扱うために必要です。)もう一つの役割を 理解するために、Chapter 2で議論した逆伝播法を思い出しましょう。逆伝播を行うには、ニューラルネットワークの出力誤差 δ^L を計算する必要があります。そして、出力誤差の形は選んだコスト関数ごとに異なります。クロスエントロピーの場合、出力誤差は式(66) で示した表式です:

$$\delta^L = a^L - y. \tag{92}$$

このため、第二のメソッド、CrossEntropyCost.delta を定義しました。このメソッドは出力誤差を計算します。この2つは ニューラルネットワークがコスト関数について知るべきことの全てですから、2つのメソッドを一つのクラスにまとめたのです。

同様に、network2.py は2乗コスト関数を表すクラスも含んでいます。これはChapter 1の結果と比較するためだけに入っており、今後の議論ではほとんど クロスエントロピーのみを用います。コードは下に示しました。QuadraticCost.fn メソッドは 実際の出力 a と正解出力 y から2乗誤差を計算します。QuadraticCost.delta から返される値は、Chapter 2で導出した2乗コスト関数の 出力誤差 (30) にもとづいています。

```
class QuadraticCost:

    @staticmethod
    def fn(a, y):
        return 0.5*np.linalg.norm(a-y)**2

    @staticmethod
    def delta(z, a, y):
        return (a-y) * sigmoid_prime_vec(z)
```

これで network2.py と network.py の主な違いは理解しました。どれも簡単なことです。以下で議論するように、L2正規化の実装など小さな変更点はたくさんあります。それらに取り掛かる前に、network2.py の完全なコードを見てみましょう。コード全体を細かく読んでいく必要はありません。その代わり、全体の構造を掴むようにしましょう。特に、ドキュメント文字列を読むのはプログラムの各部が何をしているのか理解する 助けになるでしょう。もちろん、お望みなら深く掘り下げたら良いでしょう。コードの読解に困ったら、下の文章を読んでからコードに戻ってても良いでしょう。ともかく、まずはコードを示します:

```
"""network2.py
~~~~~~~~~~~~~~

An improved version of network.py, implementing the stochastic
gradient descent learning algorithm for a feedforward neural network.
Improvements include the addition of the cross-entropy cost function,
regularization, and better initialization of network weights.  Note
that I have focused on making the code simple, easily readable, and
easily modifiable.  It is not optimized, and omits many desirable
features.

"""

#### Libraries
# Standard library
import json
import random
import sys

# Third-party libraries
import numpy as np


#### Define the quadratic and cross-entropy cost functions

class QuadraticCost:
```

```
        @staticmethod
        def fn(a, y):
            """Return the cost associated with an output ``a`` and desired output
            ``y``.

            """
            return 0.5*np.linalg.norm(a-y)**2

        @staticmethod
        def delta(z, a, y):
            """Return the error delta from the output layer."""
            return (a-y) * sigmoid_prime_vec(z)


    class CrossEntropyCost:

        @staticmethod
        def fn(a, y):
            """Return the cost associated with an output ``a`` and desired output
            ``y``.  Note that np.nan_to_num is used to ensure numerical
            stability.  In particular, if both ``a`` and ``y`` have a 1.0
            in the same slot, then the expression (1-y)*np.log(1-a)
            returns nan.  The np.nan_to_num ensures that that is converted
            to the correct value (0.0).

            """
            return np.nan_to_num(np.sum(-y*np.log(a)-(1-y)*np.log(1-a)))

        @staticmethod
        def delta(z, a, y):
            """Return the error delta from the output layer.  Note that the
            parameter ``z`` is not used by the method.  It is included in
            the method's parameters in order to make the interface
            consistent with the delta method for other cost classes.

            """
            return (a-y)


    #### Main Network class
    class Network():

        def __init__(self, sizes, cost=CrossEntropyCost):
            """The list ``sizes`` contains the number of neurons in the respective
            layers of the network.  For example, if the list was [2, 3, 1]
            then it would be a three-layer network, with the first layer
            containing 2 neurons, the second layer 3 neurons, and the
            third layer 1 neuron.  The biases and weights for the network
            are initialized randomly, using
            ``self.default_weight_initializer`` (see docstring for that
            method).

            """
            self.num_layers = len(sizes)
            self.sizes = sizes
            self.default_weight_initializer()
            self.cost=cost

        def default_weight_initializer(self):
            """Initialize each weight using a Gaussian distribution with mean 0
            and standard deviation 1 over the square root of the number of
            weights connecting to the same neuron.  Initialize the biases
```

```
        using a Gaussian distribution with mean 0 and standard
        deviation 1.

        Note that the first layer is assumed to be an input layer, and
        by convention we won't set any biases for those neurons, since
        biases are only ever used in computing the outputs from later
        layers.

        """
        self.biases = [np.random.randn(y, 1) for y in self.sizes[1:]]
        self.weights = [np.random.randn(y, x)/np.sqrt(x)
                        for x, y in zip(self.sizes[:-1], self.sizes[1:])]

    def large_weight_initializer(self):
        """Initialize the weights using a Gaussian distribution with mean 0
        and standard deviation 1.  Initialize the biases using a
        Gaussian distribution with mean 0 and standard deviation 1.

        Note that the first layer is assumed to be an input layer, and
        by convention we won't set any biases for those neurons, since
        biases are only ever used in computing the outputs from later
        layers.

        This weight and bias initializer uses the same approach as in
        Chapter 1, and is included for purposes of comparison.  It
        will usually be better to use the default weight initializer
        instead.

        """
        self.biases = [np.random.randn(y, 1) for y in self.sizes[1:]]
        self.weights = [np.random.randn(y, x)
                        for x, y in zip(self.sizes[:-1], self.sizes[1:])]

    def feedforward(self, a):
        """Return the output of the network if ``a`` is input."""
        for b, w in zip(self.biases, self.weights):
            a = sigmoid_vec(np.dot(w, a)+b)
        return a

    def SGD(self, training_data, epochs, mini_batch_size, eta,
            lmbda = 0.0,
            evaluation_data=None,
            monitor_evaluation_cost=False,
            monitor_evaluation_accuracy=False,
            monitor_training_cost=False,
            monitor_training_accuracy=False):
        """Train the neural network using mini-batch stochastic gradient
        descent.  The ``training_data`` is a list of tuples ``(x, y)``
        representing the training inputs and the desired outputs.  The
        other non-optional parameters are self-explanatory, as is the
        regularization parameter ``lmbda``.  The method also accepts
        ``evaluation_data``, usually either the validation or test
        data.  We can monitor the cost and accuracy on either the
        evaluation data or the training data, by setting the
        appropriate flags.  The method returns a tuple containing four
        lists: the (per-epoch) costs on the evaluation data, the
        accuracies on the evaluation data, the costs on the training
        data, and the accuracies on the training data.  All values are
        evaluated at the end of each training epoch.  So, for example,
        if we train for 30 epochs, then the first element of the tuple
        will be a 30-element list containing the cost on the
        evaluation data at the end of each epoch. Note that the lists
```

```
        are empty if the corresponding flag is not set.

        """
        if evaluation_data: n_data = len(evaluation_data)
        n = len(training_data)
        evaluation_cost, evaluation_accuracy = [], []
        training_cost, training_accuracy = [], []
        for j in xrange(epochs):
            random.shuffle(training_data)
            mini_batches = [
                training_data[k:k+mini_batch_size]
                for k in xrange(0, n, mini_batch_size)]
            for mini_batch in mini_batches:
                self.update_mini_batch(
                    mini_batch, eta, lmbda, len(training_data))
            print "Epoch %s training complete" % j
            if monitor_training_cost:
                cost = self.total_cost(training_data, lmbda)
                training_cost.append(cost)
                print "Cost on training data: {}".format(cost)
            if monitor_training_accuracy:
                accuracy = self.accuracy(training_data, convert=True)
                training_accuracy.append(accuracy)
                print "Accuracy on training data: {} / {}".format(
                    accuracy, n)
            if monitor_evaluation_cost:
                cost = self.total_cost(evaluation_data, lmbda, convert=True)
                evaluation_cost.append(cost)
                print "Cost on evaluation data: {}".format(cost)
            if monitor_evaluation_accuracy:
                accuracy = self.accuracy(evaluation_data)
                evaluation_accuracy.append(accuracy)
                print "Accuracy on evaluation data: {} / {}".format(
                    self.accuracy(evaluation_data), n_data)
            print
        return evaluation_cost, evaluation_accuracy, ¥
            training_cost, training_accuracy

    def update_mini_batch(self, mini_batch, eta, lmbda, n):
        """Update the network's weights and biases by applying gradient
        descent using backpropagation to a single mini batch.  The
        ``mini_batch`` is a list of tuples ``(x, y)``, ``eta`` is the
        learning rate, ``lmbda`` is the regularization parameter, and
        ``n`` is the total size of the training data set.

        """
        nabla_b = [np.zeros(b.shape) for b in self.biases]
        nabla_w = [np.zeros(w.shape) for w in self.weights]
        for x, y in mini_batch:
            delta_nabla_b, delta_nabla_w = self.backprop(x, y)
            nabla_b = [nb+dnb for nb, dnb in zip(nabla_b, delta_nabla_b)]
            nabla_w = [nw+dnw for nw, dnw in zip(nabla_w, delta_nabla_w)]
        self.weights = [(1-eta*(lmbda/n))*w-(eta/len(mini_batch))*nw
                        for w, nw in zip(self.weights, nabla_w)]
        self.biases = [b-(eta/len(mini_batch))*nb
                       for b, nb in zip(self.biases, nabla_b)]

    def backprop(self, x, y):
        """Return a tuple ``(nabla_b, nabla_w)`` representing the
        gradient for the cost function C_x.  ``nabla_b`` and
        ``nabla_w`` are layer-by-layer lists of numpy arrays, similar
        to ``self.biases`` and ``self.weights``."""
```

```
        nabla_b = [np.zeros(b.shape) for b in self.biases]
        nabla_w = [np.zeros(w.shape) for w in self.weights]
        # feedforward
        activation = x
        activations = [x] # list to store all the activations, layer by layer
        zs = [] # list to store all the z vectors, layer by layer
        for b, w in zip(self.biases, self.weights):
            z = np.dot(w, activation)+b
            zs.append(z)
            activation = sigmoid_vec(z)
            activations.append(activation)
        # backward pass
        delta = (self.cost).delta(zs[-1], activations[-1], y)
        nabla_b[-1] = delta
        nabla_w[-1] = np.dot(delta, activations[-2].transpose())
        # Note that the variable l in the loop below is used a little
        # differently to the notation in Chapter 2 of the book.  Here,
        # l = 1 means the last layer of neurons, l = 2 is the
        # second-last layer, and so on.  It's a renumbering of the
        # scheme in the book, used here to take advantage of the fact
        # that Python can use negative indices in lists.
        for l in xrange(2, self.num_layers):
            z = zs[-l]
            spv = sigmoid_prime_vec(z)
            delta = np.dot(self.weights[-l+1].transpose(), delta) * spv
            nabla_b[-l] = delta
            nabla_w[-l] = np.dot(delta, activations[-l-1].transpose())
        return (nabla_b, nabla_w)

    def accuracy(self, data, convert=False):
        """Return the number of inputs in ``data`` for which the neural
        network outputs the correct result. The neural network's
        output is assumed to be the index of whichever neuron in the
        final layer has the highest activation.

        The flag ``convert`` should be set to False if the data set is
        validation or test data (the usual case), and to True if the
        data set is the training data. The need for this flag arises
        due to differences in the way the results ``y`` are
        represented in the different data sets.  In particular, it
        flags whether we need to convert between the different
        representations.  It may seem strange to use different
        representations for the different data sets.  Why not use the
        same representation for all three data sets?  It's done for
        efficiency reasons -- the program usually evaluates the cost
        on the training data and the accuracy on other data sets.
        These are different types of computations, and using different
        representations speeds things up.  More details on the
        representations can be found in
        mnist_loader.load_data_wrapper.

        """
        if convert:
            results = [(np.argmax(self.feedforward(x)), np.argmax(y))
                       for (x, y) in data]
        else:
            results = [(np.argmax(self.feedforward(x)), y)
                       for (x, y) in data]
        return sum(int(x == y) for (x, y) in results)

    def total_cost(self, data, lmbda, convert=False):
        """Return the total cost for the data set ``data``.  The flag
```

```
            ``convert`` should be set to False if the data set is the
            training data (the usual case), and to True if the data set is
            the validation or test data.  See comments on the similar (but
            reversed) convention for the ``accuracy`` method, above.
            """
            cost = 0.0
            for x, y in data:
                a = self.feedforward(x)
                if convert: y = vectorized_result(y)
                cost += self.cost.fn(a, y)/len(data)
            cost += 0.5*(lmbda/len(data))*sum(
                np.linalg.norm(w)**2 for w in self.weights)
            return cost

        def save(self, filename):
            """Save the neural network to the file ``filename``."""
            data = {"sizes": self.sizes,
                    "weights": [w.tolist() for w in self.weights],
                    "biases": [b.tolist() for b in self.biases],
                    "cost": str(self.cost.__name__)}
            f = open(filename, "w")
            json.dump(data, f)
            f.close()

    #### Loading a Network
    def load(filename):
        """Load a neural network from the file ``filename``.  Returns an
        instance of Network.

        """
        f = open(filename, "r")
        data = json.load(f)
        f.close()
        cost = getattr(sys.modules[__name__], data["cost"])
        net = Network(data["sizes"], cost=cost)
        net.weights = [np.array(w) for w in data["weights"]]
        net.biases = [np.array(b) for b in data["biases"]]
        return net

    #### Miscellaneous functions
    def vectorized_result(j):
        """Return a 10-dimensional unit vector with a 1.0 in the j'th position
        and zeroes elsewhere.  This is used to convert a digit (0...9)
        into a corresponding desired output from the neural network.

        """
        e = np.zeros((10, 1))
        e[j] = 1.0
        return e

    def sigmoid(z):
        """The sigmoid function."""
        return 1.0/(1.0+np.exp(-z))

    sigmoid_vec = np.vectorize(sigmoid)

    def sigmoid_prime(z):
        """Derivative of the sigmoid function."""
        return sigmoid(z)*(1-sigmoid(z))

    sigmoid_prime_vec = np.vectorize(sigmoid_prime)
```

最も興味深い変更点の一つは、L2正規化を実装した点です。これは概念的には大きな変更ですが、その実装はコードを読んでも見逃してしまうほど些細なことです。この点に関わる変更の大部分は、様々なメソッド、特に Network.SGD に、パラメータ lmbda を渡すことです。そして本質的な変更は1行のプログラム、Network.update_mini_batch メソッドの 終わりから4行目です。その部分で、重み減衰するように勾配降下法の更新規則を修正しています。しかし、その修正はわずかでも、結果へは大きな影響を与えるのです！

ところで、このようなことはニューラルネットワークで新しい技術を実装する時に しばしば起こります。正規化を議論するのに大変長い文章を費やしました。正規化は概念的にはとても微妙で理解するのも難しいものです。それでも、プログラムに実装するのはとても簡単なことなのです。洗練された高度な手法が、コードの僅かな変更で実装できてしまうことは、驚くほど頻繁に起こります。

他の小さくても重要な変更点は、確率的勾配降下法を実行するメソッド Network.SGD に いくつかのオプションフラグを追加したことです。これらのフラグは、training_data か evaluation_data でのコストと精度をモニターするために用いられます。コストと精度を計算するデータ集合は Network.SGD に渡すことができます。この章の中でもこれらのフラグを何度も使ってきましたが、思い出してもらうために、今一度使い方の例を示しておきます:

```
>>> import mnist_loader
>>> training_data, validation_data, test_data = ¥
... mnist_loader.load_data_wrapper()
>>> import network2
>>> net = network2.Network([784, 30, 10], cost=network2.CrossEntropyCost())
>>> net.SGD(training_data, 30, 10, 0.5,
... lmbda = 5.0,
... evaluation_data=validation_data,
... monitor_evaluation_accuracy=True,
... monitor_evaluation_cost=True,
... monitor_training_accuracy=True,
... monitor_training_cost=True)
```

ここでは、evaluation_data に validation_data をセットしていますが、パフォーマンスのモニターには test_data を使うこともできますし、他のデータセットを使うこともできます。その他に4つのフラグをセットして、コストと精度を evaluation_data と training_data の両方でモニターするよう設定しています。これらのフラグは デフォルトで False ですが、ここでは True にセットして Network のパフォーマンスをモニターしています。さらに、network2.py の Network.SGD メソッドは、モニタリングの結果を表す4要素のタプルを返します。これは次のように利用します:

```
>>> evaluation_cost, evaluation_accuracy,
... training_cost, training_accuracy = net.SGD(training_data, 30, 10, 0.5,
... lmbda = 5.0,
... evaluation_data=validation_data,
... monitor_evaluation_accuracy=True,
... monitor_evaluation_cost=True,
... monitor_training_accuracy=True,
... monitor_training_cost=True)
```

ですから、例えば、evaluation_cost は30個の要素を持つリストで、それぞれの要素は各エポック終了後の evaluation_data に対するコストです。この種の情報はニューラルネットワークの振る舞いを理解する上で非常に有用です。例えば、時間とともに学習が進行する様子をグラフに描く時に使えます。実際、この章で示したグラフはこのようにして描いたものなのです。ただし、もしモニタリングのフラグが設定されていなければ、対応するタプルの要素は空リストになることに注意してください。

この他に追加した機能としては、Network オブジェクトをディスクに保存する Network.save メソッドと、保存した Network オブジェクトを 後から読み込む load 関数があります。保存と読み込みは、Pythonで標準的に使われる pickle や cPickle ではなく、JSONを使って行っています。JSONを使うほうが pickle や cPickle に比べて 多くのコードを必要とします。JSONを使った理由は、将来 Network クラスを修正する 可能性があるからです。例えば、シグモイドニューロンではなく他のニューロンを使うように Network を修正するとしましょう。この変更を実装するために、Network.__init__ メソッドで定義する属性を変更することになるでしょう。すると、単純に pickle を用いてしまうと load 関数は失敗してしまいます。JSONを使いシリアル化を明示的に行うことで、古い Network クラスをより簡単に確実に読み込むことができます。

他にも network2.py のコードにはたくさんの小さな変更があります。しかし、それらは全て network.py からの単純な変化に過ぎません。全体としては、74行のプログラムをより強力な152行に拡大する結果になりました。

問題

- 上のコードを修正して、L1正規化を実装してください。また、30個の隠れニューロンを持つニューラルネットワークで、L1正規化を使ってMNISTの手書き数字の分類を行ってください。正規化を行わない場合よりも精度が良くなる正規化パラメータは見つかりますか？

- network.py の Network.cost_derivative メソッドを見てください。このメソッドは2乗コスト関数用に書かれています。これをクロスエントロピーコスト関数用に書き直すにはどうしたら良いでしょうか？ また、Network.cost_derivative メソッドのクロスエントロピー版で生じる問題

を指摘してください。network2.py では、Network.cost_derivative メソッドを完全に 無くしてしまい、代わりにその機能を CrossEntropyCost.delta メソッドに 実装しました。こうすることで、先に指摘した問題がどう解決されますか？

ニューラルネットワークのハイパーパラメータをどう選ぶか?

ここまでは、学習率 η や正規化パラメータ λ 等のハイパーパラメータを私がどのように決めたのか説明せず、ニューラルネットワークが上手く働くよう私が 事前に決めた値を使ってきました。実際問題として、ニューラルネットワークを使って何かの 課題に取り組むにあたって、良いハイパーパラメータを見つけるのは困難な場合があります。例えば、MNISTの手書き数字分類問題を今ここで初めて知って、適切なハイパーパラメータの値を知らずにこれに取り掛かることを想像してみてください。その際に最初の実験から、多くのハイパーパラメータについてこの章で使ってきた値、30個の隠れニューロン、大きさ10のミニバッチ、そしてクロスエントロピーを使った30エポックの訓練を 運良く選んだとしましょう。しかし学習率については $\eta = 10.0$、そして正規化パラメータは $\lambda = 1000.0$ を選んだとします。すると次のような結果が得られます:

```
>>> import mnist_loader
>>> training_data, validation_data, test_data = ¥
... mnist_loader.load_data_wrapper()
>>> import network2
>>> net = network2.Network([784, 30, 10])
>>> net.SGD(training_data, 30, 10, 10.0, lmbda = 1000.0,
... evaluation_data=validation_data, monitor_evaluation_accuracy=True)
Epoch 0 training complete
Accuracy on evaluation data: 1030 / 10000

Epoch 1 training complete
Accuracy on evaluation data: 990 / 10000

Epoch 2 training complete
Accuracy on evaluation data: 1009 / 10000

...

Epoch 27 training complete
Accuracy on evaluation data: 1009 / 10000

Epoch 28 training complete
Accuracy on evaluation data: 983 / 10000

Epoch 29 training complete
Accuracy on evaluation data: 967 / 10000
```

なんと分類精度は全くの当てずっぽうと同等です！ ニューラルネットワークはサイコロのようなものになってしまいました！

こんな風に考える読者がいるかもしれません。「ふむ、こんなのは簡単に直せるよ。学習率と正規化ハイパーパラメータを小さくすれば良いんだろう」。しかしちょっと待ってください。そう言えるのは上手く働くハイパーパラメータを 事前に知っていたからであって、学習率と正規化パラメータを調整すれば上手くいくことを ア・プリオリには知らないのです。もしかすると、隠れニューロンの数が30個では 他のハイパーパラメータをいかに調整しても上手くいかないのかもしれません。もしかすると少なくとも100個、いや300個は隠れニューロンが必要なのでは？ あるいは隠れ層が複数必要なのでは？いや出力のエンコーディングが問題なのかもしれませんよ？ 本当は学習が進行しているけれども、もっと多くのエポックが必要という可能性はありませんか？ ミニバッチの大きさが小さすぎませんか？2乗コスト関数にしてみてはどうでしょう？ 重みの初期化方法を変えてみるのは？等など。広すぎるハイパーパラメータの空間で 途方に暮れてしまいそうです。巨大なニューラルネットワークを扱っていたり 巨大な訓練データを使ったりしている時には特に苛立たしい問題です。何しろ、訓練するのに何時間も何日も、もしかすると何週間も掛かるのに、何の結果も得られないこともあるのですから。このような状況が続けば自信を無くすかもしれません。もうこんな仕事は辞めて田舎で畑仕事でも始めたほうがマシに思えるかもしれません。

この節では、ニューラルネットワークのハイパーパラメータを定める発見的な方法を いくつか紹介します。目標は、ハイパーパラメータを上手に決めるためのワークフローを 読者が確立する手助けをすることです。もちろん、ハイパーパラメータの最適化について 全てをカバーするわけではありません。何しろとても大きな問題です。そして、完全に解決されている問題ではないし、ニューラルネットワークを使う人々の間で 普遍的に共有されている合意も無いのです。ニューラルネットワークのパフォーマンスをほんの少し向上させるために、常に一つ、何かできることがあるはずです。この節で紹介する発見的な方法は、その取っ掛かりを与えてくれるでしょう。

出発点：ニューラルネットワークを使って新しい問題に取り組む時、第1の関門は**何でも良いので**とにかく何か学習をすることです。言い換えると、ニューラルネットワークでデタラメよりはマシな結果を達成することが第1歩です。ものすごく低いハードルに思えるかもしれませんが、実際にはそれが驚くほど難しい場合もあります。特に、新しい種類の問題に取り組む時には、この取っ掛かりから困難な場合があります。まずはこの種の困難に遭った時に採りうる戦略を見てみましょう。

例えば、あなたがMNISTに初めて取り組むとしましょう。勢い勇んで取り掛かったものの、ニューラルネットワークで得た最初の結果が先程見たよ

うな散々なもので、少々落ち込んでいます。この状況を何とかするには、まず生じている問題を特定することです。手始めに、訓練データと検証データから0と1以外の画像を全て取り除きましょう。そして、ニューラルネットワークが0と1を区別できるように訓練してみましょう。この0と1を区別するという新しい問題は、元々の10種類の数字を区別する問題より簡単であるのみならず、訓練データの量を80%減らすことになるので、訓練が5倍速くなります。こうすることで問題を特定するための実験を素早く実行することが可能になりますし、良いニューラルネットワークを構築するにはどうしたら良いのか、素早い見立てが可能になります。

さらに素早く実験を行うには、まともに学習が可能であると期待できる範囲で、ニューラルネットワークをできるだけ単純化することができます。もし各層のニューロン数が [784, 10] のニューラルネットワークが MNISTの手書き数字分類問題に対して乱数よりはまともな正答率を達成できると考えるなら、実験の出発点としてこのようなニューラルネットワークから実験を採用するのが良いでしょう。ニューロン数が [784, 30, 10] のニューラルネットワークよりはずっと速く学習するでしょうから、実験も速く進むでしょう。

実験をさらにスピードアップするには、モニタリングの頻度を上げるという方法があります。network2.py では、訓練の各エポックが終了した時点でパフォーマンスを測定しました。一つのエポックに50,000枚の画像が含まれるので、ニューラルネットワークの学習状況について 情報を得るのに随分と時間が掛かります。例えば、私のノートPCで各層のニューロン数が [784, 30, 10] のニューラルネットワークを訓練すると、学習状況についてフィードバックを受けるまで約1分掛かります。もちろん、1分というのは決して長時間ではありませんが、大量のハイパーパラメータを決めたい時にはそれでも厄介です。何百、何千通りものハイパーパラメータを試そうと思うと、随分と大きな負担です。もっと頻繁に、例えば訓練画像1,000枚ごとに検証精度を監視することで、より素早いフィードバックが得られます。さらに、10,000枚の画像を全て使う代わりに100枚の検証画像から精度を推定することで、より速くパフォーマンスを測定することが可能です。大切なのは、まともな学習に必要なだけの訓練画像をニューラルネットワークに与えることと、そこそこに良いパフォーマンスの推定を得ることなのです。もちろん、network2.py は現時点でこういうモニタリングを行ってはいませんが、ここでは実例として、MNISTの訓練データを1,000枚に減らしてみます。動かしてみて何が起こるか見てみましょう。(単純にするために、0と1の画像だけを使うというアイデアは実装していません。もちろん、あと少し工夫すればそれを実装することも可能です。)

```
>>> net = network2.Network([784, 10])
>>> net.SGD(training_data[:1000], 30, 10, 10.0, lmbda = 1000.0, ¥
```

```
... evaluation_data=validation_data[:100], ¥
... monitor_evaluation_accuracy=True)
Epoch 0 training complete
Accuracy on evaluation data: 10 / 100

Epoch 1 training complete
Accuracy on evaluation data: 10 / 100

Epoch 2 training complete
Accuracy on evaluation data: 10 / 100
...
```

まだ出てくるのは純粋なノイズです！ でも一つ大きな進歩があります：今やフィードバックを得るのに1分も待つ必要がなく、1秒毎に情報が得られます。これなら、ハイパーパラメータを変えて実験を繰り返すのも容易ですし、あるいは様々な異なるハイパーパラメータをほぼ同時に試してみることさえも可能でしょう。

上の例では、λ を以前に使った $\lambda = 1000.0$ のままにしておきました。しかし、訓練データの数を変えたのだから、重み減衰の割合を一定に保つために λ も 変更するべきです。この場合は、$\lambda = 20.0$ にすると前と重み減衰の因子が前と同じになります。以下がその結果です：

```
>>> net = network2.Network([784, 10])
>>> net.SGD(training_data[:1000], 30, 10, 10.0, lmbda = 20.0, ¥
... evaluation_data=validation_data[:100], ¥
... monitor_evaluation_accuracy=True)
Epoch 0 training complete
Accuracy on evaluation data: 12 / 100

Epoch 1 training complete
Accuracy on evaluation data: 14 / 100

Epoch 2 training complete
Accuracy on evaluation data: 25 / 100

Epoch 3 training complete
Accuracy on evaluation data: 18 / 100
...
```

なんと、今度はシグナルが出ました！とても良い結果とは言えませんが、ともかく偶然よりは良い結果です。これは、さらなる改善を得るためにハイパーパラメータを修正していく際の出発点になります。学習率をもっと大きくするべきだと推測するかもしれません。（恐らく皆さんも気づいているように、これは愚かな推測です。その理由についてはすぐに議論しますが、しばらくは我慢してください。）そこでその推測を検証するために、η の値を 100.0 まで上げてみましょう：

```
>>> net = network2.Network([784, 10])
>>> net.SGD(training_data[:1000], 30, 10, 100.0, lmbda = 20.0, ¥
... evaluation_data=validation_data[:100], ¥
... monitor_evaluation_accuracy=True)
Epoch 0 training complete
Accuracy on evaluation data: 10 / 100
```

```
Epoch 1 training complete
Accuracy on evaluation data: 10 / 100

Epoch 2 training complete
Accuracy on evaluation data: 10 / 100

Epoch 3 training complete
Accuracy on evaluation data: 10 / 100

...
```

これはひどい！この結果は、学習率が小さすぎるという推測が間違っていたことを示唆します。そこで反対に、η を 1.0 まで小さくしてみましょう:

```
>>> net = network2.Network([784, 10])
>>> net.SGD(training_data[:1000], 30, 10, 1.0, lmbda = 20.0, ¥
... evaluation_data=validation_data[:100], ¥
... monitor_evaluation_accuracy=True)
Epoch 0 training complete
Accuracy on evaluation data: 62 / 100

Epoch 1 training complete
Accuracy on evaluation data: 42 / 100

Epoch 2 training complete
Accuracy on evaluation data: 43 / 100

Epoch 3 training complete
Accuracy on evaluation data: 61 / 100

...
```

こちらの方が良いですね！このように続けていって、ハイパーパラメータを一つずつ調整して、パフォーマンスを徐々に改善していきます。学習率 η の値を改善できたら、次に正規化パラメータ λ の改善に進みます。それができたら、次はより複雑な構造のニューラルネットワーク、例えば隠れニューロンを10個持つ ニューラルネットワークで実験しましょう。前と同じように η と λ を調整します。次に隠れニューロンを20個に増やして、他のハイパーパラメータも調整しましょう。以下同様に、各段階で検証データの一部を使ったパフォーマンスの測定を行い、その結果を参考にしながらより良いハイパーパラメータを見つけていきます。これを続けていくうちに、しばしばハイパーパラメータを修正した効果が現れるのに 時間が掛かるようになるので、モニタリングの頻度を徐々に下げることができます。

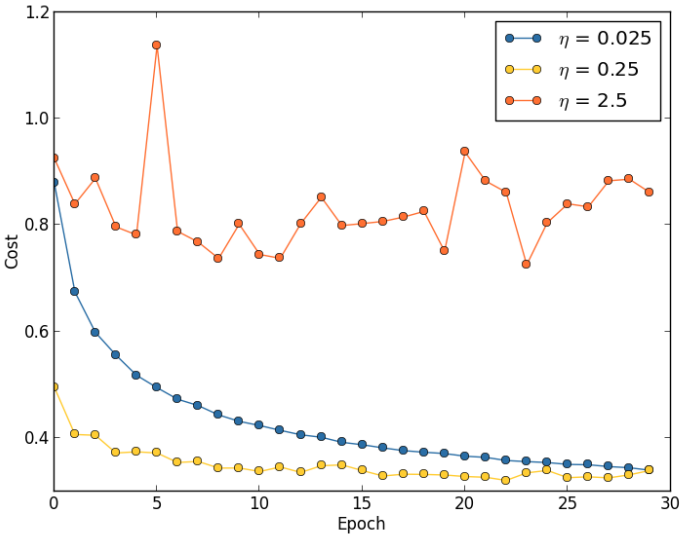
これは一般的でとても有望な戦略に見えます。しかし、実際のところ、上の議論でさえ見通しが楽観的すぎるのです。何も学習しないニューラルネットワークをいじるのは、どうしようもなく骨の折れる作業です。何日もハイパーパラメータの調整を行って、それでも無意味な結果しか得られないことだってあります。ですから、改めて強調したいのですが、初期の

段階では、実験をしたらできるだけ素早くフィードバックが得られるようにしておくことがとても重要なのです。最初に問題やニューラルネットワークの構造を単純化してしまうと、当初の目的から遠ざかってしまうように思えるかもしれません。しかし、こうすることでニューラルネットワークが意味のある学習をするまでに掛かる時間はずっと短縮されるし、一旦意味のあるシグナルが得られれば、ハイパーパラメータを調整することで ニューラルネットワークの性能はしばしば急速に向上していきます。一番最初の取っ掛かりが、最も苦労するところなのです。

ここまでは一般的な話をしてきました。ここからは、ハイパーパラメータの決め方について、私が勧める具体的な手順をいくつか見てみましょう。以下では、学習率 η、L2正規化パラメータ λ、そしてミニバッチのサイズの決め方を議論します。しかし、ここで述べることの多くは、ニューラルネットワークのアーキテクチャや他の正規化法に関わるハイパーパラメータ、あるいはモーメンタム係数などこの本で後に議論するハイパーパラメータ等にも当て余ります。

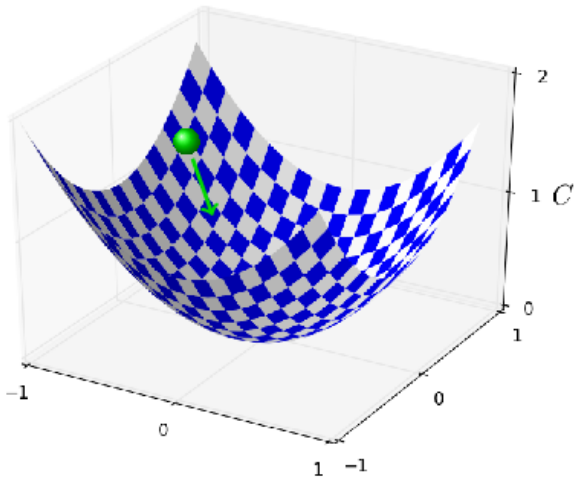
学習率: 3つの異なる学習率、$\eta = 0.025$、$\eta = 0.25$、$\eta = 2.5$ を 持つニューラルネットワークにMNISTの学習をさせることを考えます。他のハイパーパラメータについては、以前に用いたのと同様、訓練は30エポック行い、ミニバッチのサイズは10、正規化パラメータは $\lambda = 5.0$ を採用します。また、ここでは $50,000$ 枚全ての訓練画像を用いることにします。次のグラフは、訓練の経過とともにコストがどう変化するかを示しています*:

*このグラフは multiple_eta.pyを用いて生成したものです。



学習率 $\eta = 0.025$ では、最後のエポックまで滑らかにコストが減少しています。$\eta = 0.25$ では、コストが最初減少しますが、20エポックを過ぎた辺りでほぼ飽和して、その後はランダムに見える小さな振動をします。最後に、$\eta = 2.5$ では、はじめからコストは大きな振動をしています。この振動が起こる理由を理解するために、確率的勾配降下法ではコストの谷底に

向かって小さなステップで 徐々に学習を進めていくことを思い出しましょう:



しかし、もし学習率 η が大きすぎると、ステップが大きすぎてコストの最小を通り過ぎてしまう 可能性があります。そうすると、谷底に下る代わりに登ってしまうでしょう。これが恐らく $\eta = 2.5$ の時に起こっているのでしょう*。$\eta = 0.25$ の場合は、最初しばらくはコスト関数の最小に向かって進み、最小に近づいた時だけ行き過ぎの問題が生じます。$\eta = 0.025$ を選ぶと、最初の30エポックでは同様の問題は一切生じません。もちろん、これほど小さな学習率を選ぶと、確率的勾配降下法による学習が遅くなるという別の問題が生じます。より良い方法は、$\eta = 0.25$ から始めて20エポック訓練し、それから $\eta = 0.025$ に切り替えることでしょう。このように学習率を変動させる手法については後ほど議論します。しばらくは、一つの良い学習率 η をどう選ぶかについて考えましょう。

*この描像は分かりやすいですが、あくまでも直感的な説明として示したもので、完全な説明にはなっていません。より完全な説明は、手短に言うと次のようなものです: 勾配降下法では、コスト関数に対する一次近似を手がかりにコストを減少させようと試みます。大きな η では、コスト関数のより高次の項がより重要になり、場合によってはコスト関数の振る舞いを決定づけるので、勾配降下法が破綻してしまいます。これは特にコスト関数の最小や極小に近づくと起こりやすくなります。というのも、そのような領域では勾配が小さくなり、高次の項が支配的になりやすくなるためです。

この描像を頭に入れておくと、η を次のように定めることができます。まず、コスト関数が振動や増加をせずちゃんと減少してくれる学習率の上限を見積もります。この見積もりがとても正確である必要はありません。桁がわかれば良いという程度です。ですから、$\eta = 0.01$ から始めて数エポックの間コストが減少すれば、順番に $\eta = 0.1, 1.0, \ldots$ と試していき、コストが最初の数エポックで振動や増加を する値を見つけます。逆に、$\eta = 0.01$ でも最初の数エポックでコストが振動や増加を始めたら、$\eta = 0.001, 0.0001, \ldots$ と試してい、コストが数エポックの間減少する値を見つけます。これでコストを減少させられる学習率 η の上限が大雑把に見積もられます。もっと細かく、例えば $\eta = 0.5$ や $\eta = 0.2$ のような値を試してみることで、上限の見積もりを改善することも可能です(ただしこれが非常に正確である必要はありません)。

実際に使うηの値はこの上限より小さく取らねばなりません。実際のところ、選んだ学習率ηが多くのエポックに渡って使い物になるためには、例えば、上限の半分くらいにηをとるのが良いかもしれません。そのように選ぶことで、通常は学習の速度を落とし過ぎることなく、より多くのエポックを使って訓練することが可能になります。

MNISTの場合には、この戦略に従うと、コストを減少させられる学習率の上限が 桁で言って0.1程度であるという見積もりを得ます。より細かく見ていくと、$\eta = 0.5$程度が上限であることが分かります。上記の処方に従うと、学習率として$\eta = 0.25$を選ぶと良さそうだと言えます。実際のところ、$\eta = 0.5$でも30エポックくらいなら十分に良く学習しますが、たいていいはより小さなηを選んでも問題ありません。

ここで述べたことは、どれもとても単純に思えます。しかし、ηを選ぶために訓練データに対するコスト関数の値を用いるのは、この節で前に述べたことと矛盾するのではないでしょうか？つまり、ハイパーパラメータを選ぶには、取っておいた("held-out")検証データを用いると言ったはずですよね？ 実際、正規化パラメータ、ミニバッチのサイズ、そして隠れ層や隠れニューロンの数など ニューラルネットワークのパラメータを定めるために、検証データに対する精度を用います。では、学習率についてはなぜ状況が異なるのでしょうか？ 率直に言って、この選択は私の美的な好みによるもので、たぶん多少特殊ですが、次のように理由付けできます。他のハイパーパラメータが試験データに対する最終的な分類精度の改善を 目的とするので、検証データに対する分類精度に基づいてチューニングするのは理にかなっています。しかし、学習率が最終的な分類精度に与える影響は二義的なものです。学習率をチューニングする一義的な目的は勾配降下法のステップ幅を制御することですから、訓練データに対するコストを監視するのが大きすぎるステップ幅を検出する最良の方法であるはずです。とは言え、あくまでもこれは私の個人的な美的趣味の問題です。検証データに対する分類精度が改善するなら、学習の初期段階に訓練コストは減少するでしょうから、実際上は検証データの分類精度と訓練コストのどちらを基準としても大した差は生じないと考えられます。

訓練エポック数を決める手段としての早期打ち切り: この章で前に議論したように、早期打ち切りとは各エポックが終了するたびに検証データに対する 分類精度を計算し、その改善が止まったら訓練を打ち切るという処方です。こうすることで、とても簡単にエポック数を定めることができます。この処方が特に便利なのは、訓練エポック数がその他のハイパーパラメータに対してどう依存するのか 気にする必要が無い点です。他のハイパーパラメータを変えると必要なエポック数も変わりますが、その効果

は自動的に取り込まれます。その上、早期打ち切りは自動的に過適合を防ぎます。これはもちろん良いことです。ただし、実験の初期の段階では、あえて早期打ち切りを行わない方が良い場合もあります。早期打ち切りを止めることで過適合を起こし、その様子を調べることで正規化の指針を得るのです。

早期打ち切りを実装するには、まずは「分類精度の改善が止まる」という一文が意味するところを より正確に述べておく必要があります。既に見たように、分類精度が全体的に改善の傾向にあっても、エポックを重ねるごとに改善し続けるわけではなく上がったり下がったりを繰り返すこともあります。もし分類精度が始めて減少した時点で訓練をやめてしまうなら、ほぼ間違いなく改善の余地を残してしまうことになるでしょう。もっと良いやり方は、分類精度の最高値がしばらく更新されなかった時点で打ち切ることです。例えば、MNISTの問題を考えましょう。そこで、分類精度が過去10エポックに渡って改善しなかった 時点で訓練を打ち切ることにするとどうでしょう。こうしておけば、訓練中の不運のために 早すぎる打ち切りを行ってしまうことは無いでしょうし、全く改善の得られないまま永久に待ち続ける心配も無いでしょう。

この「10エポック・ルール」はMNIST問題の取っ掛かりに用いるには良いでしょう。しかし、ニューラルネットワークでは時折、長期間に渡って分類精度が停滞した後に 再び改善を始める場合があります。もし本当に良いパフォーマンスを目指すなら、10エポック・ルールでは忍耐が足りないかもしれません。その場合の私の提案は、初期の実験で10エポック・ルールを採用し、ニューラルネットワークの訓練状況が明らかになるに従い徐々に改善を待つ期間を長くしていくことです。20エポック・ルール、50エポック・ルール、というように。この戦略はもちろん、最適化の必要な新たなハイパーパラメータを生み出してしまいます！ しかし実際には、多くの場合このハイパーパラメータを決めるのは難しくありません。MNIST以外の問題に対しても同様に、10エポック・ルールが早すぎるかそうでもないかは 問題の詳細に依存することですが、普通は少し実験をしてみれば早期打ち切りの良い戦略が 簡単に見つかるものです。

これまで示したコードでは、MNIST問題の実験で早期打ち切りを用いていません。その理由は、多くの異なる学習手法を試して比較してきたからです。このような比較では、それぞれの場合で同数のエポックを使うのが便利でしょう。しかし、読者の皆さんは是非自分で `network2.py` を修正して早期打ち切りを実装してみてください：

問題

- network2.py を修正して「nエポック・ルール」を使った早期打ち切りを実装してください。ここで、n は指定できるパラメータとします。

- 「nエポック・ルール」以外に早期打ち切りの方法を挙げてみましょう。理想的には、検証データに対する分類精度を上げることと訓練時間を短縮することを 上手くバランスさせるような規則であるべきです。そのような規則を network2.py に 追加した上で実験を3回行い、検証精度と訓練エポック数を10エポック・ルールと比較してください。

  学習率のスケジューリング: これまでは訓練の間、学習率 η を定数に固定してきました。しかし、しばしば学習率を変化させることが有効です。学習過程の初期には、重みがひどく間違った値を取っているでしょうから、大きな学習率を用いて学習の進みを早めるのが良いでしょう。その後、学習率を小さくしていくことで、重みをより精密に調整することができます。

  学習率を訓練中に変更する具体的な手順、学習率のスケジューリングには、様々なものが考えられます。一つの自然なアプローチは、早期打ち切りと同様の発想に基づくものです。それは、まずは学習率を固定しておき、検証精度が悪くなり始めると、学習率を例えば半分や10分の1程度に減らすというものです。これを学習率が、例えば初期値の1,024分の1(あるいは1,000分の1)になるまで 繰り返してから学習を終了します。

  学習率のスケジューリングはパフォーマンスを改善しますが、学習率のスケジューリングを決定するという新たな自由度をもたらします。この選択が頭痛の種になりえます。学習率のスケジューリングを決めるために 無限の時間を掛けて試行錯誤することも可能でしょう。ですから、最初の実験に対する私の提案は、ただ一つ、定数の学習率を用いることです。それで良い第1近似が得られるでしょう。その後、ニューラルネットワークの発揮しうる最高のパフォーマンスを実現したいと考えるなら、上で述べたような学習率のスケジューリングを試すと良いでしょう*。

*MNIST問題に取り組む際に学習率を変化させるご利益を示した最近の読みやすい論文として、Deep, Big, Simple Neural Nets Excel on Handwritten Digit Recognition, by Dan Claudiu Cireşan, Ueli Meier, Luca Maria Gambardella, and Jürgen Schmidhuber (2010) を勧めます。

Exercise

- network2.py を修正して、次のような学習率のスケジューリングを実装してください: 検証精度が10エポック・ルールを満たすごとに学習率を半分にし、学習率が初期値の 1/128 に なった段階で訓練を終了させてください。

正規化パラメータ λ: まずは正規化無し($\lambda = 0.0$)で、上述のように学習率 η を最適化しましょう。そこで決めた η を使って、λ の良い値を選びます。その時は、$\lambda = 1.0$* から 始めて、検証データに対する分類精度が改善するように λ を10倍、あるいは10分の1倍していきます。一旦 λ の良い大きさがどのくらいの桁にあるのか分かったら、さらに細かく λ の値を調整しても良いでしょう。それが終わったら、再び η を最適化し直しましょう。

*初期値として $\lambda = 1.0$ を使うべき原理的な理由があるわけではありません。λ の探索をどこから始めるべきか、原理的な議論を知っている人がいたら、教えていただけるとありがたく思います (mn@michaelnielsen.org)。

演習

- λ や η などのハイパーパラメータの良い値を決めるために、勾配降下法を使いたくなるかもしれません。勾配降下法を使った λ の決定にはどのような障害があるでしょうか？ また、η の決定に勾配降下法を使うことについてはどうでしょうか？

私がこの本で使ったハイパーパラメータの選び方: この節で紹介してきた手順に従うと、この本でこれまで使ってきた η や λ の値と 必ずしも一致しない値が得られるでしょう。そうなる理由は、本書の中では説明の都合という制約のために、いつも最適なハイパーパラメータを使うことができなかったためです。これまで見てきた学習手法の比較を思い返してみましょう。例えば、2乗コスト関数とクロスエントロピーコスト関数の比較、重みの初期化に関する新旧2つのアプローチの比較、正規化の有無の比較、等など。このような比較を意味あるものにするために、異なるアプローチで同じハイパーパラメータ (あるいは適切にスケールしたハイパーパラメータ)を使うように心掛けました。当然、採用する学習手法によってハイパーパラメータの最適値は異なるのが普通です。ですから、異なる手法を同じハイパーパラメータで比較するには、最適値をある程度諦めることになります。

これに対する代案として、比較対象の手法のそれぞれでまず最適なハイパーパラメータの値を 求めておいて、最適な場合のパフォーマンス同士を比較するという方法も考えられます。原理的には、こちらの方がより公平で良いアプローチかもしれません。というのも、そうすることでそれぞれの手法の発揮しうる最高の性能を見ることができるからです。しかし実際的には、様々な手法を取り上げ比較していく中で、計算量が多すぎるのです。それが、私がこの本で部分的に最適なハイパーパラメータの使用を諦めた理由です。

ミニバッチのサイズ: ミニバッチの大きさはどう決めるべきでしょうか？この問いに答えるために、まずはオンライン学習、つまりミニバッチのサイズが 1 の学習を考えましょう。

まず心配になるのは、たった一つの訓練例しか含まないミニバッチでは、推定した勾配が大きな誤差を持つのではないかということです。しかし実際のところ、この誤差は大した問題でないことがわかります。勾配の推定値が一つ一つ非常に正確である必要はないのです。必要なのは、学習を進める間、コスト関数の減少傾向が続く程度の正確さです。例えるなら、見るたびに10度か20度くらいズレる気まぐれなコンパスを持って北極を目指すようなものです。コンパスが平均的には正しい方向を指して、しかも十分頻繁にコンパスを確認すれば、最終的には北極にたどり着くことができるでしょう。

この議論からは、オンライン学習がベストのように思えます。ところが実際には、状況はもっと複雑です。前の章の問題では、行列のテクニックを使うと、ミニバッチに含まれる訓練例に関するループを用いることなく、ミニバッチ全体を使って一度に勾配の計算ができます。計算を行うハードウェアや 線形代数ライブラリの実装によっては、個別の訓練例を使った勾配計算を100回行うより、100個の訓練例を含むミニバッチについて行列を使って勾配を計算した方がずっと早いことも あるのです。行列を使うと、1個の訓練例で勾配を計算するより、例えば、100倍ではなく50倍しか時間が掛からないかもしれません。

今の話は大して役に立たないと感じるかもしれません。100個の訓練例を含むミニバッチを使って重みを学習する規則は次のようなものです:

$$w \rightarrow w' = w - \eta \frac{1}{100} \sum_x \nabla C_x, \tag{93}$$

ここで、ミニバッチが含む訓練例に関する和を取っています。一方、オンライン学習では、

$$w \rightarrow w' = w - \eta \nabla C_x \tag{94}$$

となります。いくらミニバッチ更新に50倍の時間しか掛からないからと言っても、まだオンライン学習の方が50倍頻繁に更新できるわけですから、やはりオンライン学習の方が良いと結論してしまいそうです。しかしもし、ミニバッチ学習の場合に学習率を100倍できるとしたらどうでしょう。この場合、更新規則は次のように変わります:

$$w \rightarrow w' = w - \eta \sum_x \nabla C_x. \tag{95}$$

こうなると、100個の個別の訓練例について、学習率 η でオンライン学習を行うのと同じに見えます。しかも、1個の訓練例について学習するより50倍しか時間が掛からないのです。もちろん、このミニバッ

チ学習はオンライン学習を100個の訓練例で実行するのと 全く同じというわけではありません。オンライン学習では個別の例を学習するごとに 重みを更新し、次の例では新しい重みを用いて勾配の計算を行うのに対して、ミニバッチ学習では100個の訓練例について同じ重みを用いて計算を行うからです。この点を考慮してもなお、ミニバッチ学習がオンライン学習よりも速い可能性は十分にあります。

これらの要因を考慮すると、最適なミニバッチサイズはある種の妥協のもとで決まると言えます。もし小さすぎれば、高速なハードウェアに最適化された行列ライブラリの恩恵を十分に受けられません。もし大きすぎれば、重みの更新頻度が下がってしまいます。やりたいのは、学習スピードを最大化するようなミニバッチサイズを決めることです。幸い、学習スピードを最大化するミニバッチサイズは、ニューラルネットワークの アーキテクチャ以外のハイパーパラメータに対して比較的独立に決められるので、良いミニバッチサイズを見つける前に予め他のハイパーパラメータを最適化しておく必要はありません。ミニバッチサイズ以外のハイパーパラメータについては程よい値を入れておいて、学習率 ηも上述のようにスケーリングしながら異なるミニバッチサイズを試してみれば良いのです。そして、検証データに対する精度を**時間**についてプロットしましょう。ここで、時間はエポック数ではなく、掛かった実時間であることに注意してください。このプロットに基づいて、最もパフォーマンスの改善が速かったミニバッチサイズを 選べば良いのです。他のハイパーパラメータの最適化は、ここで選んだミニバッチサイズを使って行います。

もう気づいている読者も居るでしょうが、この本ではここに述べた最適化を行ってはいません。実は、ここで述べたようなミニバッチ学習を高速化する手法を使ってさえいないのです。これまでほとんど全ての具体例で、説明なしに大きさ10のミニバッチを用いてきました。ですから、ミニバッチサイズを小さくすることで学習を高速化することもできたでしょう。そうしなかった理由として、一つはミニバッチ更新のやり方を示したかったこと、そしてもう一つは事前に行った実験からミニバッチサイズを小さくしても大して 学習速度は改善しないと考えられたことが挙げられます。しかし実際の実装では、ほぼ間違いなくミニバッチ更新を高速な行列ライブラリ等を用いて実装し、全体の学習スピードを最大化するようにミニバッチサイズの最適化も行うでしょう。

自動化技術：ここまでは、ハイパーパラメータの最適化を手で行うかのように説明してきました。自分の手を動かして最適化を行うことは、ニューラルネットワークの振る舞いについて 感覚を養うのに良い

方法です。しかし、当然ながら、実際は多くの場合にこの過程は自動化されています。よく使われるのは**グリッド探索**という方法で、ハイパーパラメータの空間を グリッド状に系統的に探索します。グリッド探索の成果と限界(そして簡単に実装できる代替手法)に関するレビューとして、James BergstraとYoshua Bengioによる2012年の論文*を挙げておきます。より洗練された手法も多数提案されています。ここでそれらの全ては紹介しませんが、ベイズ統計に基づくハイパーパラメータの自動最適化を用いた2012年の論文は 特に有望と思われるので紹介しておきます*。この論文のコードは公開されており、他の研究者にも利用されいくつかの成功を収めています。

*Random search for hyper-parameter optimization, by James Bergstra and Yoshua Bengio (2012).

*Practical Bayesian optimization of machine learning algorithms, by Jasper Snoek, Hugo Larochelle, and Ryan Adams.

まとめ: ここで大雑把に説明してきたやり方に従っても、ニューラルネットワークが持つ最大限の性能を 発揮できることが保証されているわけではありません。とは言え、さらなる改善を前に 十分良い出発点に立つことはできるはずです。特に、ここでの説明はハイパーパラメータごとに ほぼ独立して行ってきましたが、実際にはハイパーパラメータ同士の関係も問題になります。例えば、まず η の最適化を行ってから λ の最適化に取り掛かると、η が再び最適からは程遠い状態になってしまう、なんてことも起こりえます。実際上は、両者を行ったり来たりしながら徐々に良い値に近づいていくのが上手いやり方です。そして何より、ここまで説明してきたことはあくまで経験則であって、金科玉条の類ではないことを心に留めておいてください。ここに書いたやり方に従っているときも、常に上手くいっていないところが無いか気を配り、その兆しがあれば自分で実験してみることが大切です。これは、ニューラルネットワークの振る舞い、特に検証精度を注意深くチェックすることを意味します。

さらに厄介なのは、ハイパーパラメータの選び方のコツがあらゆるところに散らばっている点です。たくさんの研究論文やプログラムの中に散らばっていますし、個々の技術者の頭の中にしか無いコツも少なくありません。本当にたくさんの論文が様々な(そして時に互いに矛盾する)方法を勧めています。しかし、中には僅かながら特に役立つ論文があり、雑多な情報の中からエッセンスをまとめてくれています。Yoshua Bengioによる2012年の論文*には、逆伝播法や勾配降下法を用いた (多層)ニューラルネットワークの学習について実践的なアドバイスが記されています。Benigoは、より系統的なハイパーパラメータの決め方など、多くの事項について この本よりもずっと詳細に議論しています。他におすすめしたい論文は、Yann LeCun, Léon Bottou, Genevieve Orr, そしてKlaus-Robert Müllerによる1998年の論文*です。どちらの論文も、ニューラルネットワークで用いられる様々な技法を集めた、非常に有用な2012年の本

*Practical recommendations for gradient-based training of deep architectures, by Yoshua Bengio (2012).

*Efficient BackProp, by Yann LeCun, Léon Bottou, Genevieve Orr and Klaus-Robert Müller (1998)

*に収められています。この本は高価ですが、恐らく出版社のご厚意で、多くの論文は各著者のウェブサイトで公開されており、検索エンジンを使って探し出せます。

*Neural Networks: Tricks of the Trade, edited by Grégoire Montavon, Geneviève Orr, and Klaus-Robert Müller.

これらの文献を読んで、自分自身でも実験を行ってみると、ハイパーパラメータの最適化は 完全解決のない試みであることがはっきりしてくるでしょう。ある手法を試みれば、そのパフォーマンスを改善できる別の手法が常に見つかるものです。作家の間には、執筆は決して終わることは無く、ただ放り出してしまうだけなのだ、という格言があります。ニューラルネットワークの最適化についても同じことが言えます：ハイパーパラメータの空間は あまりに広いので最適化に終わりはなく、どこかで放り出してしまうしか無いのです。ですから、十分に良い最適化を手際よく行う作業工程を確立し、必要なときにだけより詳細な最適化を行う柔軟性を残すことを目指すべきです。

ニューラルネットワークを他の機械学習手法と比較して、ハイパーパラメータを決めることの困難について文句を言う人も居ます。例えば、おおよそ次のような文句はたくさん聞きました：「確かに、良くチューニングされた ニューラルネットワークはこの問題に対して最高のパフォーマンスを発揮するかもしれない。一方で、ランダムフォレストを（あるいはSVMを、あるいは…、何でも好きな手法を入れてください） 試してみることもできる。そしてその手法でも上手く行くだろう。ニューラルネットワークに取り組んでいる時間なんて無いよ。」もちろん、実用上は 簡単に利用できる技術があるのは良いことです。何か新しい問題に取り掛かる時、そして機械学習が役に立つかさえ明らかでは無いような状況では、特にそう言えます。一方で、最適なパフォーマンスを得ることが重要なら、より高度な知識を必要とするアプローチを 試してみることも必要でしょう。機械学習がいつでも簡単に使えるなら嬉しい限りですが、そのように期待すべきア・プリオリな理由もないのです。

その他の手法

この章で開発したさまざまな手法は、それ自身知っておく価値があるものたちばかりです。しかし、私がそれらの手法を説明したのは、単にそれ自体の価値のためだけではありません。ニューラルネットワークを扱う上で起こりうる問題や、それを克服する助けとなる解析のスタイルについて、皆様にも慣れていただく、というのが、より重要な目標でした。ある意味では、私たちはニューラルネットについて考える方法を学習していたのです。この章の残りでは、これまでにカ

バーしきれなかった他の手法を軽く紹介します。これからの紹介は、これまでの議論よりは深みに欠けるものになってしまいますが、ニューラルネットワークを扱う手法の多様さを感じていただければ幸いです。

確率的勾配降下法のバリエーション

逆伝播法による確率的勾配降下法はMNIST手書き数字分類問題に対して十分な働きをしてきました。しかし、コスト関数の最適化については他にも多くのアプローチがあり、時にはそれらの手法がミニバッチを使った確率的勾配降下法よりも優れた性能を発揮する場合もあります。この節ではヘッセ行列法とモメンタム法の2つを概観します。

ヘッセ行列法: この手法について議論する上で、ひとまずニューラルネットワークのことは脇においておきましょう。その代わり、多変数 $w = w_1, w_2, \ldots$ のあるコスト関数 $C(w)$ を最小化するという抽象的な問題を考えましょう。テイラーの定理により、コスト関数は点 w の周りで

$$C(w+\Delta w) = C(w) + \sum_j \frac{\partial C}{\partial w_j}\Delta w_j + \frac{1}{2}\sum_{jk}\Delta w_j \frac{\partial^2 C}{\partial w_j \partial w_k}\Delta w_k + \ldots \tag{96}$$

と近似できます。これをよりコンパクトに、

$$C(w+\Delta w) = C(w) + \nabla C \cdot \Delta w + \frac{1}{2}\Delta w^T H \Delta w + \ldots, \tag{97}$$

と書き換えることもできます。ここで、∇C はコスト関数の勾配、H は**ヘッセ行列**と呼ばれる行列で、その j 行 k 列の要素は $\partial^2 C/\partial w_j \partial w_k$ です。上で省略した高次の項を無視すると、次の近似式が得られます:

$$C(w+\Delta w) \approx C(w) + \nabla C \cdot \Delta w + \frac{1}{2}\Delta w^T H \Delta w. \tag{98}$$

簡単な計算により、この右辺を最小化するには*

$$\Delta w = -H^{-1}\nabla C. \tag{99}$$

と選べば良いことが分かります。式 (98) が 良い近似であるとすると、点 w から $w+\Delta w = w - H^{-1}\nabla C$ へ移動することで コスト関数を大きく減少させられると期待できます。このことから、次のようなコスト関数の最小化アルゴリズムが考えられます:

*厳密に言うと、これが極小ではなく最小であるためには、ヘッセ行列が正定値である必要があります。直感的には、C が局所的に山や鞍点ではなく谷になっていることを意味します。

- 出発点 w を選びます。
- 点 w を新しい点 $w' = w - H^{-1}\nabla C$ に更新します。ここで、∇C と ヘッセ行列 H は点 w で計算します。
- 点 w' を新しい点 $w'' = w' - H'^{-1}\nabla' C$ に更新します。ここで、ヘッセ行列 H' と $\nabla' C$ は点 w' で計算します。
- ...

実際には、式 (98) はあくまで近似であり、更新はもっと小さい幅で行うべきです。そのためには、**学習率** η を導入し、w の変化量を $\Delta w = -\eta H^{-1}\nabla C$ にすれば良いです。

コスト関数を最小化するこのアプローチは **ヘッセ行列法** あるいは **ヘッセ行列最適化** と呼ばれます。ヘッセ行列法が標準的な勾配降下法と比べて少ないステップ数で収束することが 理論的・経験的に知られています。特に、コスト関数の2次の変化を取り入れることで、勾配降下法について知られている多くの病的な振る舞いを避けられることが知られています。さらに、ヘッセ行列を計算するために拡張された逆伝播法も知られています。

ヘッセ行列最適化がそんなに素晴らしいのなら、なぜこれまで見てきたニューラルネットワークで 使っていないのでしょうか？確かにヘッセ行列法は多くの良い性質を持ってはいるものの、一つ重要な、非常に好ましくない性質も持っています：ヘッセ行列は非常に大きいのです。例えば、重みとバイアスを合わせて 10^7 個のパラメータを持つニューラルネットワークを 訓練するとします。すると、そのヘッセ行列は $10^7 \times 10^7 = 10^{14}$ 個もの要素を持ちます。この大量にある要素を計算しなければならないのみならず、逆行列まで計算しなければならないのです！ このため、ヘッセ行列法を実際に適用するのは難しいのです。しかし、この方法には理解しやすいというメリットもあります。実は、勾配降下法の亜種には、ヘッセ行列法に触発された、しかし巨大な行列の問題を伴わない手法が 数多く有ります。次に、そのような手法の一つ、モメンタム勾配降下法を見てみましょう。

モメンタム勾配降下法： 直感的に、ヘッセ行列法の強みは勾配の情報に加えて勾配の変化に関する情報を 取り入れられる点にあると言えます。モメンタム勾配降下法は同様の直感に基づいていますが、2階微分の巨大な行列を取り扱うことを巧妙に回避しています。モメンタム法を理解するために、以前に示した勾配降下法の 直感的な描像に戻って、谷を転がり落ちるボールを考えてみましょう。勾

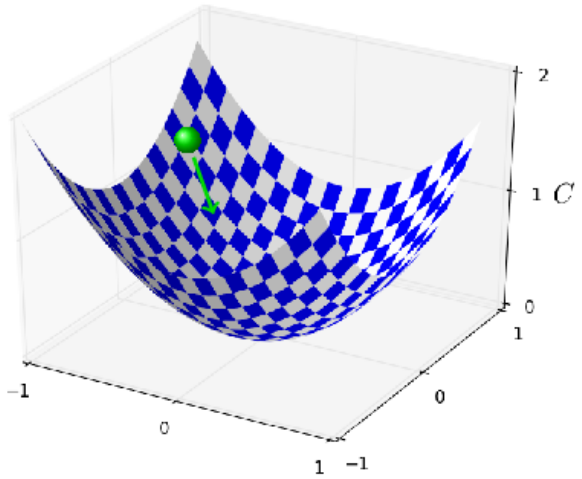
配降下法はその名前の割に、実際にはそれほど谷底に落ちていくボールと似ていないことは既に指摘しました。モメンタム法では、2つの点で勾配加工法を修正し、より物理的な描像と近づけています。第一に、この手法では最適化するパラメータに「速度」の概念を導入します。勾配はこの速度を変化させますが、「位置」を(直接的には)変化させません。これは、物理的な「力」は速度を変化させますが、位置の変化は速度の変化を介して間接的にしか 引き起こさないことと似ています。第2に、モメンタム法ではある種の摩擦項を導入し、自然と徐々に減速するように仕向けます。

より精密で数学的な説明をしましょう。それぞれのパラメータ変数 w_j に対して、速度変数 $v = v_1, v_2, \ldots$ を導入します*。そして、勾配降下法におけるパラメータの更新規則を $w \rightarrow w' = w - \eta \nabla C$ から

*もちろん、ニューラルネットワークのw_j 変数は全ての重みとバイアスを含みます。

$$v \rightarrow v' = \mu v - \eta \nabla C \tag{100}$$
$$w \rightarrow w' = w + v' \tag{101}$$

に置き換えます。ここで、μ は系の減衰、あるいは摩擦の大きさを決めるハイパーパラメータです。この方程式の意味を理解するには、まず摩擦が無い場合に相当する $\mu = 1$ を考えてみるのが良いでしょう。その場合、∇C が「力」として速度 v を変化させること、そして速度が w の変化率を与えていることが見て取れます。直感的には、勾配項を繰り返し足し上げることで速度が生じます。学習の数ステップに渡って勾配が(大まかに)同じ方向ならば、その方向にかなりの勢いがつきます。例えば、下り坂を真っ直ぐ下る時に何が起こるか考えてみてください:



ステップ毎にスロープを下る速度が増していき、加速しながら谷底まで素早く移動していきます。これがモメンタム法が通常の勾配降下

法よりも高速に学習する仕組みです。もちろん、このままでは谷底を通り過ぎてしまう問題が有ります。また、勾配が素早く変化する場合には、間違った方向に進んでしまうこともあるでしょう。この問題を解決するために、μ というハイパーパラメータを式 (100) に導入したのです。既に、μ が摩擦を表すハイパーパラメータであるという説明をしました。より正確に言えば、$1-\mu$ が摩擦の大きさを表しています。既に見たように、$\mu=1$ の時には摩擦が無く、∇C が加速度の役割をして速度を変化させます。反対に、$\mu=0$ の場合には、摩擦が大きくて加速がつかず、式 (100) と (101) は元の勾配下降法 $w \rightarrow w'=w-\eta \nabla C$ に帰着します。実用的には、μ を0と1の間に取ることで、谷底を行き過ぎること無く、学習を加速することが可能になります。検証データを用いると、η や λ を決定したのと同じように μ の値を決定できます。

ここまで、ハイパーパラメータ μ に名前を付けるのを避けてきました。その理由は、μ の標準的な名前があまりに酷いからです：**モメンタム係数** と呼ばれています。この名前は混乱の元になると思います。というのも、μ は物理におけるモメンタムの概念とは全く何の関係も無いからです。むしろ、摩擦と非常に関係があります。しかし、モメンタム係数という言葉は広く使われているので、今後はこの言葉を使うことにします。

モメンタム法の優れた点は、元の勾配降下法の実装を修正してモメンタムの効果を取り入れるのが 非常に容易であることです。以前と全く同じように逆伝播法を使って勾配を計算できますし、ミニバッチを使った確率的サンプリングの考えも同じように適用できます。このように、勾配の変化の情報を使うことでヘッセ行列法の利点を部分的に活用できます。しかも、ほとんどデメリットが無く、少しだけ実装を修正するだけで良いのです。実際の現場でもモメンタム法は広く利用され、学習の加速に貢献しています。

演習

- もしモメンタム法で $\mu>1$ とするとどのような問題が起こるでしょうか？
- もし $\mu<0$ とするとどのような問題が起こるでしょうか？

問題

- モメンタム法に基づいた確率的勾配降下法を `network2.py` に追加してください。

コスト関数を最小化するその他の手法: コスト関数を最小化する手法はこの他にも多数開発されてきましたが、どれが最適な手法かという問いに対しては一致した回答がありません。ニューラルネットワークを使い込んでいく中で、他の手法に取り組み、その挙動や利点・欠点を理解することは役に立つでしょう。前に示した論文*では、共役勾配降下法やBFGS法(これと非常に関係の深い、L-BFGS法と呼ばれる 記憶制限BFGS法も重要です)を含め、幾つかの方法が紹介・比較されています。他に最近優れた結果*を出した手法としては、モメンタム法を修正したNesterovによる加速勾配法があります。しかし、多くの問題に対して、単純な確率的勾配降下法も、特にモメンタム法と組み合わせた時には 良い性能を発揮します。ですから、この本の残りの部分では確率的勾配降下法を使うことにします。

*Efficient BackProp, by Yann LeCun, Léon Bottou, Genevieve Orr and Klaus-Robert Müller (1998).

*例えば、次の論文を参照してください。On the importance of initialization and momentum in deep learning, by Ilya Sutskever, James Martens, George Dahl, and Geoffrey Hinton (2012).

人工ニューロンのその他のモデル

ここまでは、シグモイドニューロンを使ってニューラルネットワークを組み立ててきました。原理的には、シグモイドニューロンから成るニューラルネットワークは任意の関数を計算できます。しかし現実的には、他のニューロンモデルを使ったニューラルネットワークが シグモイドニューラルネットワークよりも高性能になる場合もあります。応用先によっては、そのような別のモデルに基づいたニューラルネットワークの方が 速く学習したり試験データに対してより良い汎化性能を発揮したりします。ここでは、シグモイドニューロン以外のモデルを幾つか紹介します。

恐らく最も単純なのは、tanhニューロンでしょう。これは、ニューロンのシグモイド関数を 双曲線正接関数に置き換えたものです。入力 x、重み w、バイアス b の時、tanhニューロンの出力は

$$\tanh(w \cdot x + b) \tag{102}$$

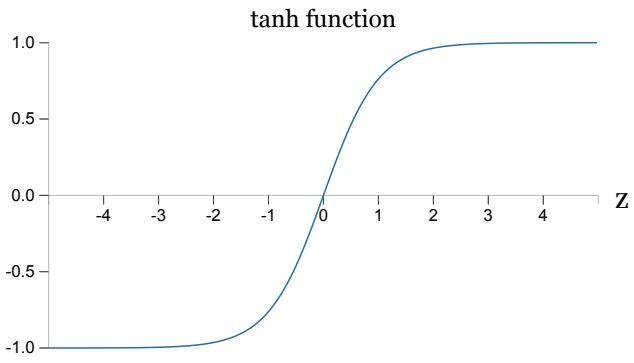
で与えられます。ここで、tanh はもちろん双曲線正接関数です。実は、シグモイドニューロンとは非常に関係しています。tanh 関数の定義

$$\tanh(z) \equiv \frac{e^z - e^{-z}}{e^z + e^{-z}}. \tag{103}$$

を思い出すと、簡単な計算で

$$\sigma(z) = \frac{1 + \tanh(z/2)}{2} \tag{104}$$

という関係が示せます。つまり、tanh はシグモイド関数をリスケールしただけなのです。グラフを描いてみても、tanh 関数のグラフがシグモイド関数と同じ形をしていることが分かります。



tanhニューロンとシグモイドニューロンの違いの一つは、tanhニューロンの出力の値域が -1から1である点です。このため、tanhニューロンを使ってニューラルネットワークを構築する際には、シグモイドニューロンを使った時とは少し違った方法で出力を規格化する必要があります。この規格化の選択は、場合によっては解くべき問題の詳細や入力にも依存するでしょう。

シグモイドニューロンの場合と同様に、tanhニューロンのネットワークも原理的には、入力を-1から1に写す任意の関数を計算できます*。さらに、逆伝播法や確率的勾配降下法を tanhニューロンのネットワークに移植することも容易に可能です。

*細かいことを言えば、どのような種類のニューロンを用いるにせよ、この主張を正しく理解するためには少々技術的な注意が必要です。しかし、大雑把に言えばニューラルネットワークが任意の関数を任意の精度で近似できると考えて差し支えありません。

演習

- 式 (104) の等号 を示してください。

tanhとシグモイド、どちらのニューロンを使うべきなのでしょうか？穏やかに言えば、**ア・プリオリ**には明らかな答えはありません。しかし、時としてtanhの方が 高い性能を発揮できることを示唆する理論的な議論や経験的な証拠があります *。理論的な議論とはどういうものか感じてもらうために、その内容を手短に紹介します。シグモイドニューロンを使っていて、ニューラルネットワーク中の全ての活性が正の値を取るとしましょう。そして、第$l+1$層のj番目のニューロンに入力する重み w^{l+1}_{jk} を考えましょう。逆伝播法（こちら）によると、この重みに関する勾配は $a^l_k \delta^{l+1}_j$ と書けます。活性は全て正なので、この勾配の符号は δ^{l+1}_j の符号と一致します。このことは、もし δ^{l+1}_j が正ならば、**全ての**重み w^{l+1}_{jk} は 勾配降下法で減少し、反対に δ^{l+1}_j が負ならば**全ての** w^{l+1}_{jk} が減少することとなります。言い換

*参考文献としては、例えば、Efficient BackProp, by Yann LeCun, Léon Bottou, Genevieve Orr and Klaus-Robert Müller (1998), そして Understanding the difficulty of training deep feedforward networks, by Xavier Glorot and Yoshua Bengio (2010) をご覧ください。

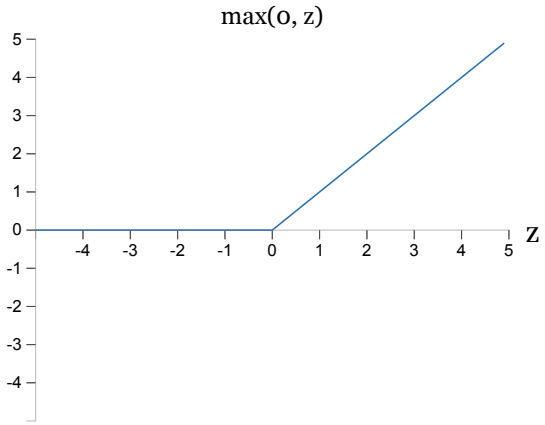
えると、同じニューロンに入力する全ての重みは一緒に増加するか一緒に減少するかしか できないのです。重みの一部だけ増加して残りの重みが減少するといった変化も 必要かもしれませんから、この性質は問題です。このことから、シグモイド関数を tanh のような正の値も負の値も取りうる活性関数で置き換えた方が良いように 思われます。実際、tanh は奇関数ですから、大雑把に言って隠れ層の活性は 正と負の両方にバランス良く分布するでしょう。それによって、重みの更新に系統的なバイアスが掛かるのが避けられるでしょう。

この議論はどの程度真面目に考えるべきでしょうか？この議論は示唆的ですが ヒューリスティックですし、tanhニューロンがシグモイドニューロンより高性能である ことの厳密な証明にはなっていません。ここで説明した問題を補うような良い性質を、シグモイドニューロンが備えているのでしょうか。実際のところ多くの課題に対して、tanhニューロンはシグモイドニューロンと比べて良くても僅かな改善しかしないことが 経験的に知られています。残念ながら、いかなる問題に対しても、学習を最速化する、あるいは汎化性能を最大化するニューロンの種類を決定する 規則は存在しないのです。

シグモイドニューロンのもう一つの変種は、**Rectified Linear Unit** あるいは **ReLU** と呼ばれるものです。入力 x, 重みベクトル w, そして バイアス b が与えられた時、ReLUの出力は

$$\max(0, w \cdot x + b). \tag{105}$$

と書けます。グラフにすると、次のようになります：



明らかにシグモイドともtanhとも相当異なる形をしています。しかし、シグモイドやtanhと同様、ReLUを使ったニューラルネットワークは

任意の関数を計算可能であることが知られており、逆伝播法や確率的勾配降下法を用いて 訓練できます。

シグモイドやtanhニューロンよりもReLUが力を発揮できるのは どのような場面でしょうか？画像認識に関する最近の研究によると* 多くのニューラルネットワークでReLUの大きなご利益が得られるそうです。しかし、tanhニューロンの場合もそうでしたが、ReLUをいつ、そしてなぜ 使うべきなのか、という問いに対しては、未だに深い理解は得られていません。幾つかの論点を概観しておきましょう。まず、シグモイドニューロンの学習が遅くなるのは、ニューロンが飽和している時、すなわち、ニューロンの出力が 0 か 1 に近いときです。この章で何度も見てきたように、σ' という因子が勾配を小さくしてしまうことで 学習が遅くなるのでした。tanhニューロンも飽和してしまうと同様の問題にぶつかります。反対に、ReLUは重み付き入力値を大きくしても飽和することがないでしょう。そのため、飽和に伴った学習速度の低下を起こさないのです。一方で、ReLUへの 入力が負の値を取るときには、勾配がゼロになります。そのため、ニューロンは学習を 完全にやめてしまうでしょう。この2つは、いつ、なぜ、ReLUが優れているのかという問題の理解を妨げているたくさんある要因の一部に過ぎません。

ここまで私は、活性関数の選び方に関する確固とした理論が存在しないことを 強調してきました。しかし現実は、ここまで説明してきたよりもさらに困難です。というのも、活性関数の選択肢は無限にあるのですから。与えられた問題に対する最良の選択はどれか？どれが最高の試験精度を達成するか？ どれが学習を最速で行うか？これらの問いに対して、真に深く系統的な 研究が行われていないことに私は驚いています。理想的には、活性関数の選び方を 教えてくれる理論があってもおかしくないはずです。とはいえ、完全な理論が無いからといって、私たちが立ち止まる必要はありません！ 既に強力なツールはたくさんあるので、それらを使えば多くのことを成せるはずなのですから。この本の残りの部分では、シグモイドニューロンを使い続ける ことにします。シグモイドニューロンも十分に強力ですし、ニューラルネットワークの 中核を成すアイデアを具体的に示すには良い道具だからです。しかし、同じアイデアはシグモイドニューロンに限らず他の種類のニューロンにも 当てはまることは覚えておいてください。そして時には他のニューロンの方が 高い性能を発揮する場合もあることを心に留めておいてください。

*関連する文献としては、Kevin Jarrett, Koray Kavukcuoglu, Marc'Aurelio Ranzato, Yann LeCunによる What is the Best Multi-Stage Architecture for Object Recognition? (2009), Xavier Glorot, Antoine Bordes, Yoshua Bengio による Deep Sparse Rectifier Neural Networks (2011), そして、Alex Krizhevsky, Ilya Sutskever, Geoffrey Hinton による ImageNet Classification with Deep Convolutional Neural Networks (2012) 等があります。これらの文献では、ReLUを用いたニューラルネットワークで 出力層、コスト関数、正規化等をどう設計すべきかという重要な点について詳細な議論を 行っています。ここで行う簡単な説明では、そのような詳細は省略します。また、これらの論文では、ReLUの長所も短所も詳細に議論しています。その他に有益な論文としては、Vinod Nair と Geoffrey Hinton による Rectified Linear Units Improve Restricted Boltzmann Machines (2010) があります。この論文では、ニューラルネットワークに対する少々違ったアプローチにおいて正規化線形 ユニットを使うご利益を議論しています。

ニューラルネットワークに関する「お話」について

質問： 機械学習の手法は数学的にほとんど理解されておらず、 それが上手くいくかどうかは経験的な裏付けしかありません。 これらの手法を利用したり研究したりしたい場合、どうやって取り掛かれば良いのですか？ また、これら機械学習の手法が破綻するのはどのような状況ですか？

回答: まず、手元にある理論的な道具はとても貧弱であることを認める必要があります。時には、ある特定の手法が効果的に働くであろうという、数学的な直感に基づく予測が可能な場合もあります。また時には、私たちの直感が誤りだと後になって気づくこともあります。（中略）質問はこう言い換えるべきでしょう: この手法はこの特定の問題に対してどのくらい効果的ですか？ そして、この手法が上手く働く問題はどのくらいたくさんありますか？

- Question and answer, ニューラルネットワーク研究者 Yann LeCun

かつて量子力学の基礎付けに関する会議に出席した時、発言者達が非常に興味深い習慣を 持っていることに気づきました。発表が終わると、聴衆からの質問がしばしば「あなたの視点にはとても共感しました。しかし、...」から始まるのです。 量子力学の基礎付けは、私が普段研究している分野ではありません。この質問のしかたに気づいたのは、この分野以外の研究会で発表者の視点に共感を表現する 質問者を見かけることは皆無だからです。当時は、この種の質問が蔓延るのは、この分野がほとんど進展していなくて、皆が無駄な努力をしていることの現れだろうと 考えていました。後になって、この評価は厳しすぎたことに気づきました。 発表者は人類の頭脳が向き合った中で最も難しい問題の一つに取り組んでいたのです。 進展が遅いのは当たり前です！そして、はっきりと新しい進展が無かったとしても、 人々がどう考えているのかについて最新の話を聞くことには価値があったのです。

この本を読んでいて、読者の皆さんは「とても共感しました」と同種の決まり文句に 気づいたかもしれません。何かを説明しようとする度に、「ヒューリスティックに」や「大雑把に言って」等と口にした後、何らかの現象等を説明するお話を続けました。これらのお話はもっともらしいですが、経験的な証拠としては非常に心許ないものです。ニューラルネットワークの研究に関する文献等を調べてみると、多くの論文に 同様の体裁を取ったお話が見られることでしょう。それも、多

くの場合証拠としてはとても弱いものです。この状況はどう理解するべきでしょうか？

科学の多くの分野、特に単純な現象を扱う分野で、非常に一般的な仮説に対する、非常に強固で非常に信頼度の高い証拠が得られます。しかし、ニューラルネットワークの場合、多数のパラメータやハイパーパラメータが存在し、それらが極度に複雑に絡み合っています。このように非常に複雑な系では、信頼できる一般的な議論を行うことは非常に困難です。ニューラルネットワークを完全に一般的に理解することは、量子力学の基礎付けのように、人類の頭脳の限界に挑戦する課題です。一般論を展開する代わりに、その具体例を幾つか取り上げて、一般的な主張がもっともらしいかどうかの議論を行うのです。そして時には、後の検証で修正されたり破棄されたりする結果となるのです。

ヒューリスティックに頼るという事実が問題の困難さを示していると見ることもできます。例えば、ドロップアウト法の説明で引用した文章を思い出してみましょう*:「この手法はニューロン間の複雑な相互適合を軽減します。というのも、訓練に際してニューロンは特定の他のニューロンを 頼りにすることができないからです。したがって、あるニューロンは、ランダムに選ばれた他のニューロンと一緒にされても役に立つような、データが持つより強固な特徴を学ぶように強制されるのです。」この主張は内容が豊富で刺激的です。この主張を一つ一つ検証していくこと、すなわち、この主張のどこが正しくどこが誤っていて、どのような変更や修正が必要なのか 明らかにしていくことで、立派な研究ができるでしょう。 実際、ドロップアウトやその変種について、それが上手くいくメカニズムや限界を調べている 研究者の一団がいます。彼らは、この本でここまで議論してきたヒューリスティックに 取り組んでいるのです。それぞれのヒューリスティックは(潜在的に)ある現象を 説明してくれるだけでなく、それ自体がより詳しく調べて理解すべき課題となるのです。

*出典は Alex Krizhevsky, Ilya Sutskever, Geoffrey Hinton による ImageNet Classification with Deep Convolutional Neural Networks (2012) です。

もちろん、一人の人間がこれら全てのヒューリスティックな議論を詳しく調べるには、全く時間が足りません。ニューラルネットワークの学習についてエビデンスに基づいた強力な 理論を構築するには、これから何十年も(あるいはもっと)時間が掛かるでしょう。 では、私たちはヒューリスティックな説明を根拠に欠ける曖昧なものとして捨て去るべきだと いうのでしょうか？そんなことはありません！むしろ、そのようなヒューリスティックには、私たちの思考を刺激し導くという重要な役割があるのです。これは大航海時代のようなものです:初期の探検家たちは、時に完全な誤解に基づいて探検し、そして新し

い発見をしてきたのです。それらの誤解は、後になって地理に関する知識が 蓄積されるにつれ、修正されてきました。ニューラルネットワークに対する理解が今日非常に 貧弱であるように、何かをほとんど理解できていない時こそ、一歩ずつ完全に正しい思考を 進めるよりも、大胆に探検していくことが重要なのです。ですから、この本でしてきた ヒューリスティックな議論は、ニューラルネットワークについて考える際の便利なガイドである と考えるべきです。ただし、それらの話に限界があることも認識し、どの程度強く裏付けられているのかを意識しておく必要があります。 言い換えると、ニューラルネットワークを使っていく上で発想の助けになるお話と、実際に起こっていることを明らかにする厳密で詳細な検討が必要なのです。

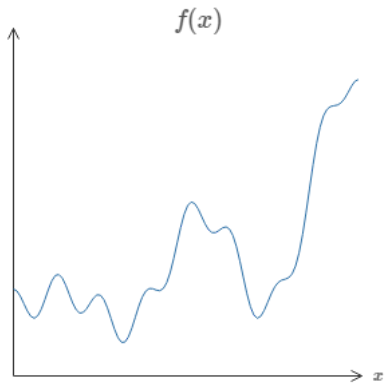
In academic work, please cite this book as: Michael A. Nielsen, "Neural Networks and Deep Learning", Determination Press, 2014

Last update: Tue Sep 2 09:19:44 2014

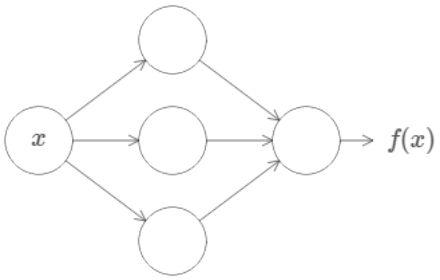
CHAPTER 4

ニューラルネットワークが任意の関数を表現できることの視覚的証明

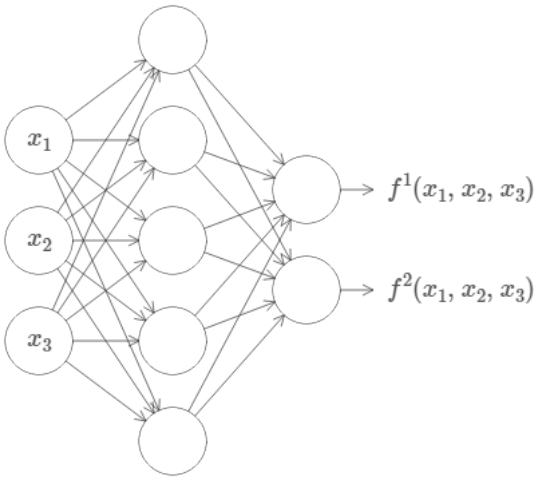
ニューラルネットワークに関して最も衝撃的な事実の1つは任意の関数を表現できることです。例えば誰かから複雑で波打った関数$f(x)$を与えられたとします:



それがどんな関数であっても、考えられるすべての入力xに対して、出力値が$f(x)$(もしくはその近似)であるニューラルネットワークが存在します。例えば下図のようなものです。



この結果は入力が複数の関数$f = f(x_1, \ldots, x_m)$や出力が複数の関数でも成立します。例えば、下図は$m = 3$個の入力と$n = 2$個の出力を持つ関数を計算するニューラルネットワークです:



この結果はニューラルネットワークが一種の**普遍性**を持っている事を示しています。計算したい関数が何であろうとも、その計算を行えるニューラ

Sponsors

この本は目下のところベータ版で、開発続行中です。エラーレポートは mn@michaelnielsen.org まで、日本語版に関する質問は muranushi@gmail.com までお送りください。その他の質問については、まずはFAQをごらんください。

Resources

Code repository

Mailing list for book announcements

Michael Nielsen's project announcement mailing list

著:Michael Nielsen / 2014年9月-12月

ルネットワークが存在することがわかっているのです。

訳:「ニューラルネットワークと深層学習」翻訳プロジェクト

しかも、この普遍性定理は入力層と出力層の間の中間層、いわゆる隠れ層をたった1層しか持たないニューラルネットワークに限っても成立しています。つまり、極めて単純なネットワーク構成でも表現力は極めて高いのです。

普遍性定理はニューラルネットワークを扱う人々の間ではよく知られています。しかし、なぜそれが正しいのかはそれほど広くは理解されていません。よく見られる説明の多くは極めてテクニカルです。例えば、この結果を証明している原著論文* では、ハーン・バナッハの定理とリースの表現定理とフーリエ解析を利用しています。この論文の議論を追うのは数学者にとっては難しくないかもしれませんが、大多数の人にとっては簡単ではありません。これは悲しいことです。なぜなら、ニューラルネットワークの普遍性を成り立たせている原因は、本当はシンプルで美しいものだからです。

*Approximation by superpositions of a sigmoidal function, George Cybenko (1989). この結果は広く知られ、他にもいくつかのグループが関連した結果を証明しました。Cybenkoの論文ではその仕事に関して多くの有益な議論がなされています。初期の論文でその他の重要なものとして、Multilayer feedforward networks are universal approximators, Kurt Hornik, Maxwell Stinchcombe, Halbert White (1989)があります。この論文はストーン・ワイエルシュトラスの定理を用いて類似の結果を得ています。

本章では、普遍性定理のシンプルで大部分が視覚的な説明を行います。背景にあるアイデア達を1つずつ順を追って見ていきます。なぜニューラルネットワークが任意の関数を表現できるのかの理由、結果にある種の制限がついている事、そしてこの結果と深いニューラルネットとの関連が理解できるようになるはずです。

本章を読むのにこの本のこれ以前の章を読む必要はありません。その代わりに自己完結的なエッセイとして楽しめるよう構成されています。ニューラルネットワークについて少し慣れていれば、説明を追えるはずです。ただ、知識のギャップを埋めるのに役立つよう、以前の章へのリンクは必要に応じて示すつもりです。

普遍性に関する定理はコンピュータ科学では珍しくなく、それらが驚くべき定理である事をしばしば忘れてしまいます。しかし、任意の関数を計算できるのは本当に著しい性質であることは今一度思い起こす価値のあることです。あなたが思いつく処理は大抵どれも関数の計算と思うことができます。例えば、音楽の短いサンプルだけを聞いて曲名を当てる問題を考えてみてください。これも、一種の関数の構成であると考える事ができます。または、中国語の文章を英語に翻訳する問題を考えてみてください。やはり、これも関数の構成だと考える事ができます*。もしくは、mp4形式の映画ファイルからその映画の物語のプロットを作成する問題を考えてみてください。これも、一種の関数構成と考える事ができます*。普遍性定理は、ニューラルネットワークがこれらやそれ以外の様々な処理を原理的にはできることを示しています。

実際には1つの文章にはたくさんの妥当な訳し方があるので、多くの考えられる関数のうちの1つを構成している事になります。

先程の翻訳の場合と同様ですが、妥当な関数として様々なものが考えられます。

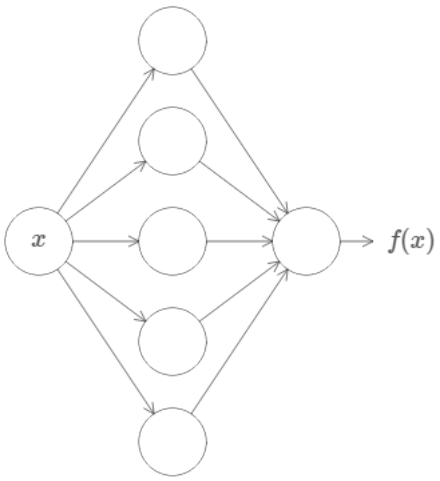
もちろん、例えば中国語の文章を英語に翻訳するニューラルネットワークの存在が分かることは、そのようなニューラルネットワークを構成したり、ネットワークがその性質を持つか判定する良い方法がわかることを意味しません。ブーリアン回路のようなモデルに対する古典的な普遍性定理にもこの制限は適用されます。しかし、この本の前の章で見てきたように、ニューラルネットワークには関数を学習する強力なアルゴリズムがありま

す。学習アルゴリズムと普遍性の組み合わせは魅力的です。ここまで、この本では学習アルゴリズムに焦点を置いてきました。本章では、普遍性とそれが意味する所に焦点を置きます。

2つの注意点

普遍性定理が何故正しいかを説明する前に、「ニューラルネットワークが任意の関数を計算できる」という砕けた表現について2つの注意点を挙げたいと思います。

まず、この表現はニューラルネットワークは任意の関数を**完全に**計算できる事を意味するのではありません。そうではなく、好きなだけ近い**近似**関数を得られるという意味です。隠れ層のニューロンを増やすことで、近似の精度を上げることができます。例えば、前にある関数$f(x)$を3つの隠れニューロンを用いて計算するニューラルネットワークを説明しました。大抵の関数については3個の隠れニューロンでは、精度の低い近似しか得られません。隠れニューロンの数を(例えば5個に)増やす事で、より良い近似が得られます。



そして、隠れニューロンをさらに増やす事で、さらに近似を良くできます。

ステートメントをより正確にする為に、私達が計算したい関数$f(x)$と希望の精度$\epsilon > 0$が与えられたとします。十分な数の隠れニューロンを用いる事で、出力$g(x)$が$|g(x) - f(x) < \epsilon|$を任意の入力xに対して満たすニューラルネットワークを常に見つけられることを普遍性定理は保証しています。言い換えれば、希望の精度の範囲内で考えられるすべての入力に対して良い近似であることを示しているのです。

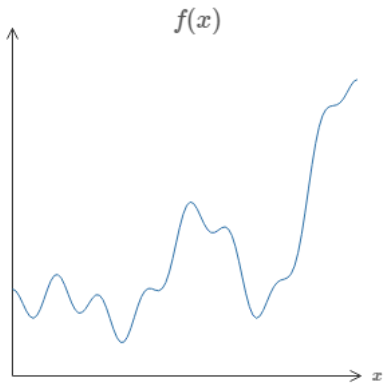
2つ目の注意点は、この方法で近似できる関数のクラスは**連続**関数であるという点です。もし関数が不連続、すなわち急激なジャンプが突然発生する場合、ニューラルネットワークを用いた近似は一般的には不可能です。これは驚くべき事ではありません。というのも、私達のニューラルネットワークが計算できるのは入力に対して連続な関数だからです。しかし、計算したい関数が不連続でも、連続関数による近似で十分な場合も

あります。その場合には、ニューラルネットを利用できます。通常はこの制限は重大なものではありません。

まとめると、普遍性定理のより正確なステートメントは、「隠れ層を1つ持つニューラルネットワークを用いて任意の連続関数を任意の精度で近似できる」となります。本章では隠れ層が1層ではなく2層の場合のもう少し弱いバージョンの定理を証明します。少し証明をひねる事で本章での証明を隠れ層が1層しかない場合に適用する方法を、演習問題において簡単に説明します。

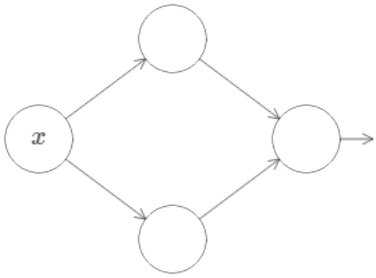
入出力が1つの場合の普遍性定理

普遍性定理の正しさを理解するために、まずは1つの入力と1つの出力を持つ関数を近似するニューラルネットワークの構成方法を理解する所から始めましょう:

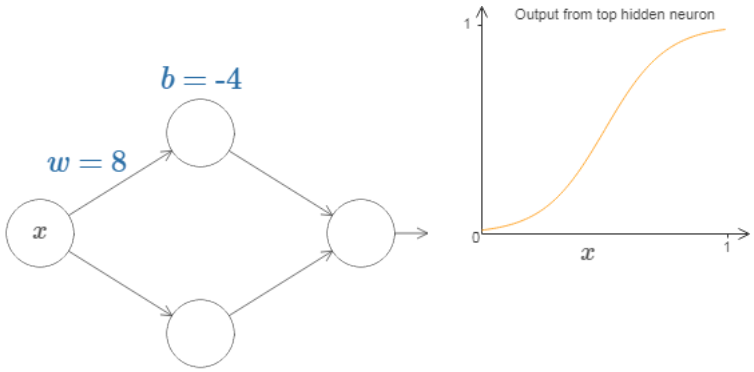


この場合が普遍性の問題の中核をなします。この特別な場合を理解すれば、入出力が多数の場合に拡張するのは容易です。

fを計算するニューラルネットワークの構成方法について直感を養うために、隠れニューロンを2つ持つ隠れ層を1層持ち、出力層がニューロンを1つ持つ場合を考えます。



ニューラルネットワークを構成する各要素の挙動について感覚をつかむ為に、上の隠れニューロンに注目してみましょう。下図で重みwの値をクリックしマウスを右に少しドラッグするとwの値が増加します。それに応じて上の隠れニューロンの出力関数の変化する様子がわかります。



この本の前の方で学習したように、隠れニューロンで計算しているのは$\sigma(wx+b)$です。ここで$\sigma(z) \equiv 1/(1+e^{-z})$はシグモイド関数です。これまで、このような数式による表現を頻繁に使用してきました。しかし、普遍性定理の証明においてはこの計算式を完全に忘れて、グラフの形を操作・観察する方がより洞察を得ることができます。このようにすることで、単に何が起こっているかを感覚的に掴めるだけではなく、シグモイド関数以外の活性化関数に適用する普遍性定理の証明 *も得られます。

*厳密に言えば、私が取る視覚的なアプローチは伝統的には証明と考えられているものではありません。しかし、視覚的なアプローチはなぜこの結果が正しいのかについて、伝統的な証明よりもより多くの洞察が得られると信じています。そしてもちろん、この種の洞察こそが証明の背後にある真の意図なのです。私が示す推論の中にはいくつか小さなギャップが存在します：つまり視覚的な議論を行っていて妥当だけれど厳密ではない部分です。もしこのことが気になるようでしたら、欠けている行間を埋める事に挑戦してみてください。しかし、なぜ普遍性定理が正しいのかを理解するという真の意図を見失わないようにしてください。

証明を始める前に、上図のバイアスbをクリックして右にドラッグすることで値を増加させてみてください。バイアスが大きくなるに従いグラフは左に移動しますが、形は変化しないことがわかります。

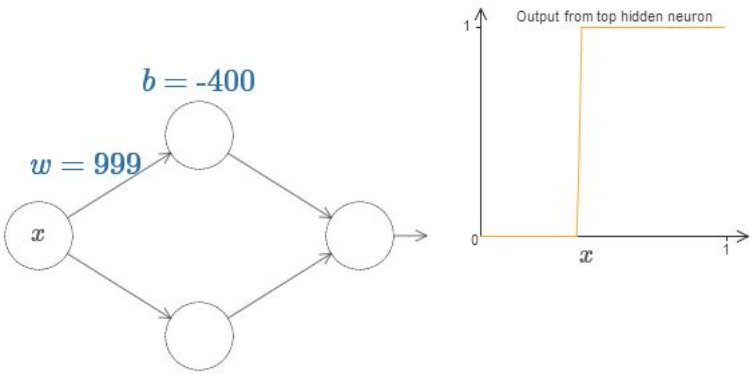
次にクリック・左ドラッグをしてバイアスを減らしてみてください。バイアスが減るにつれてグラフが右に移動しますが、やはり形は変化しない事がわかります。

次に、重みを2か3程度まで減らしてみてください。重みを減らすにつれて、曲線が広がっていくのがわかります。曲線をフレーム内に収めるために、バイアスも変える必要があるかもしれません。

最後に重みを$w = 100$過ぎまで増やしてみてください。増加させるにつれて曲線の勾配がきつくなり、最終的にステップ関数のような形になります。段差が$x = 0.3$あたりに来るようにバイアスを調節してみてください。下のクリップは想定する挙動を示しています。再生ボタンを押すとビデオが再生（もしくはリプレイ）されます。

重みを増加させ、出力をステップ関数に十分近づけることで、解析を著しく単純にすることができます。下では、重みが$w = 999$の時の上の隠れ

ニューロンの出力を図示しています。この図は静的で、重みなどのパラメータを変化できない事に注意してください。

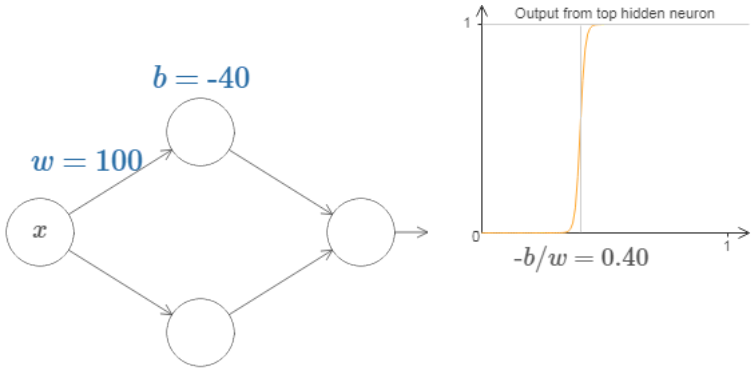


一般のシグモイド関数に比べてステップ関数で考える方が簡単です。その理由は出力層がすべての隠れニューロンからの寄与を足しあわせるからです。ステップ関数達の和を解析するのは簡単ですが、シグモイドの形をした曲線達を足しあわせた時に何が起こるのかを解析するのはそれに比べるとずっと難しいです。ですので、隠れニューロンがステップ関数を出力していると仮定することでずっと簡単になります。具体的には、重みを適当なとても大きな値に固定した後にバイアスを変化させて段差の位置を調整する事でこれを実現できます。もちろん出力をステップ関数として扱うのは近似です。しかしこれは十分良い近似になっているので、しばらくは厳密にステップ関数であるとして扱います。後でこの部分に戻ってきて、この近似によるずれの影響を議論します。

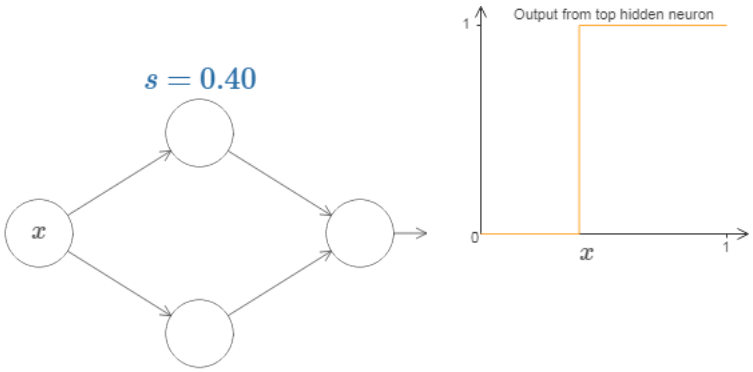
ステップがあるのはxの値でいえばどこでしょうか？ 言い換えると、段差の位置は重みや階段にどのように依存するでしょうか？

この答えに答えるために、上図の重みやバイアスを変化させてみてください（少しスクロールする必要があるかもしれません）。段差の位置がwやbにどのように依存するかがわかりますか。少し試してみると、段差の位置はbに**比例し、wに反比例している事がわかると思います。**

下図の重みとバイアスを変化させるとわかりますが、実は段差は$s = -b/w$の部分に生じます：

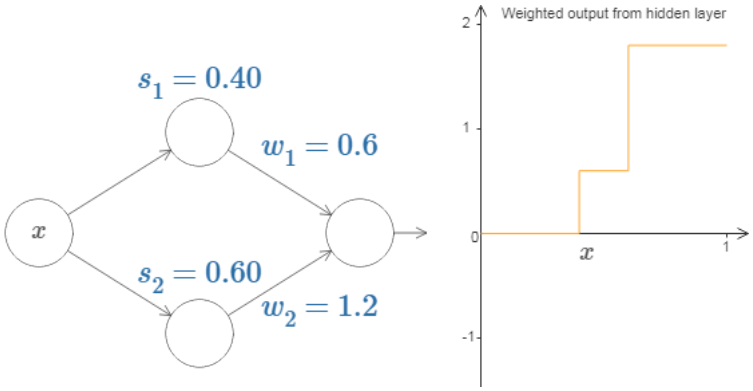


隠れニューロンを記述するのに段差の位置を示すパラメータ$s = -b/w$を用いると、解析を著しく単純にできます。新しいパラメータ付けに慣れるために、下図のsを変化させてみてください。



前述したように、入力の重みwを十分大きく取り、ステップ関数が良い近似になっていることを我々は暗黙のうちに仮定しています。バイアスを$b = -ws$と選ぶことで、1つのパラメータsで特徴づけられたニューロンを前のモデルに戻す事ができます。

これまで、私達は上の隠れニューロンの出力に注目してきました。ここでニューラルネットワーク全体の挙動を見てみましょう。2つの隠れニューロンは段差の位置がs_1（上ニューロン）とs_2（下ニューロン）でパラメータ付けられたステップ関数を計算しているとします。さらに、出力の重みをそれぞれw_1, w_2とします。ニューラルネットワークは以下の通りです：



右に図示しているのは隠れ層からの**重み付き出力**$w_1 a_1 + w_2 a_2$です。ここで、a_1とa_2はそれぞれ上下の隠れニューロンからの出力です* これらの出力はしばしば**活性(activation)**と呼ばれるため、aで表す事にします。

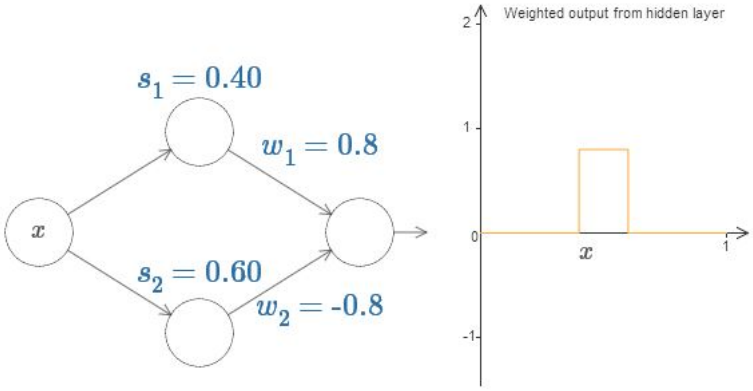
* ところで、bは出力ニューロンのバイアスとすれば、全ニューラルネットワークの出力は$\sigma(w_1 a_1 + w_2 a_2 + b)$である事に注意してください。もちろんこの値は今図示している隠れ層の重み付き出力とは異なります。今私達は隠れ層からの重み付き出力に注目しているので、ニューラルネットワーク全体の出力との関連付けはその後を考えます。

上の隠れニューロンの段差地点s_1を増減させて、隠れ層からの重み付き出力をどのように変化させるかについて感覚を掴んでください。特にs_1をs_2に通り越した時に何が起こるかを理解するのは有用です。上の隠れニューロンが反応する状況から下の隠れニューロンが反応する状況に変化するために、グラフの形が変化するのがわかると思います。

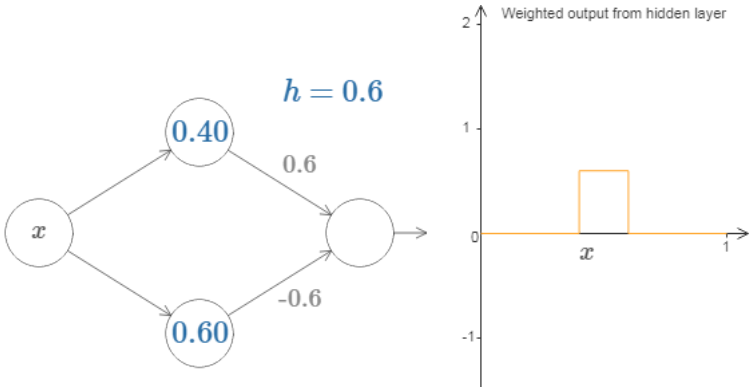
同様に、下側の隠れニューロンでの段差地点s_2を増減させて、出力が変化する様子の感覚を掴んでください。

出力の重みをそれぞれ増減させてみてください。それに応じてそれぞれの隠れニューロンからの寄与が拡大・縮小される事がわかります。重みのうちの1つを0にするとどのようなことが起こるでしょうか？

最後にw_1を0.8、w_2を-0.8にセットしてみてください。s_1から始まりs_2で終わる高さ0.8のコブ状の関数が得られます。例えば、重み付き出力はこのような感じです:



もちろん、コブの高さを任意に拡大・縮小できます。高さを表すパラメータhを導入しましょう。煩わしさを減らす為に"$s_1 = \ldots$"や"$w_1 = \ldots$"などの式を省略します。



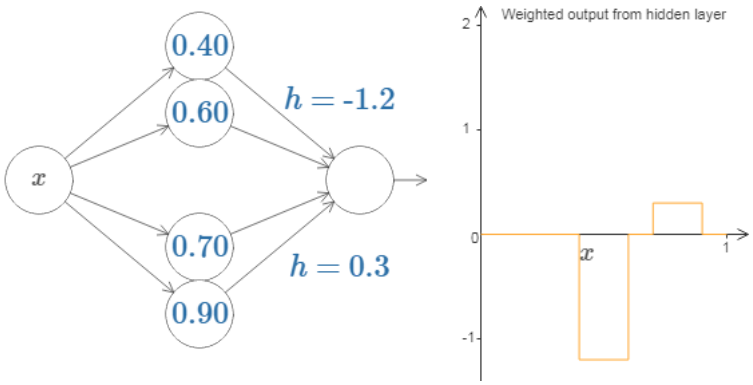
hの値を増減させてみて、コブの高さが変化する様子を見てください。高さを負の値に変化させて何が起こるかを観察してください。さらに、段差地点を変更してコブの形がどのように変化するかを見てください。

ところで、我々はニューロンを視覚的説明の観点からだけではなく、プログラミングの観点からも見ることができ、ニューロンを以下の様なif-then-else構文のようもみなせる事に気づいたかもしれません。例えば次の通りです

```
if input >= step point:
    重み付き出力に1を加える
else:
    重み付き出力に0を加える
```

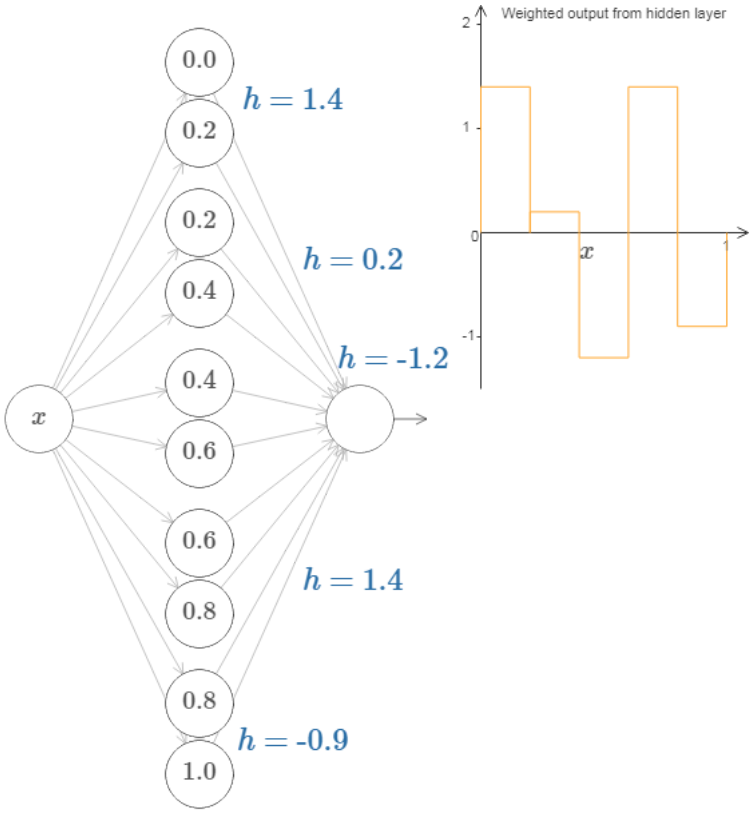
以降の説明の大部分では視覚的な観点にこだわろうと思います。しかしこれ以降の説明を、if-then-elseの観点で考えると理解に役立つかも知れません。

このコブを作るトリックを利用し、2組の隠れニューロンのペアを1つのニューラルネットワーク内でくっつける事で、2つのコブを作る事ができます:



ここでは重みは書かず、それぞれの隠れニューロンのペアに対してhを書きました。両方のhの値を増減させて、グラフがどのように変化するかを観察してください。また、段差の地点を変更させることでコブを移動させてください。

より一般的には、このアイデアを利用して好きな高さで好きな数のピークを構成できます。大きな数Nを用いて区間$[0, 1]$をN個の部分区間に分割します。そして、N組の隠れニューロンのペアを用いて好きな高さのピークを構成できます。$N = 5$の場合について見てみましょう。たくさんのニューロンがあるので少し詰めて描いています。図が複雑になってしまいすみません：省略して描けば複雑さを隠せるのですが、ニューラルネットワークにふるまいを具体的にイメージするため、若干の複雑さは我慢する価値はあると思います。



5組の隠れニューロンのペアが見て取れると思います。それぞれのペアの段差地点は$(0, 1/5), (1/5, 2/5), \ldots, (4/5, 5/5)$です。これらの値は固定されており、5つの等間隔に配置されたコブを持つグラフが得られるようにします。

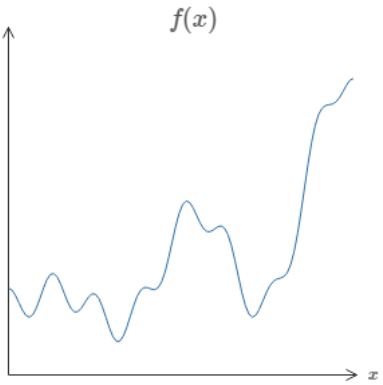
それぞれのニューロンペアにはhの値が伴っています。これらのニューロン達から出ている枝にはそれぞれh、$-h$の重みが与えられています(これらは図には載せていません)。hのうち一つをクリックし、左右にドラッグして値を変えてみてください。どのように関数が変化するかを観察しましょう。重みを変えることで、関数を**設計**することができるのです！

逆に、グラフをクリックし上下にドラッグして、どれかのコブの高さを変えてみてください。高さを変化するに対応して、hの値が変化するのがわかると思います。図には現れませんが、対応する出力の重み(これらの値は$+h$と$-h$です)も変化しています。

言い換えると、我々は右のグラフ内の関数を直接操作すると、それが左のニューラルネット内のhの値として反映されているのがわかります。マウスのボタンを押しっぱなしにして、一方からもう一方までドラッグしてみると面白いでしょう。関数を様々な形に引き伸ばすのに応じて、ニューラルネットワークのパラメータが変化する様子を見て取れます。

それではチャレンジの時間です。

この章の最初に私が描いた関数を思い出してください。



その時には言いませんでしたが、私が描いたのは

$$f(x) = 0.2 + 0.4x^2 + 0.3\sin(15x) + 0.05\cos(50x), \quad (106)$$

という関数のxが0から1の部分で、y軸は0から1の値を取っています。

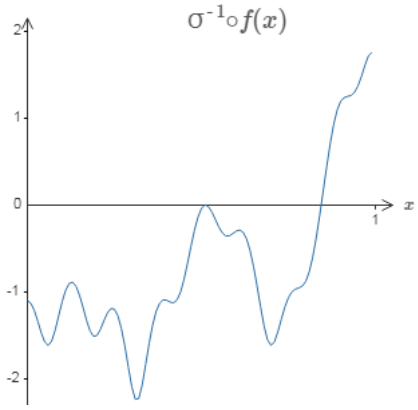
見ての通りこれは簡単な関数ではありません。

これをニューラルネットワークで計算する方法を導き出してみましょう。

前述のニューラルネットワークでは、隠れニューロンからの出力の重み付き和$\sum_j w_j a_j$を解析してきました。私達は既にこの値の調節方法を良く知っています。しかし、前に指摘したようにこの値はニューラルネットワークの出力ではありません。ニューラルネットワークの本来の出力は$\sigma(\sum_j w_j a_j + b)$です。ここで、$b$は出力ニューロンのバイアス項です。ニューラルネットワークの実際の出力を調節する方法は何かないでしょうか？

答えは、隠れ層の重み付き出力が$\sigma^{-1} \circ f(x)$を持つニューラルネットワークを設計することです。ここでσ^{-1}はσ関数の逆関数です。すなわち、

隠れ層からの重み付き出力を以下のようにします:

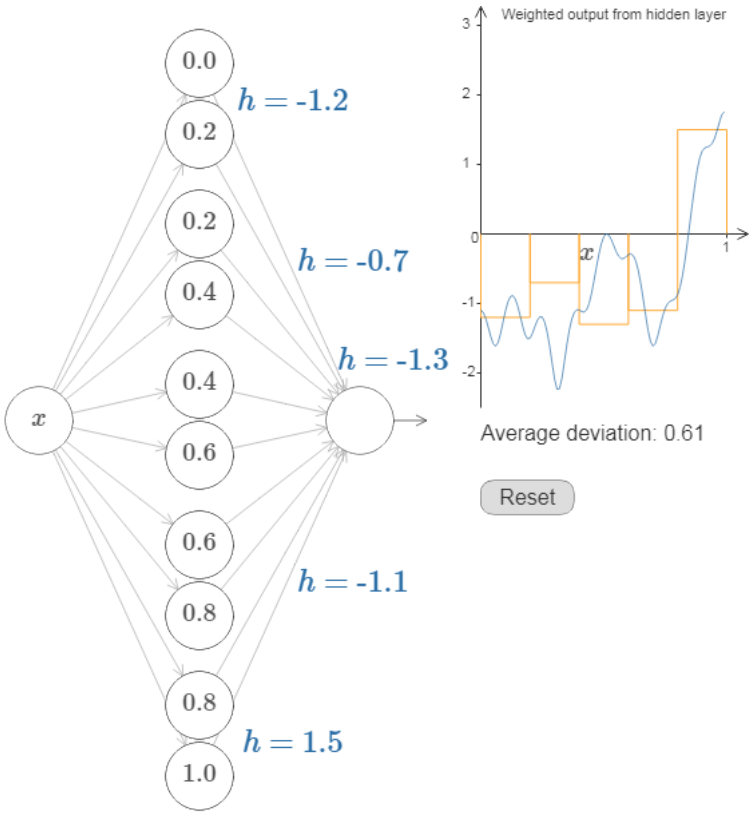


もしこれができれば、ニューラルネットワーク全体での出力は$f(x)$の良い近似となります*。

*出力ニューロンのバイアスを0としている事に注意してください。

チャレンジするのはニューラルネットワークを設計し、前述した目標関数を近似することです。できるだけ多くのことを学ぶために、この問題を2回解いてほしいと思っています。1回目はグラフをクリックし、それぞれのコブの高さを直接調節してください。目標関数によくマッチする関数を得るのは比較的簡単だと感じるはずです。調整がどの程度上手く行えているかは、目的関数とニューラルネットワークが計算している関数との間の**平均偏差**で測定できます。チャレンジするのは平均偏差を出来るだけ**小さく**することです。平均偏差を0.40かそれ未満に抑える事ができたらチャレンジは終了です。

1度目がうまくできたら、リセットボタンを押してコブをランダムに初期化し直してください。2度目はグラフをクリックしたくなる気持ちを抑えて問題を解いてください。その代わり左側のhの値を変更することで平均偏差を再び0.40かそれ未満に抑えてみてください。



これでニューラルネットワークを用いて関数$f(x)$を近似的に計算するために必要な要素が全て揃いました。これは粗い近似ですが、隠れニューロンのペアの数を増やすことで簡単に近似精度を良くできます。

私達が見つけ出したデータをニューラルネットワークの通常のパラメータでの表現に戻すのは簡単です。どのように行うかを簡単におさらいします。

最初の層のニューロンの重みは全て、例えば$w = 1000$などの適当な大きな定数です。

隠れニューロンのバイアスは$b = -ws$です。例えば、2つ目の隠れニューロンで$s = 0.2$ならば、$b = -1000 \times 0.2 = -200$です。

最終層の重みはhの値によって決まります。例えば、最初のhについて$h = $ -1.2 を選んでいるので、1番上の2つの隠れニューロンでの出力側の重みは、それぞれ -1.2 と 1.2 です。他の出力側の重みでも同様です。

最後に出力ニューロンでのバイアスは0です。

以上で、目標関数を十分良く計算できるニューラルネットワークを記述できました。また、隠れニューロンの数を増やして近似の精度を良くする方法もわかりました。

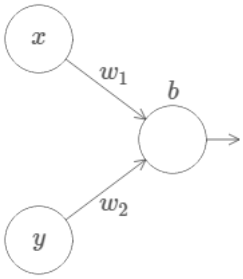
さらに、私達は目標関数
$f(x) = 0.2 + 0.4x^2 + 0.3\sin(15x) + 0.05\cos(50x)$について特別な仮定を置いていません。私達は$[0,1]$から$[0,1]$への任意の連続関数に対してこの手順を利用できます。本質的な部分は関数のルックアップテーブルを構築するのに、1層のニューラルネットワークを用いていることです。

このアイデアを用いて、一般の場合の普遍性定理の証明を行うことできます。

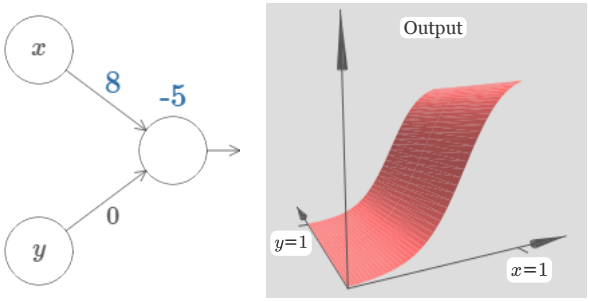
多変数の場合

以上の結果を多変数の場合に拡張しましょう。これは複雑そうに聞こえるかも知れません。しかし、必要なアイデアは全て入力が2つの場合で理解できますので、2入力の場合に注目して考えてみましょう。

ニューラルネットワークが2入力を持つ場合に何が起こるかを考える所から始めましょう:

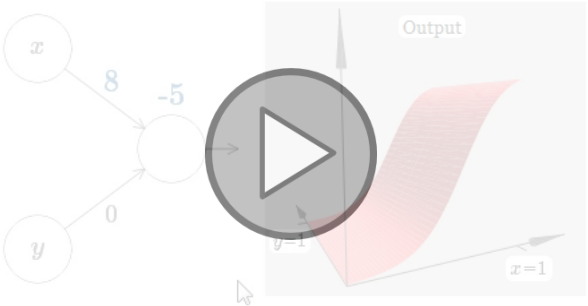


入力x, yとそれぞれに対応した重みw_1, w_2とバイアスbがあります。w_2の重みを0にした状態で1つ目の重みw_1とバイアスbをいじり、それらがニューロンの出力にどのように影響を与えるかを見てみましょう。



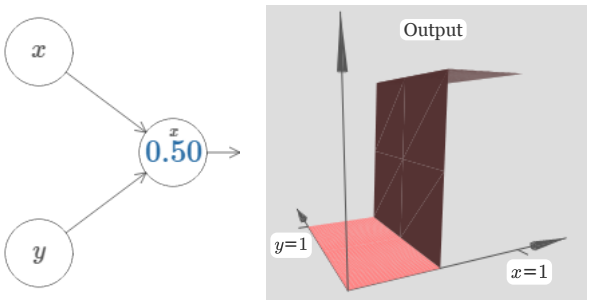
見ての通り、$w_2 = 0$とすると入力yの値はニューロンの出力に何の違いも生み出しません。まるでxのみが入力であるかのように振る舞います。

これを踏まえて、w_2を0としたままw_1の重みを増やして$w_1 = 100$とした時、何が起こると思いますか。もしすぐにこの答えが分からなかったら何が起こるかを少し考えてから、それが正しいかを試してみてください。以下の動画で何が起こるかを示しています:

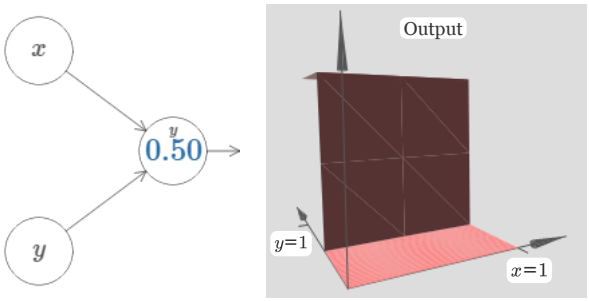


以前議論したように、入力の重みが大きくなるにつれて出力はステップ関数に近づきます。前と異なるのは今回はステップ関数が3次元である点です。前と同様にバイアスを変更することで段差の位置を動かすことができます。実際の段差の位置は$s_x \equiv -b/w_1$です。

段差の位置をパラメータとして、同じことをもう1度行ってみましょう:

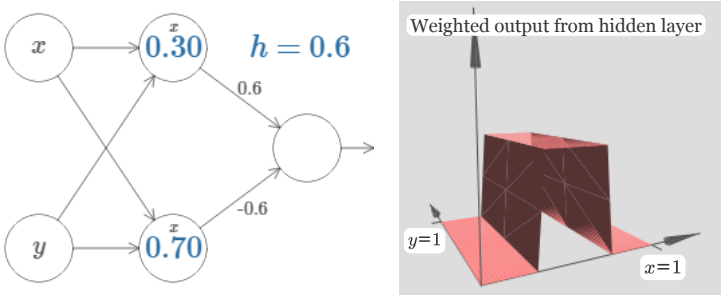


ここで、xの重みは適当な大きな値(私は$w_1 = 1000$を用いました)とし、$w_2 = 0$としています。ニューロン内の数字は段差の位置を示し、数字の上の小さなxの文字は段差がx方向である事を表しています。もちろん、yの重みを大きな値(例えば$w_2 = 1000$)としxの重みを0とする(すなわち$w_1 = 0$とする)ことで 段差をy方向に向けられます:



前と同様に、ニューロン内の数は段差地点です。数字の上の小さなyは段差が今度はy方向を向いていることを表しています。xとyそれぞれの重みを明示することも出来ましたが、そうすると図が混雑してしまうので書きませんでした。しかし、小さな文字yにより暗にyの重みが大きくてxの重みが0であることを示していることを忘れないで下さい。

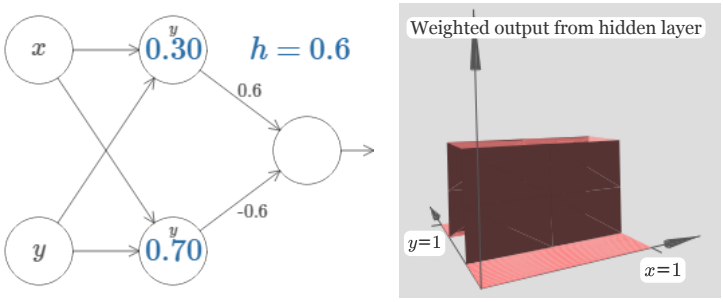
今つくったステップ関数を用いて3次元のコブ状の関数を構成できます。そのためには、x方向のステップ関数を計算する2つのニューロンを用意し、それらのステップ関数をそれぞれ重みhと$-h$で結合すればよいです。ここでhは作りたいコブの高さです。以上の内容が下図に表されています。



高さhの値を変更してみて、それがネットワークの重みとどのように関係しているかを観察してみてください。また、hの値により右のコブ状の関数の高さがどのように変化するかを見てください。

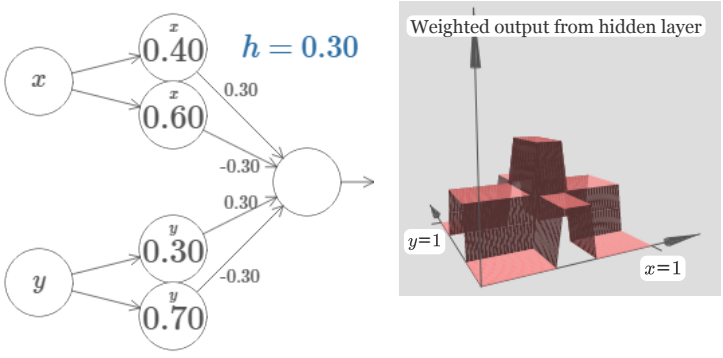
さらに、上の隠れニューロンによって決められている段差地点を0.30から変更してください。コブの形がどのように変化するかを見てみましょう。下の隠れニューロンによって決められる段差地点を0.70から変えた時に何が起こるでしょうか？

私達はx方向にコブ状の関数を作る方法を見つけました。もちろん、y方向のステップ関数を用いることでy方向のコブ状の関数を簡単に作れます。そうするには、入力yの重みを大きくして入力xの重みを0にすればよい事を思い出してください。下図が結果です:



これは先程のネットワークとほとんど同じものです！ 図中に示した中で唯一変更しているのは、隠れニューロン内の小さな文字をyにした点だけです。これは、今作っているのがx方向ではなくy方向のステップ関数であることを思い出すためのものです。つまり入力yはとても大きくてxは0であり逆ではありません。前と同様に図が煩雑にならないようにこの事は明示しませんでした。

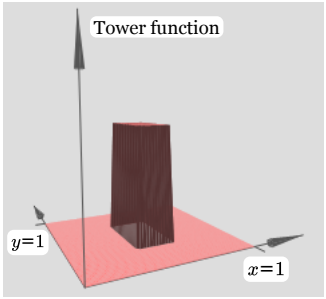
共に高さがhであるx方向とy方向の2つのコブ状の関数を足しあわせた時に、何が起こるかを考えてみましょう:



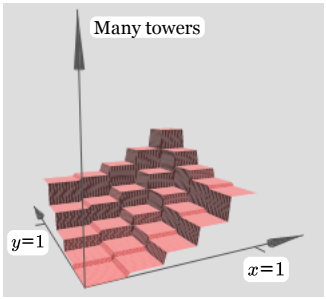
図を簡単にするために、重み0の枝は図から取り除きました。この図ではどちらの方向のコブ状の関数を計算しているかを忘れないように隠れニューロン内のxとyの小さな文字は残しておきました。しかし、コブの方向は入力変数からわかるので今後はこれらの文字も取り除きます。

パラメータhを変化させてみてください。見ての通りこれにより出力の重みが変化し、xとy両方のコブ状の関数の高さも変化します。

これにより少し**塔**に似た形の関数を構成することが出来ました:



もし、このような塔状の関数を構成できれば、様々な位置にある様々な高さの塔を足しあわせることで任意の関数を近似できます:



もちろん、我々はこのような塔状の関数を構成する方法をまだ見出していません。私達が実際に構成できているのは中央に高さ$2h$の塔がありそれを高さhの台地が囲むような関数です。

ところが、私達はここから塔状の関数を構成できるのです。ニューロンを`if-then-else`構文の形の関数を用いたことを思い出してください。

```
if input >= threshold:
    1を出力
else:
    0を出力
```

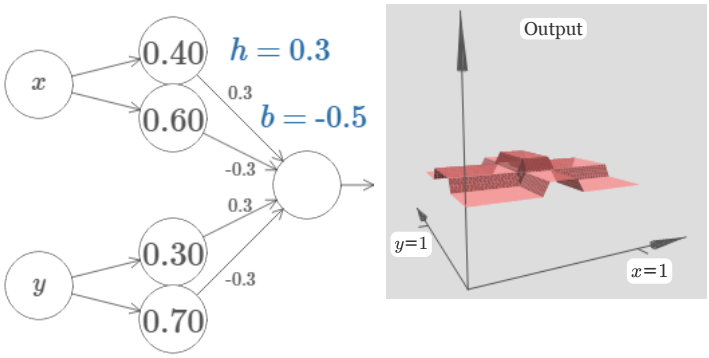
これは1入力しかないニューロンについてのものでした。今欲しいのは同様のアイデアを複数の隠れニューロン集合からの出力をまとめた値に対して適用したものです:

```
if 隠れニューロン集合からの出力をまとめた値 >= threshold:
    1を出力
else:
    0を出力
```

`threshold`を適切に、例えば台地の高さと中央の塔の高さに挟まれている$3h/2$のように、設定すると、塔が立った状態で残したまま、台地を高さ0に潰すことができます。

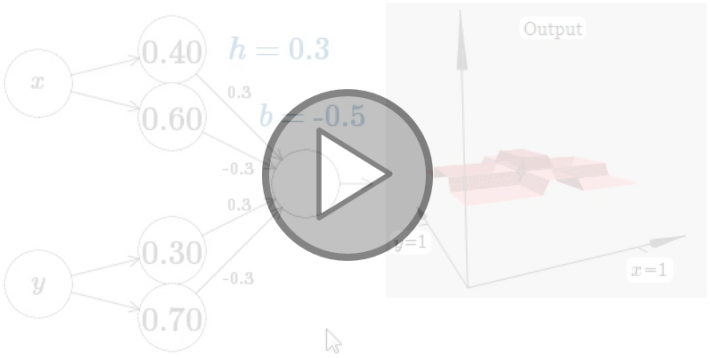
これをどのように行っているかわかりますか？ 理解するために次のニューラルネットワークで実験してみましょう。今回は隠れ層からの重み付き出力ではなく、ニューラルネットワーク全体の出力を図示していることに気をつけてください。つまり、隠れ層からの重み付き出力にバイアス項を加え、シグマ関数を適用しています。塔を作るためのhとbの値を見つけられますか。若干トリッキーなので、しばらく考えてそれでもわからなかったら次の2つのヒントを見てください: (1) 出力ニューロンが適切な`if-then-`

elseの挙動を示すためには、入力の重み(全てのhと$-h$)が大きな値でなければなりません。(2) bの値はif-then-elseの閾値を決定します。



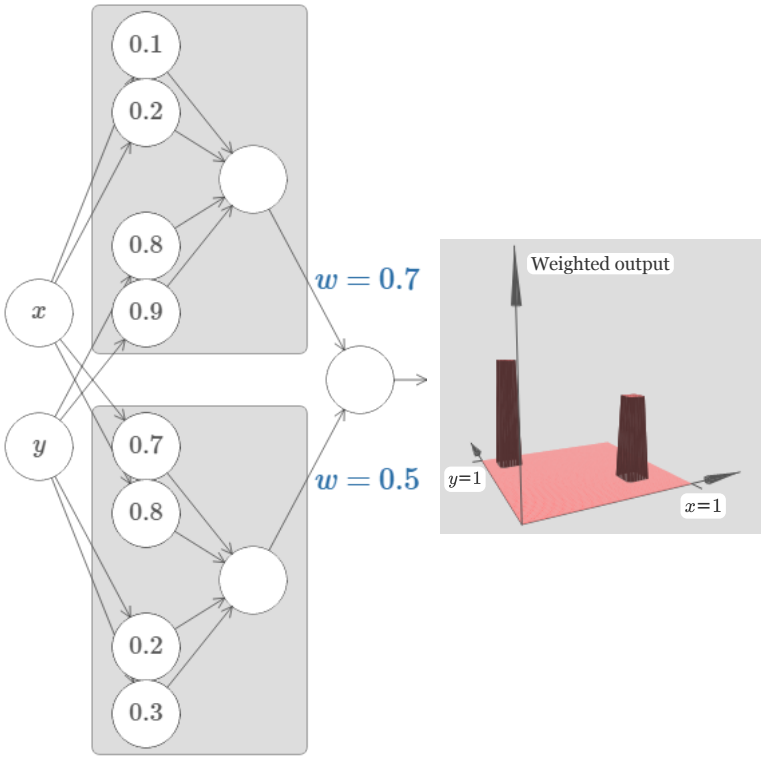
初期パラメータでは、出力は前の図での塔と台地を潰したような形をしています。望みの塔状の関数を作るには、まずパラメータhの値を十分に大きくしてください。そうすると、関数をif-then-elseの形の閾値で切るような形にすることが出来ます。次に、$b \approx -3h/2$として適切な閾値を設定してください。実際にやってみてどのようになるかを確かめてください！

下の動画は$h = 10$とした場合の様子です:



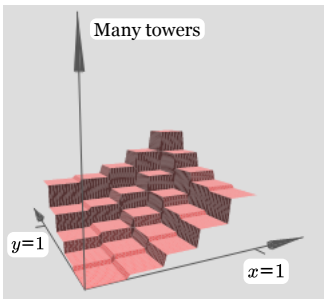
このような比較的小さな値のhでも、かなり良い塔状の関数を作る事ができます。もちろんhをさらに大きくしつつ$b = -3h/2$を保つことで、この関数の形をいくらでも良くすることができます。

今度は、2つの塔状の関数を表現するために、2つのネットワークをくっつけてみましょう。それぞれの部分ネットワークの役割を明確にするために、下図ではそれぞれを別の箱に入れました。それぞれの箱はこれまでのテクニックを用いて塔状の関数を計算しています。右側のグラフは**2層目**の隠れ層からの重み付き出力、すなわち、2つの塔状の関数の重み付きの重ねあわせを表しています。



特に、最終層の重みを変える事で、出力される塔の高さを変えることができます。

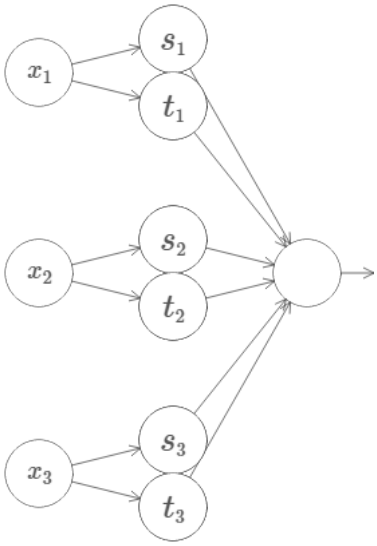
同じアイデアで好きな数の塔を計算することができます。さらにそれぞれの塔は好きな幅で好きな高さにすることもできます。従って2つ目の隠れ層の出力はどんな2変数関数も近似できることがわかります：



特に、2つ目の隠れ層の重み付き和を$\sigma^{-1} \circ f$の良い近似にする事で、ニューラルネットワーク全体の出力を好きな関数fの良い近似とする事ができます。

2変数関数以上の関数ではどうでしょうか？

3つの変数x_1, x_2, x_3を考えてみましょう。次のニューラルネットワークは4次元空間の中の塔状の関数を表現する事ができます。



ここで、x_1, x_2, x_3はネットワークの入力を表します。s_1, t_1等でニューロンの段差の点を表します。すなわち、最初の層の全ての重みを大きくし、バイアスを段差点が$s_1, t_1, s_2, \ldots$となるようにセットします。2つ目の層の重みは交互に$+h$と$-h$とします。ここでhは適当な大きな値です。さらに出力のバイアスを$-5h/2$とします。

「x_1がs_1とt_1の間にある」「x_2がs_2とt_2の間にある」「x_3がs_3とt_3の間にある」の3つの条件が満たされると、このネットワークは1を出力します。それ以外の場合には0を出力します。すなわち、この関数は入力空間の微小な領域では1を出力しそれ以外では0を出力する一種の塔状の関数を表しています。

このようなネットワークをたくさんつなげる事で、好きな数の塔を作り3変数の任意の関数を近似できます。同様のアイデアをm次元の場合にも当てはめられます。唯一変更しなければならない点は、台地の高さにあわせて適切な閾値を設定するために出力のバイアスを$(-m+1/2)h$とする点のみです。

これで、ニューラルネットワークを用いて実数値の多変数関数を近似する方法がわかりました。それでは、ベクトル値関数$f(x_1, \ldots, x_m) \in R^n$の場合はどうでしょうか？ もちろん、このような関数はn個の独立した実数値関数$f^1(x_1, \ldots, x_m), f^2(x_1, \ldots, x_m)$とみなす事ができます。従って、$f^1$を近似するニューラルネットワーク、$f^2$を近似する別のニューラルネットワーク・・・を構成できます。それらを単純につなぎ合わせれば、簡単にベクトル値関数の場合にも対応することができます。

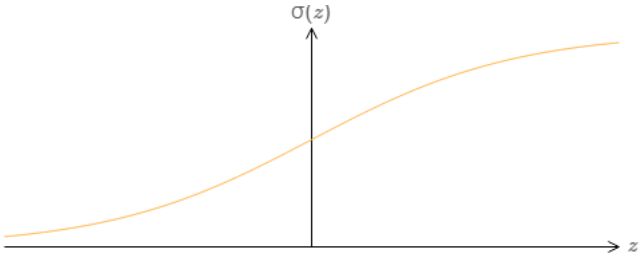
問題

- **ここまで2つの隠れ層を用いて任意の関数を近似してきました。 それでは、1つの隠れ層だけを用いても近似が可能であることを証明できますか？ ヒントとして、入力変数が2つしかない場合で次のことを示してください： (a) 階段上の関数はxやy方向だけではなく任意の方向にも構成できること (b) (a) での構成した関数を足し合わせて、長方形ではなく円周の形をした塔状の関数を近似できること (c) この塔状の関数を**

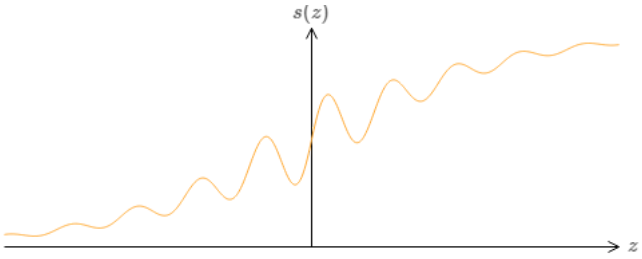
用いて任意の関数を近似できる事。 (c) を解く際に、この章で少し後に紹介するアイデアが役立つかもしれません。

シグモイド関数以外への拡張

私達はこれまでシグモイドニューロンでできたニューラルネットワークが任意の関数を計算できることを証明してきました。シグモイドニューロンでは、入力$x_1, x_2, \ldots$から$\sigma(\sum_j w_j x_j + b)$を出力することを思い出してください。ここで$w_j$は重み、$b$はバイアス、$\sigma$はシグモイド関数です。

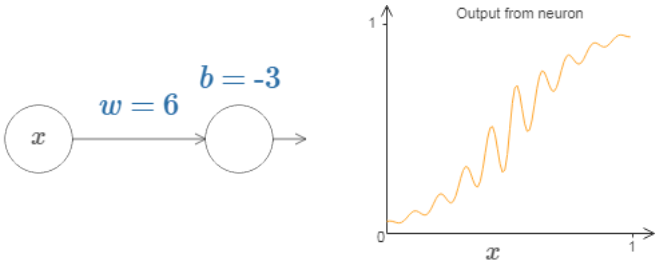


それでは、これとは異なる活性化関数$s(z)$を用いるニューロンを考えたらどうなるでしょうか:



つまり、ニューロンが入力$x_1, x_2, \ldots$、重み$w_1, w_2, \ldots$、バイアスbから、$s(\sum_j w_j x_j + b)$を出力するとします。

シグモイド関数の時に行ったのと同様にして、この活性化関数を用いてステップ関数を作ることが出来ます。 次の図の重みを例えば$w = 100$まで増加させてみてください:



シグモイド関数の時と同様に活性化関数は縮んでいき、最終的にはステップ関数の良い近似となります。バイアス項を変化させてみてください。すると、段差を自分が選んだ好きな位置に置けることがわかるはずです。これにより、私達は任意の好きな関数を計算するためにこれまでと同様のトリックを使うことができるのです。

この話がうまくいくためには、$s(z)$はどのような性質を持っている必要があるでしょうか。$s(z)$は$z \to -\infty$と$z \to \infty$でwell-definedでなければなりません。これらの極限はステップ関数がとる2つの値です。また、これらの2つの値は異なることも仮定する必要があります。もしそうでなかったとすると、階段のないただの平坦なグラフになってしまいます。$s(z)$がこの性質さえ持っていれば、それを活性化関数にもつニューロンを用いたニューラルネットワークは普遍性を持ちます。

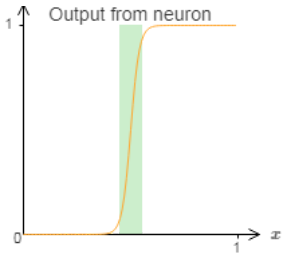
問題

- **この本の前の方で、Rectified Linear Unitと呼ばれる、別の種類のニューロンを取りあげました。 このニューロンが今説明した普遍性のための条件を満たさない理由を説明してください。 それにも関わらず、Rectifier Linear Unitによるニューラルネットワークは普遍性を持ちます。 その事の証明をしてください。**

- **線形ニューロン、つまり、$s(z) = z$という活性化関数を持つニューロンを考えます。 線形ニューロンは普遍性定理の条件を満たさない事を説明してください。 また、このニューロンによるニューラルネットワークは普遍性を持たないことを示してください。**

ステップ関数

これまで、我々が考えていたニューロンはステップ関数を計算できると仮定してきました。この仮定はとてもよい近似ですが、近似でしかありません。実際、次のグラフで示した非常に狭い「窓」において、ニューロンの出力はステップ関数とは大きく異なる振る舞いをしています:



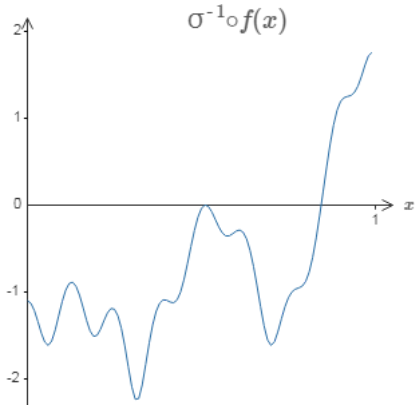
この窓の範囲では、普遍性について私が説明してきたことは成り立っていません。

しかし、これはそれほどひどい問題ではありません。ニューロンへの入力対する重みを十分大きくすることで、この窓を好きなだけ小さくすることが出来ます。実際、この窓を前に示した図中での幅よりも狭くしていき、目で判別できないほどの狭さにすることができます。ですのでおそらく私達はこの問題に気にすることはないでしょう。

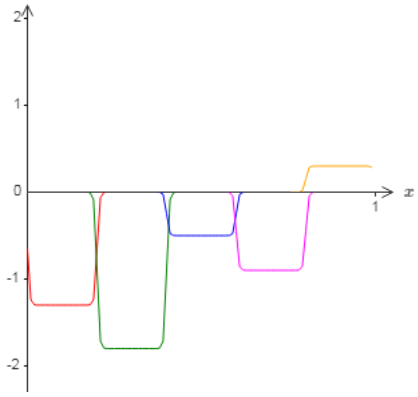
ですが、この問題について何らかの対処をしておくのは悪いことではありません。

実際、この問題は簡単に解決出来ます。入力も出力も1つしかないニューラルネットワークについてそれをみていきましょう。もっと多くの入出力がある場合も同じアイデアが使えます。

ニューラルネットワークをある適当な関数fを計算するようにしたいとします。前と同じように、ニューラルネットワークが隠れ層の重み付き出力が$\sigma^{-1} \circ f(x)$を出力するように設計してこれを実現します:

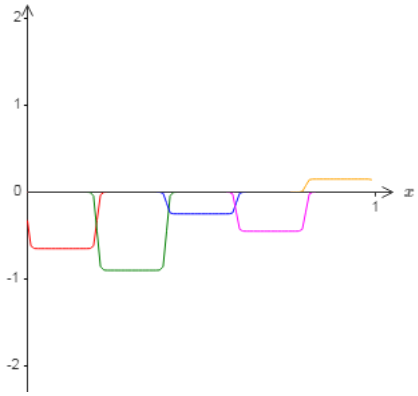


これまで説明したテクニックを用いて、隠れ層ニューロンを用いてコブ状の関数の列を作ることが出来ます。

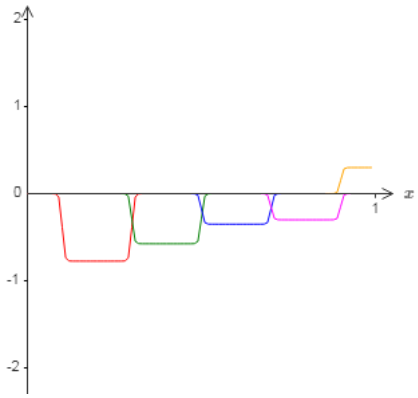


見やすくするために窓の幅を強調しています。これらのコブ状の関数を足し合わせれば、窓の外では$\sigma^{-1} \circ f(x)$の良い近似になることは比較的明らかでしょう。

この近似方法を用いる代わりに、目的の関数の**半分**、すなわち$\sigma^{-1} \circ f(x)/2$を近似するように隠れニューロンを調整したとします。もちろんこれは、直前のグラフの高さを低くしたもののような形となります。



そして、もう1組の隠れニューロンのセットを用いて、同様に$\sigma^{-1} \circ f(x)/2$を近似します。しかし、今度は位置をコブの幅の半分だけ**ずらした**関数を用います:



これにより$\sigma^{-1} \circ f(x)/2$の異なる近似が1つ得られました。これらを足し合わせることで、$\sigma^{-1} \circ f(x)$全体の近似が得られます。この関数は依然として近似がうまくできていない微小な窓を持っています。しかし、前と比べるとその問題はずっと小さくなっています。というのもm一方の近似関数での窓は他方では窓になっていないからです。従ってこれらの窓において近似はおおよそ2倍程度良くなっています。

大きな数Mについて、M個の互いに重なりあった$\sigma^{-1} \circ f(x)/M$の近似関数を足し合わせることで、近似をさらに良くできます。窓が十分狭ければ数直線上の各点は高々1個の関数の窓の内側に含まれます。そして、重なり合っている関数の個数Mを十分大きく取れば、得られる関数は全体を十分に近似できるものになります。

結論

ここまで行ってきた普遍性定理の説明はニューラルネットワークを用いた計算に対する現実的な処方箋ではありません！ その観点ではNANDゲートに対する普遍性定理と状況は同じです。ですのでニューラルネットの構成方法は明快で解説が追いやすくなることに注力し、構成方法の細かい部分の最適化はしませんでした。この構成方法を更に良くするのは面白く勉強になるでしょう。

本章での結果はニューラルネットワークを構成するのに直接有用ではありませんが、任意の関数をニューラルネットワークで計算可能かという問いを議題から外すことができる、という点では重要です。その問いへの答えは常に「可能である」だからです。従ってある特定の関数を計算可能かではなく、その関数を計算する**良い**方法は何かというのが正しい問題設定となります。

普遍性定理の証明では任意の関数を計算するのに、2層の隠れ層を用いました。さらに、前述のように1層の隠れ層でも同様の結果が得られます。そうするとなぜ私達が深いネットワーク、つまり隠れ層をたくさん持つネットワークに興味を持っているかを疑問に思うかもしれません。このよう

な深いネットワークを隠れ層が1層しかない浅いネットワークに置き換えることは出来ないでしょうか？

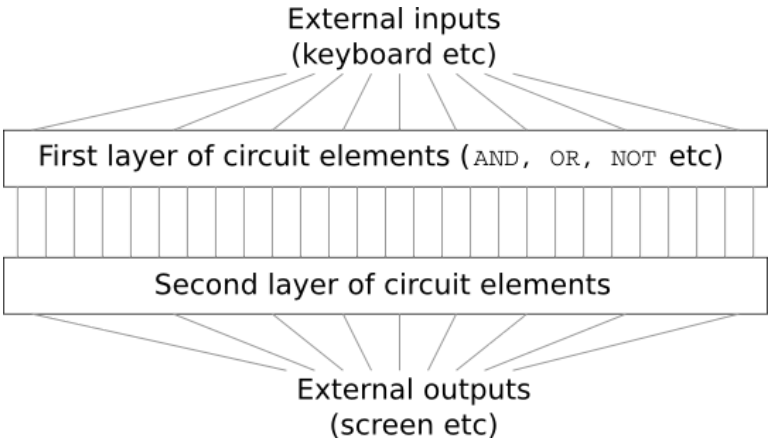
原理的にはそのような浅いネットワークへの置き替えは可能ですが、深いネットワークを利用するのにはきちんとした現実的な理由があります。第1章で議論した通り、深いネットワークは階層的な構造をしています。このことは知識の階層構造を学習するのに適しており、現実世界の問題を解くのに有用です。もう少し具体的に言うと、画像認識などの問題に取り組む場合、個々のピクセルを認識するだけではなく、輪郭や単純な図形から複雑で多物体からなるシーンまで、より複雑な概念を理解するのに役立ちます。この後の章では、深いネットワークが浅いネットワークに比べてこのような知識階層を学習するのに適していることを示唆する証拠を見ていきます。まとめると、普遍性定理からニューラルネットワークが任意の関数を計算できることがわかりました。そして、深いネットワークは多くの現実世界の問題を解くのに有用な関数を学習するのに適していることが経験的にわかっています。

謝辞 Jen DoddとChris Olahにはニューラルネットワークの普遍性に関して多くの議論を行ったことを感謝します。特にChrisには、普遍性定理の証明の中でルックアップテーブルを利用することを提案してもらいました。本章のインタラクティブな図は Mike Bostock、Amit Patel、Bret Victor、Steven Wittens などの仕事を参考にしました。

CHAPTER 5

ニューラルネットワークを訓練するのはなぜ難しいのか

自身が、コンピュータを一から設計するよう求められているエンジニアであると想像してみてください。あなたはある日、オフィスの外にいます。論理回路を設計するため、ANDゲートやORゲートから手を付け始めています。そこへ、あなたの上司が悪い知らせを持ってきました。顧客が驚くべき設計要件を追加したのです。それは、コンピュータ全体をたった2層の深さの回路で作れという要件でした。



あなたは慌てて、上司に言いました。「この顧客は狂っています！」

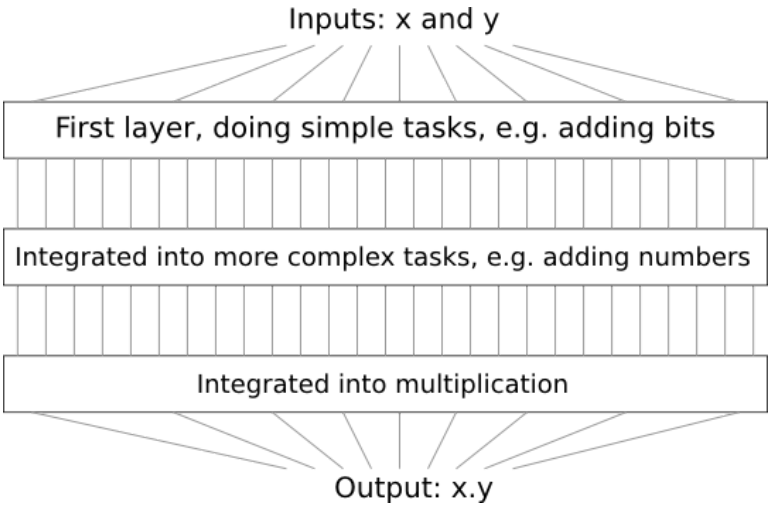
「彼らは狂ってると私も思う。でも、顧客が望むものは提供するしかないんだ」と上司は答えます。

実際には、顧客が狂ってるわけではありません。どんなに入力が多くてもANDを取得できる、特別な論理ゲートを使えると想定してみてください。加えて、多入力のNANDを取得できる論理ゲートも使えるとします。このNANDゲートは、複数入力のANDを取得し、その結果を反転させて出力するゲートです。これらの特別な論理ゲートを使えば、ちょうど2層の回路で、どんな関数でも計算できます。

でも、実現可能だからと言って、良いアイデアとは限りません。実際、回路設計問題（もしくはアルゴリズムのあらゆる問題）を解くときには、私たちはまず、部分問題から解き始めます。そして、徐々に部分問題の解を組み合わせていくのです。すなわち、多数の層の解を抽象化させて、全体の解を作り上げていきます。

たとえば、2つの数の掛け算を行う論理回路を設計するとしましょう。きっとあなたは、2つの数の足し算を行う部分回路を組み合わせたいと思います。そして2つの数の足し算を行う部分回路を作るときには、今度は、2

つのビットの足し算を行う、部分-部分回路を組み合わせるでしょう。大ざっぱに言うと、設計した論理回路はこんな形をしています。



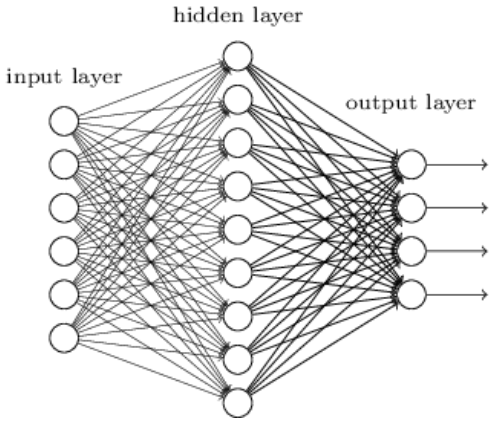
最終的な論理回路は、少なくとも3層の回路要素から構成されます。私が述べたものよりも、さらに部分問題にブレークダウンしていけば、3層以上含むことになるでしょう。ただ、これまでの話の中で、あなたは汎用的に使えるアイデアを得たはずです。

このように、回路を深くすれば、設計は簡単になっていきます。でも設計が楽になるだけではありません。実際、同じ計算を行う際、深い回路に比べると、浅い回路を使う場合には指数関数的に多くの回路要素が必要となることが、数学的に証明されています。たとえば、1984年のFurst、Saxe、そしてSipserの有名な論文* では、ビットのパリティを計算する際に、浅い回路を用いると、指数関数的に多くのゲートが必要となることが示されています。一方で、深い回路を使うとなれば、小さな回路要素でパリティ計算できます。そのときには、まずビットの組のパリティを計算します。その結果を使って、ビットの組の組のパリティを計算します。それを繰り返して、全体のパリティを素早く計算できます。深い回路は浅い回路よりも、本質的に遥かに強力なのです。

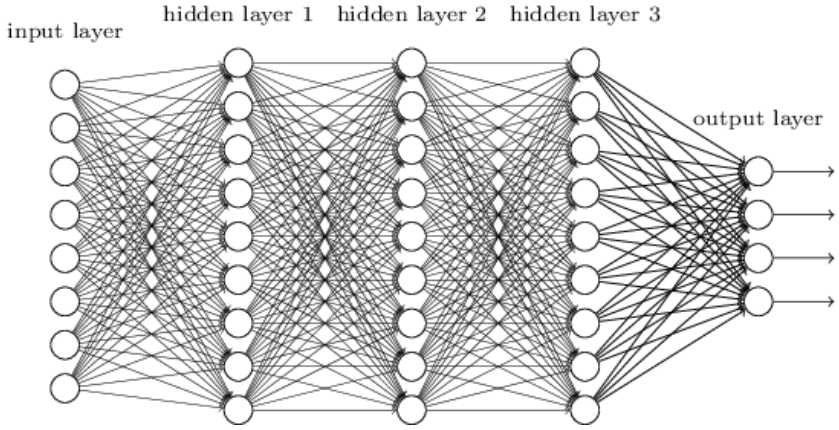
これまで、この本はニューラルネットワークに対して、先ほど述べた狂った顧客のような取り組み方をしてきました。すなわち、これまで扱ってきたほぼ全てのネットワークは、たった一つの隠れ層（に加えて入力層と出力層）しか持っていませんでした。

著：Michael Nielsen / 2014年9月-12月
訳：「ニューラルネットワークと深層学習」翻訳プロジェクト

* Parity, Circuits, and the Polynomial-Time Hierarchy, by Merrick Furst, James B. Saxe, and Michael Sipser (1984) を確認してください。



これらのシンプルなネットワークは、驚くほど有用です。前章までに、このネットワークを使った手書き数字の分類で、98%以上の精度を出しました！ それにも関わらず、多くの隠れ層を備えるネットワークの方が強力なのではないか、という直感的な期待があるのです。



多層のネットワークは、先ほどの論理回路の例と同じように、中間層を抽象化のために組み合わせて使います。たとえば、私たちが視覚パターン認識をするとしましょう。そのときには、第一層のニューロンはエッジを認識するようになります。第二層のニューロンは、三角形や四角形などの複雑な形状を、エッジの情報から認識します。第三層のニューロンはさらに複雑な形状を認識します。以降はこれの繰り返しです。どうやら、これらの抽象化層のおかげで、ニューラルネットワークは複雑なパターン認識問題を解く方法を学習できるようです。加えて、回路の例と同じように、深いネットワークは、浅いネットワークよりも元来強力であるという理論的な結果があります* 。

このような深いネットワークを訓練するにはどうすればよいのでしょうか？ この章では、働き者の学習アルゴリズムである 逆伝播による 確率的勾配降下法 を使って、深いネットワークを訓練していこうと思います。しかしその過程で、深いネットワークを使っていても、浅いネットワークのときよりもパフォーマンスが良くならないという問題にぶつかります。

*特定の問題とネットワーク構造については、このことが以下で証明されています。On the number of response regions of deep feed forward networks with piece-wise linear activations, by Razvan Pascanu, Guido Montúfar, and Yoshua Bengio (2014). さらなる非公式な議論については、以下のセクション2を参照してください。Learning deep architectures for AI, by Yoshua Bengio (2009).

上記の議論の後では、そのような失敗が起きることが、不思議に思えるでしょう。失敗するからと言って、深いネットワークに見切りをつけずに、深いネットワークを訓練する難しさの原因を掘り下げて、理解していきましょう。よく観察すると、深いネットワーク中の異なる層は、まったく異なるスピードで学習していることがわかります。特に、後ろの方の層は大きく学び、前の方の層はしばしば訓練中にもたついて、全く何も学ばなかったりします。この前方の層のどん詰まりの状況は、単に運が悪かったというわけではありません。むしろ、勾配を利用した学習テクニックによく見られる、学習が遅くなってしまう本質的な理由がそこにはあります。

問題を深く掘り下げていくと、対照的な現象の発生も目撃します。前の方の層が大きく学び、後ろの方の層が学ばないという現象です。実際、深く多層からなるニューラルネットワークの場合、勾配降下法による学習には不安定性が伴うことを学びます。この不安定性により、前方の層か後方の層のどちらかが、訓練時に学習しなくなる傾向があります。

これは悪い知らせに聞こえます。しかし、これらの問題を掘り下げていくと、深いネットワークを効果的に訓練するための必要事項に関して洞察を得ることができます。そして、この調査が次の章への良い準備になるのです。次の章では、深層学習を使って画像認識問題に取り組みます。

勾配消失問題

さて、深いネットワークを訓練しようとするときに、何が上手く行かないのでしょうか？

この問いに答えるために、隠れ層1つのみのネットワークの例を再び見てみましょう。いつも通り、MNISTの手書き文字に対する分類問題を、学習と実験の場として使います* 。

*MNIST の問題とデータは ここ と ここで紹介しました

お望みなら、自身のコンピュータでネットワークを訓練して追うことができます。もちろん、ただ読み進めるだけでも構いません。自身で実験をしつつ内容を追いたい場合、Python 2.7、Numpy、そしてコードが必要です。コードはコマンドラインを使ってレポジトリから複製できます。

```
git clone https://github.com/mnielsen/neural-networks-and-deep-learning.git
```

もしgitを使わない場合、データとコードをここからダウンロードできます。srcサブディレクトリへ行ってください。

そして、Pythonのシェルから MNISTのデータをロードしてください。

```
>>> import mnist_loader
>>> training_data, validation_data, test_data = ¥
... mnist_loader.load_data_wrapper()
```

ネットワークをセットアップします。

```
>>> import network2
>>> net = network2.Network([784, 30, 10])
```

このネットワークは入力層に784のニューロンを持ち、これらが入力画像の $28 \times 28 = 784$ ピクセルに対応します。30の隠れニューロンと、10の出力ニューロンを使います。この出力ニューロンがMNISTの文字の ('0', '1', '2', ..., '9') の10分類に対応します。

さあ、10個の訓練画像を含むミニバッチ、学習率は $\eta = 0.1$ 、正規化パラメータ $\lambda = 5.0$ の条件下で、ネットワークを30エポック分、訓練してみましょう。validation data*を分類する正確さの変化を、訓練の進行にしたがい観測します*

```
>>> net.SGD(training_data, 30, 10, 0.1, lmbda=5.0,
... evaluation_data=validation_data, monitor_evaluation_accuracy=True)
```

*ネットワークの訓練にはかなり時間がかかることに注意してください。コンピュータの性能によりますが、エポックごとに最大2,3分程度かかります。なので、コードを実行しているなら、終了を待たずにそのまま読み進めて、後で振り返るとよいでしょう。

分類精度は96.48%(もしくは、実行するごとにちょっとだけ変化しますが大体その程度)です。この結果は、同じ設定で実施した先の結果とほぼ同じ数値です。

さて、同じ30個のニューロンを持つ隠れ層を追加してみましょう。このネットワークを、同じハイパーパラメータのもとで訓練します。

```
>>> net = network2.Network([784, 30, 30, 10])
>>> net.SGD(training_data, 30, 10, 0.1, lmbda=5.0,
... evaluation_data=validation_data, monitor_evaluation_accuracy=True)
```

分類精度が向上し、96.90%となりました。この結果は励みになります。少し深くしたことで、良い結果が得られたようです。さあ、さらに30個のニューロンを持つ隠れ層を追加しましょう。

```
>>> net = network2.Network([784, 30, 30, 30, 10])
>>> net.SGD(training_data, 30, 10, 0.1, lmbda=5.0,
... evaluation_data=validation_data, monitor_evaluation_accuracy=True)
```

今回は結果が良くなりませんでした。精度が96.57%へ落ちています。一番最初の浅いネットワークの結果に近づきました。もう1つ隠れ層を追加しましょう。

```
>>> net = network2.Network([784, 30, 30, 30, 30, 10])
>>> net.SGD(training_data, 30, 10, 0.1, lmbda=5.0,
... evaluation_data=validation_data, monitor_evaluation_accuracy=True)
```

分類精度は再び悪化し、96.53%となりました。統計的に有意な悪化ではおそらくないですが、良い結果ではありません。

この振る舞いは奇妙に思えます。直感的には、隠れ層を追加したことで、ネットワークは複雑な分類関数を学習でき、分類タスクもうまくいくは

ずです。最悪の場合でも、隠れ層に単に何もさせないようにすれば、悪化しないはずなのです*。しかし、実際には悪化が起きています。

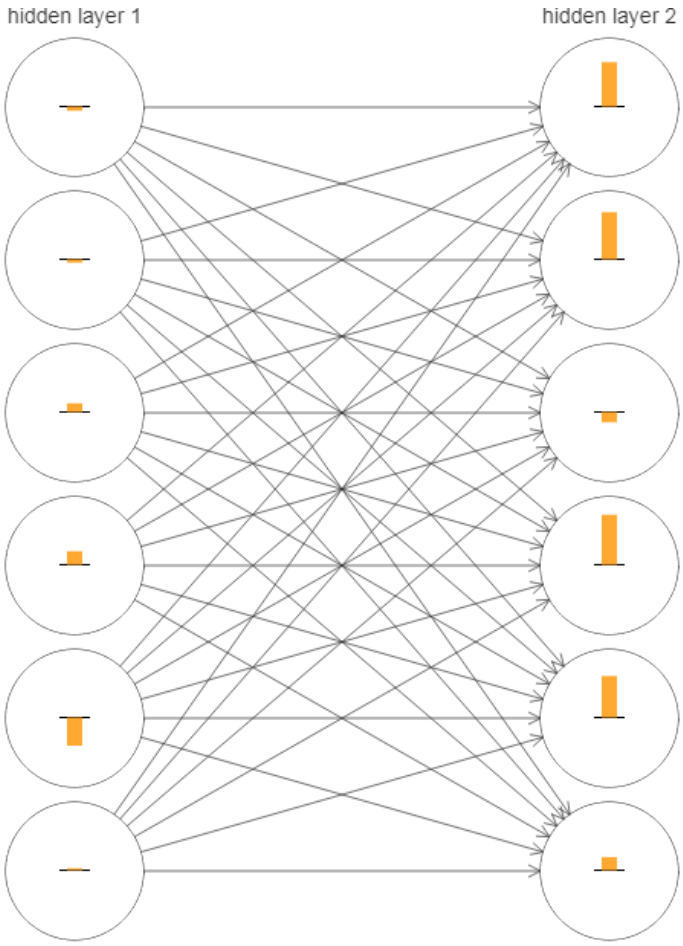
*何もしない隠れ層がどう生まれるかを理解するための後に紹介する問題を確認してください。

では一体、何が起きているのでしょう？ 隠れ層を追加することで、原理的には上手くいくとすると、問題は学習アルゴリズムが適切な重みとバイアスを探せていないことです。学習アルゴリズムの上手くいかない点とその対処法を見つけたいです。

何が悪いのかの洞察を得るために、ネットワークの学習の様子を可視化してみましょう。下図に、$[784, 30, 30, 10]$ のネットワークの一部をプロットしました。このネットワークは、2層の隠れ層を持ち、それぞれが30個のニューロンを持っています。図の中の各ニューロンには小さなバーがあります。このバーは、ネットワークの学習に伴ってニューロンが変化する速さを示しています。バーが大きいと、ニューロンの重みとバイアスは急速に変化しており、一方で、バーが小さいと、重みとバイアスの変化が遅いことを示します。より正確に言うと、バーは各ニューロンの勾配 $\partial C/\partial b$ を示しています。この勾配は、ニューロンのバイアスに対するコスト関数の変化率のことです。2章で見たように、この勾配の大きさが、学習中のバイアスの変化の速さだけでなく、ニューロンに入力される重みの変化の速さも制御します。詳細を思い出せなくても心配しないでください。このバーは、ネットワークの学習時の、ニューロンの重みとバイアスの変化の速さだということだけ覚えておけばよいです。

図をシンプルにするために、隠れ層のうち単純に上から6個のニューロンのみ表示しました。学習すべき重みやバイアスを持たない入力ニューロンは省略してあります。同じ数のニューロンを持つ層ごとの比較を行いたいので、出力ニューロンも省略しました。プロットした結果は、訓練開始時のもの、すなわちネットワークが初期化された直後のものです* 。

*プロットしたデータは、generate_gradient.pyを使って生成されたものです。同じプログラムはこのセクションの後に引用される結果でも使われています。



ネットワークはランダムに初期化されていたので、ニューロンの学習速度のバラつきがあるのには驚きません。しかし一つ気になるのは、2個目の隠れ層のバーが1個目の隠れ層のバーより遥かに大きい点です。つまり、2個目の隠れ層は1個目の隠れ層よりも高速に学習するということのようです。これは単なる偶然でしょうか、それとも一般的に2個目の隠れ層は1個目の隠れ層よりも速く学習するものなのでしょうか？

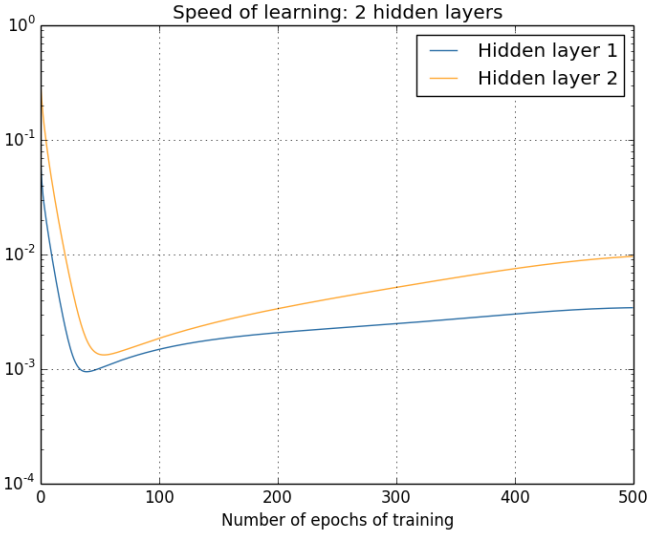
これが本当かどうか調べるために、一般的な方法で1層目と2層目の学習速度を比較してみます。このために、勾配を $\delta^l_j = \partial C/\partial b^l_j$ とします。これは l 層目の中の j 個目のニューロンの勾配*です 。勾配 δ^1 は1個目の隠れ層の学習速度を定めるベクトルとしてみなすことができ、δ^2 は2個目の隠れ層の学習速度を定めるベクトルとみなせます。これらのベクトルの長さを、層の学習速度の一般的な尺度（大ざっぱ！）として使いましょう。したがってたとえば、長さ $\|\delta^1\|$ は1層目の隠れ層の学習速度を示し、一方で、長さ $\|\delta^2\|$ は2層目の隠れ層の学習速度を示すこととします。

これらの定義に従って、上図の設定のときの値を確認すると、$\|\delta^1\| = 0.07\ldots$ であり $\|\delta^2\| = 0.31\ldots$ です。したがって、先程の疑いを確かめたことになります。実際に、2個目の隠れ層のニューロンは、1個目の隠れ層のニューロンよりも遥かに速く学習していたのです。

*2章を振り返ると、これを誤差と呼んでいました。しかし、ここでは「勾配」という非公式な呼び方を採用します。「非公式」と言っているのは、もちろんこの式が、コスト関数の重みに関する偏微分 $\partial C/\partial w$ を明示的に含んでいないからです。

隠れ層をさらに追加したとしたら、何が起こるでしょうか？ もし、隠れ層が3個となったときには、各層での学習速度は 0.012、0.060、そして0.283となります。やはり、前方の隠れ層の学習は、後方の隠れ層よりかなり遅くなっています。さらに30個のニューロンを持つ隠れ層を追加してみます。この場合には、各層での学習速度は、0.003、0.017、0.070、そして0.285となります。パターンとしては同じです。前方の層は後方の層よりも遅く学習しています。

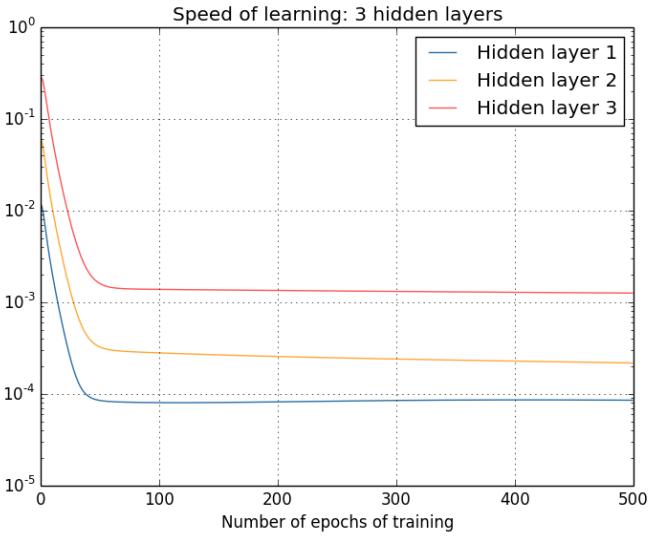
これまでは訓練開始頃、すなわちネットワーク初期化直後の学習速度を見てきました。ネットワークの訓練が進むと、学習率はどのように変化するのでしょうか？ 2個の隠れ層のみ持つネットワークに立ち戻って確認してみましょう。学習速度は次のように変化します。



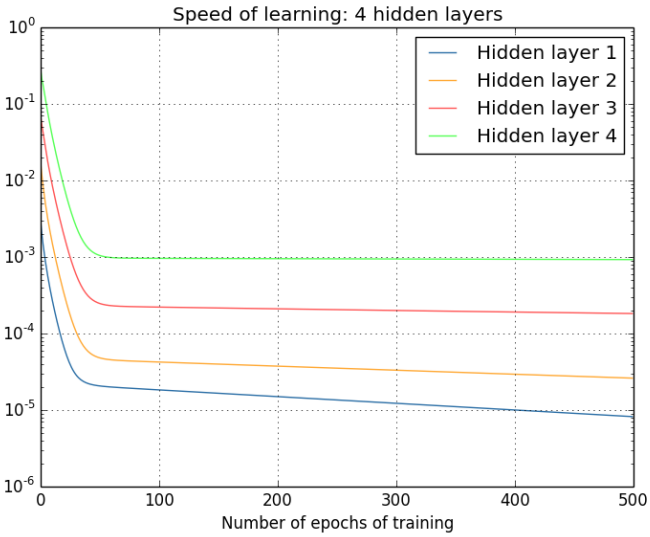
この結果を生成するために、1000個の訓練画像に対して、バッチ勾配降下を500エポックに渡り行いました。これは通常の訓練方法と少し異なります。ミニバッチを使わずに、しかも50000の訓練画像全体ではなく、たった1000の訓練画像のみを使ったからです。もちろん卑怯な方法を使ったわけでも、あなたの目をくらまそうとしたつもりもありません。ミニバッチ確率的勾配降下法を使うと、ノイズが混じった結果となってしまうのです（ノイズを平均化すればとても似た結果になりますが）。私が選んだパラメータを使った方が、結果を簡単に滑らかにできるため、何が起きてるか見やすくなるはずです。

（既に知っているとおり、）2つの層が大きく異なる速度で学習し始めているのがわかります。そして、どちらの層も急速に減少し、リバウンドします。しかし全体的に、1個目の隠れ層は、2個目の隠れ層よりもゆっくり学習していると言えます。

もっと複雑なネットワークの場合はどうなるでしょうか？ 次の図は、3個の隠れ層 $[784, 30, 30, 30, 10]$ を持つネットワークの場合の実験結果です。



今回も、前方の隠れ層は後方の隠れ層より遅くなっています。最後に、4個目の隠れ層を追加し($[784, 30, 30, 30, 30, 10]$ のネットワークとして)、訓練時に何が起きるか見ましょう。



またもや、前方の隠れ層は後方のものよりも遅くなりました。今回、最初の隠れ層は、最後の隠れ層よりも約100倍遅いという結果になっています。前方のネットワークを訓練する難しさがわかったことでしょう！

私たちは重要な現象を観察しています。少しでも深いニューラルネットワークでは、勾配は隠れ層を前方に伝わるにつれて小さくなる傾向があります。これにより、前方の層のニューロンの学習は、後方の層のニューロンよりも遥かに遅くなります。この現象をたった1つのネットワークの例で確認しましたが、実は多くのニューラルネットワークにおいて、同様の現象が発生する根源的な理由があります。これは**勾配消失問題***として有名な現象です 。

*Gradient flow in recurrent nets: the difficulty of learning long-term dependencies, by Sepp Hochreiter, Yoshua Bengio, Paolo Frasconi, and Jürgen Schmidhuber (2001)を確認してください。この論文はrecurrent neural networkを扱ったもの

なぜ勾配消失問題が起きるのでしょうか？ この問題を避けられる方法はあるのでしょうか？ 深いニューラルネットを訓練するときにどう対処すべきなのでしょうか？ 実は、代替策がなくもないということをすぐに学びます。しかしその代替策はそこまで魅力的なものではありません。勾配が前方の層でかなり大きくなってしまう、という現象がしばしば発生してしまうのです！ これは**勾配爆発問題**と言い、勾配消失問題と同じく悩ましい問題です。一般的には、深いニューラルネットワークにおける勾配は**不安定**であり、前方の層で爆発もしくは消失する傾向があります。この不安定性は、深いニューラルネットワークを勾配を利用して訓練するときの重大な問題です。この問題は理解する必要があり、もし可能であれば、対処する手を打たなくてはいけません。

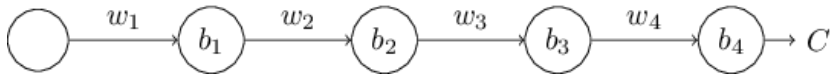
勾配消失（もしくは勾配の不安定性）への1つの反応は、本当にそれが問題になるのかと疑問に思うことです。ニューラルネットから一瞬離れて、ある1変数関数 $f(x)$ を数値的に最小化することを考えてみましょう。微分 $f'(x)$ が小さいことは良い知らせではないでしょうか？ 既に極値に近づいていることを意味しているのではないでしょうか？ こんな風に考えると、深いネットワークの前方の層での勾配が小さいということは、重みとバイアスをこれ以上調整する必要が無いことを意味しているのではないのでしょうか？

もちろん、これらの考えは間違っています。ネットワークの重みとバイアスをランダムに初期化したことを思い出してください。どんなネットワークであっても、重みとバイアスの初期値がいい感じになっているなんて、ありえないことです。具体的に確認するために、MNIST問題用の $[784, 30, 30, 30, 10]$ のネットワークにおける最初の層の重みを考えてみます。ランダムに初期化するということは、最初の層は、入力画像についての情報を捨てているということです。たとえ、後方の層が素晴らしく訓練されていたとしても、後方の層からは入力画像を同定することができない状況となるのです。なので、最初の層での学習は必要です。深いネットワークを訓練する場合には、この勾配消失問題にどう取り組むべきかを明らかにする必要があります。

勾配消失問題の原因とは？ 深いニューラルネットにおける不安定な勾配

なぜ勾配消失問題が起きるかを考察するために、一番シンプルな深層ニューラルネットワークを考えます。それは各層にただ1つニューロンをもつネットワークです。下図のネットワークは3つの隠れ層を持ちます。

ですが、私たちの調べている順伝播ネットワークと本質的な現象は同じです。Sepp Hochreiterの学位論文、Untersuchungen zu dynamischen neuronalen Netzen (1991, in German)についても見てください



ここで、$w_1, w_2, \ldots$ は重み、$b_1, b_2, \ldots$ はバイアス、C はコスト関数です。j 番目のニューロンの出力 a_j は $\sigma(z_j)$ とします。σ はいつもの シグモイドの活性化関数です。$z_j = w_j a_{j-1} + b_j$ はニューロンへの入力に重みを施したものです。コスト関数 C は、ネットワーク全体の出力 a_4 のコストを表すことを強調しておきます。もしネットワークの実際の出力が目標に近かった場合、コストは小さくなり、一方、実際の出力が目標とかけ離れていれば、コストは高くなります。

1個目の隠れ層のニューロンに関連する勾配 $\partial C/\partial b_1$ について私たちは学んでいきます。$\partial C/\partial b_1$ の式を学ぶことで、勾配消失問題がなぜ起きるかを理解したいと思います。

まず $\partial C/\partial b_1$ の式を見てみましょう。とっつきにくく見えますが、実際にはシンプルな構造をしています。すぐに解きほぐしていきます。これが数式です(とりあえず、ネットワークは無視してください。σ' は σ のただの導関数です)

$$\frac{\partial C}{\partial b_1} = \sigma'(z_1) \times w_2 \times \sigma'(z_2) \times w_3 \times \sigma'(z_3) \times w_4 \times \sigma'(z_4) \times \frac{\partial C}{\partial a_4}$$

式の構造は次のようになっています。$\sigma'(z_j)$ の項が各ニューロンに相当する積の中に現れています。重み w_j の項はネットワークの各重みに対応します。そして、最後の項 $\partial C/\partial a_4$ はコスト関数に対応します。上に挙げた項が、それぞれ対応するネットワーク部分の上に置かれていることを確認してください。つまり、ネットワークはそれ自体が、数式の疑似表現になっているのです。

この数式を当たり前にとらえて、勾配消失問題とどう関係するのかの議論へ進んでも構いません。そうして問題はありません。なぜならこの数式は、先の逆伝播の議論における特別な場合であるからです。でも、なぜこの数式が成り立つのかには簡潔な説明が可能であって、その説明を理解するのも面白い(ですし、加えてたぶん知恵もつくはず)です。

バイアス b_1 の微小な変化 Δb_1 を想像してみてください。その変化が、残りのネットワークへカスケード的な変化を引き起こしていくでしょう。まず第一に、1個目の隠れ層のニューロンの出力の変化 Δa_1 を起こします。それが今度は、2個目の隠れ層への重み付き入力の変化 Δz_2 を起こします。さらに、2個目の隠れ層からの出力 Δa_2 も変化します。そうして、出力でのコスト ΔC まではるばる変化していくのです。

$$\frac{\partial C}{\partial b_1} \approx \frac{\Delta C}{\Delta b_1}. \tag{114}$$

このことは、カスケードの各ステップの影響を注意深く辿っていくことで、最終的に $\partial C/\partial b_1$ がわかることを示しています。

これを行うために、Δb_1 がどのように1個目の隠れ層の出力 a_1 を変化させるのか考えてみましょう。$a_1 = \sigma(z_1) = \sigma(w_1 a_0 + b_1)$ という式があるので、以下のようになります。

$$\Delta a_1 \approx \frac{\partial \sigma(w_1 a_0 + b_1)}{\partial b_1} \Delta b_1 \tag{115}$$

$$= \sigma'(z_1) \Delta b_1. \tag{116}$$

$\sigma'(z_1)$ の項は馴染みがあるはずです。勾配 $\partial C/\partial b_1$ の式中の最初の項でした。直感的には、この項はバイアスの変化 Δb_1 を出力の活性の変化 Δa_1 へと変換します。Δa_1 の変化は、今度は重みの付与された入力 $z_2 = w_2 a_1 + b_2$ として2個目の隠れ層へ伝わります。

$$\Delta z_2 \approx \frac{\partial z_2}{\partial a_1} \Delta a_1 \tag{117}$$

$$= w_2 \Delta a_1. \tag{118}$$

Δz_2 と Δa_1 の式を組み合わせると、バイアス b_1 の変化がネットワークをどう伝わって z_2 へ影響を及ぼすのかがわかります。

$$\Delta z_2 \approx \sigma'(z_1) w_2 \Delta b_1. \tag{119}$$

再び馴染みのある式が現れました。勾配 $\partial C/\partial b_1$ の式の最初の2つの項です。

このやり方で進めて、変化が残りのネットワークを伝播するのを追いかけていきます。各ニューロンで $\sigma'(z_j)$ の項と、各重みごとに w_j の項が現れます。最終結果は、最初のバイアスの変化 Δb_1 に対する、最後のコスト関数の変化 ΔC の式となります。

$$\Delta C \approx \sigma'(z_1) w_2 \sigma'(z_2) \ldots \sigma'(z_4) \frac{\partial C}{\partial a_4} \Delta b_1. \tag{120}$$

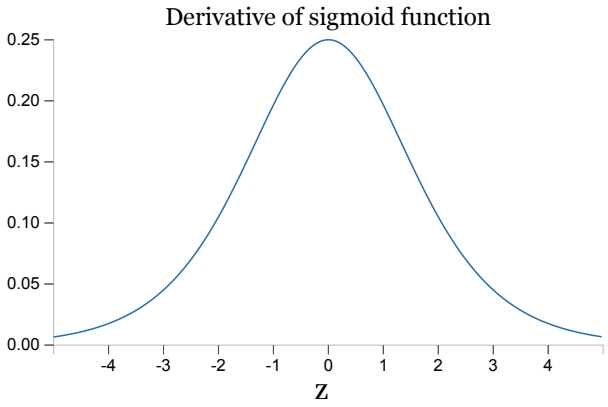
Δb_1 で割ることで、本当に欲しい勾配を手に入れます。

$$\frac{\partial C}{\partial b_1} = \sigma'(z_1) w_2 \sigma'(z_2) \ldots \sigma'(z_4) \frac{\partial C}{\partial a_4}. \tag{121}$$

なぜ勾配消失問題が起きるのか: 勾配消失問題の要因を理解するために、勾配全体の式を書き出してみましょう。

$$\frac{\partial C}{\partial b_1} = \sigma'(z_1)\, w_2 \sigma'(z_2)\, w_3 \sigma'(z_3)\, w_4 \sigma'(z_4)\, \frac{\partial C}{\partial a_4}. \tag{122}$$

最後の項以外は、$w_j\sigma'(z_j)$ の積の項からなります。これらの積の項がどう振る舞うかを理解するために、σ' のプロットを見てみます。



この導関数は $\sigma'(0) = 1/4$ の地点で最大値に達します。ところで、標準的な方法 でネットワークの重みを初期化する場合には、平均 0 かつ標準偏差 1 のガウス分布から重みを選びます。このとき、重みは通常 $|w_j| < 1$ を満たします。これらの考察を組み合わせると、$w_j\sigma'(z_j)$ の項は通常 $|w_j\sigma'(z_j)| < 1/4$ を満足します。そして、そのような項をいくつも掛けていくと、積は指数関数的に減少していく傾向があります。すなわち、項が多くなればなるほど、積は小さくなっていきます。上記は勾配消失問題の説明として妥当な気がします。

この説明をもう少し明確化するために、$\partial C/\partial b_1$ の式を、後方の層のバイアスの勾配 $\partial C/\partial b_3$ と比べてみましょう。もちろん、まだ $\partial C/\partial b_3$ の式は計算してないのですが、$\partial C/\partial b_1$ の上述のパターンと同じになります。こちらが2つの式の比較です。

$$\frac{\partial C}{\partial b_1} = \sigma'(z_1) \overbrace{w_2\sigma'(z_2)}^{<\frac{1}{4}} \overbrace{w_3\sigma'(z_3)}^{<\frac{1}{4}} \underbrace{w_4\sigma'(z_4)\frac{\partial C}{\partial a_4}}$$

$$\updownarrow \text{ common terms}$$

$$\frac{\partial C}{\partial b_3} = \sigma'(z_3) \overbrace{w_4\sigma'(z_4)\frac{\partial C}{\partial a_4}}$$

2つの式は共通の項を多く持っています。しかし、勾配 $\partial C/\partial b_1$ は $w_j\sigma'(z_j)$ の形をした2つの項を余分に含んでいます。上で述べたように、$\sigma'(z_j)$ の項は $1/4$ より明らかに小さいです。なので、通常は勾配 $\partial C/\partial b_1$ は $\partial C/\partial b_3$ よりも 16 倍（以上）小さくなります。これこそが勾配消失問題の本質的な要因なのです。

もちろん、これは大雑把な議論であり、勾配消失問題の発生に関する厳格な証明ではありません。幾つかの例外事項もあります。特に、重み w_j

が訓練中どんどん大きくなる可能性については気になることでしょう。その場合、積の中の $w_j\sigma'(z_j)$ はもはや $|w_j\sigma'(z_j)| < 1/4$ を満たさなくなります。実際にこの項が十分大きくなったとしたら、つまり 1 より大きくなったら、勾配消失問題はなくなるでしょう。代わりに、勾配は層を逆伝播するごとに指数関数的に大きくなっていきます。勾配消失問題の代わりに、勾配爆発問題が起きるのです。

勾配爆発問題: 勾配爆発の起きる典型的な例を見てみましょう。この例は恣意的なもので、勾配爆発が確実に起きるように、ネットワークのパラメータを修正していきます。例として恣意的ではありますが、勾配爆発は仮説上の現象ではなく、実際に十分起きうるものです。

勾配爆発を引き起こすまでには、ステップが2つあります。1つ目は、ネットワークの中の全ての重みを大きくしておくことです。たとえば、$w_1 = w_2 = w_3 = w_4 = 100$ とします。2つ目のステップは、$\sigma'(z_j)$ の項があまり小さくなりすぎないように、バイアスを選ぶことです。実際の操作は簡単です。私たちが行うべきのは、各ニューロンへの重み付けされた入力が $z_j = 0$ を、(そして $\sigma'(z_j) = 1/4$ を) 満たすようにすることです。したがって、たとえば、$z_1 = w_1 a_0 + b_1 = 0$ を得たいとしましょう。私たちは $b_1 = -100 * a_0$ と設定することで上記を実現できます。同じアイデアを、他のバイアスを選ぶときにも使えます。このとき、全ての $w_j\sigma'(z_j)$ の項は $100 * \frac{1}{4} = 25$ に等しいことを確認しておきます。これらの選択により、勾配爆発を起こせるのです。

不安定勾配問題: ここでの根本的な問題は、勾配消失問題でも、勾配爆発問題でもありません。前方の層の勾配が、それ以降の層の勾配の積となっていることが問題なのです。多層の場合、本質的に不安定な状況となります。全部の層がほぼ同じ速度で学ぶ唯一の方法は、それらの項の積をほぼバランスさせることです。バランスさせるための仕組みや要因が何もないとき、偶然には上手く行きようがありません。つまり本当の問題は、ニューラルネットワークが **不安定勾配問題** を伴っていることです。結果的に、勾配を利用する標準の学習テクニックを使うときには、ネットワーク中の異なる層は、かなり異なる速度で学習する傾向があります。

演習

- 勾配消失問題の議論の中で、$|\sigma'(z)| < 1/4$ である事実を利用しました。別の活性化関数を使ったと想定し、その導関数はとても大きくなってしまったとします。その場合、勾配不安定問題は回避できるでしょうか？

勾配消失問題の広がり: これまでに、前方の層での勾配は消失するか爆発する可能性があることを見てきました。実際、シグモイドのニューロンを使ったときには、勾配は通常、消失してしまうでしょう。その理由については、$|w\sigma'(z)|$ の式を再び考えてみてください。勾配消失問題を避けるためには、$|w\sigma'(z)| \geq 1$ とする必要があります。もし w がとても大きければ、上記は達成できると思っているでしょう。しかし実際は、見た目よりずっと難しいのです。その理由は、$\sigma'(z)$ の項は w にも依存していることにあります。すなわち a が活性化した入力とすると、$\sigma'(z) = \sigma'(wa+b)$ となります。だから、w を大きくするときには、同時に $\sigma'(wa+b)$ を小さくしないよう注意する必要があります。これは大きな制約です。なぜかというと、w を大きくすると、$wa+b$ がとても大きく傾向があるからです。σ' のグラフの中では、σ' の「両翼」に相当する確率が高くなります。そこではとても小さな値をとります。これを避ける唯一の方法は、活性化された入力を、かなり狭い範囲の値(この定性的な説明は下記の定量的な説明により明らかです)に落ち着かせることです。たまたま起きることはあります。しかし、大抵は起きません。したがって、一般的には勾配が消失してしまうのです。

問題

- 積 $|w\sigma'(wa+b)|$ を考えます。$|w\sigma'(wa+b)| \geq 1$ とします。**(1)** これが $|w| \geq 4$ の場合にだけ起こることを証明してください。**(2)** $|w| \geq 4$ としたとき、$|w\sigma'(wa+b)| \geq 1$ である入力の活性 a を考えてください。この制約を満たす a の組は、

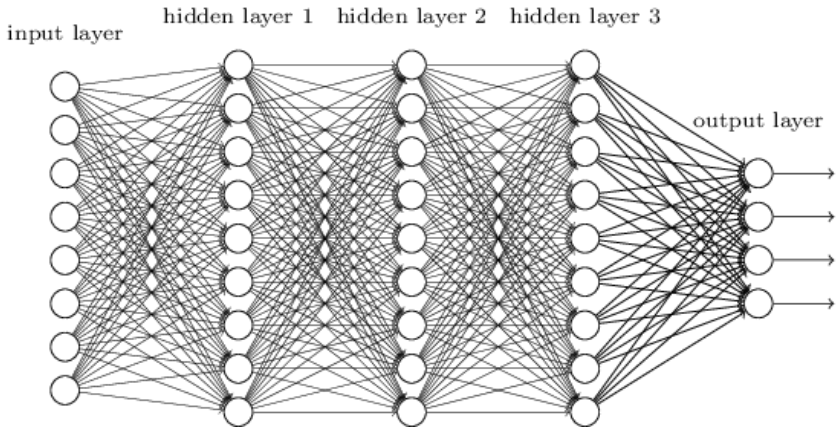
$$\frac{2}{|w|}\ln\left(\frac{|w|(1+\sqrt{1-4/|w|})}{2}-1\right). \tag{123}$$

  の幅より小さいインターバルに分布することを示してください。**(3)** この範囲の幅の境界にある上式は $|w| \approx 6.9$ において最大値 ≈ 0.45 をとることを示してください。完璧に全てが並んだときでさえ、活性化された入力を狭い範囲で持てば、勾配消失問題を避けられるのです。

- **Identity neuron:** 1つの入力 x 、重み w_1 、バイアス b 、出力への重み w_2 を持つニューロンを考えてください。重みとバイアスを適切に選べば、$x \in [0,1]$ に対して $w_2\sigma(w_1x+b) \approx x$ が成り立つことを示してください。そのようなニューロンはidentity neuronとして使えます。それはすなわち、出力が入力と同じとなるニューロンのことです。**ヒント:** $x = 1/2 + \Delta$ **を記述し、** w_1 **が小さいと想定し、** $w_1\Delta$ **のテイラー展開を使ってみてください。**

さらに複雑なネットワークにおける不安定な勾配

これまで、各隠れ層に一つのニューロンだけ持つような、トイネットワークを使って勉強してきました。もっと複雑な深層の、各隠れ層にたくさんのニューロンを持つようなネットワークの場合どうなるのでしょうか？



実際、そのようなネットワークにおいても、同じ振る舞いが見られます。逆伝播の章で見たように、全 L 層のネットワークの l 層目 の勾配は次の式で 与えられます。

$$\delta^l = \Sigma'(z^l)(w^{l+1})^T \Sigma'(z^{l+1})(w^{l+2})^T \ldots \Sigma'(z^L)\nabla_a C \qquad (124)$$

ここで、$\Sigma'(z^l)$ は対角行列で、各成分は、l 層目に対する重み付き入力への $\sigma'(z)$ となります。w^l は異なる層に対する重み行列です。$\nabla_a C$ は活性化された出力に対する C の偏微分ベクトルです。

これは各層に1ニューロンの場合よりも遥かに複雑です。でも、よく見ると、本質的な形状はとてもよく似ています。$(w^j)^T \Sigma'(z^j)$ の項の組がたくさんあります。加えて、行列 $\Sigma'(z^j)$ の対角成分は $\frac{1}{4}$ よりも小さくなります. もし、重み行列 w^j がそこまで大きくない場合、$(w^j)^T \Sigma'(z^l)$ の各項は勾配ベクトルを小さくする働きがあり、勾配消失問題が発生します。一般的に言うと、先の例のように積の中に項が多くなると、勾配が不安定になります。経験的には、シグモイドのネットワークにおいて、前方の層で勾配が急速に消失する現象が顕著に見られます。結果的に、前方の層での学習は遅くなります。学習が遅いことは、偶然でも単なる不都合でもありません。学習のために私たちが採用した方法のもつ根本的な問題なのです。

ディープラーニングに関する他の問題

この章では、ディープラーニングの問題として勾配消失(や、もっと一般的には勾配の不安定性)に焦点を当ててきました。勾配の不安定性は

重要で本質的な問題ではありますが、実際にはディープラーニングの一つの問題に過ぎません。現在も多くの研究が、深層ネットワークの訓練時に発生する難問を理解しようと試みています。ここで包括的にまとめることはしませんが、いくつかの論文を簡単に紹介します。人々が解決したいと思っている問題の雰囲気をあなたに届けたいと思います。

1つ目の例は、2010年にGlorotとBengio* が、シグモイドを活性化関数に使うと、深いネットワークでの訓練時に問題が生じる証拠を発見した論文です。特に、シグモイドが最後の隠れ層の活性化において、訓練の初期にほぼ 0 に飽和して、実質的に学習を遅くさせる証拠を見つけています。彼らは、この飽和の問題の発生しないような、代わりの活性化関数を提案しました。

*Understanding the difficulty of training deep feedforward neural networks, by Xavier Glorot and Yoshua Bengio (2010)。シグモイドの利用に関する先の議論である、Efficient BackProp, by Yann LeCun, Léon Bottou, Genevieve Orr and Klaus-Robert Müller (1998)も確認してください。

2つ目の例として、2013年にSutskever、Martens、Dahl、Hinton* を挙げます。彼らはランダムな重みの初期化と、momentumに基づいた確率的勾配降下法のmomentumのスケジューリングによるディープラーニングへの影響を調べました。どちらの場合にも、パラメータをうまく選択することで、深層ネットワークの訓練が成功しました。

*On the importance of initialization and momentum in deep learning, by Ilya Sutskever, James Martens, George Dahl and Geoffrey Hinton (2013)。

これらの例が示唆するのは、「深いネットワークを訓練するときの難しさは何か」というのは複雑な問いであるということです。この章では、深いネットワークにおいて、勾配を利用する学習を行うときの不安定性に焦点を当ててきました。直前の2つの段落での結果は、活性化関数の選択や、重みの初期化の仕方、さらに勾配降下の学習方法の実装によっても、学習の効果が変わってくることを示すものでした。そしてもちろん、ネットワークの構造や他のハイパーパラメータの選択も重要です。したがって、多くの要素が深いネットワークの訓練を難しくさせうるのです。そしてこれらを全て理解するために、研究が現在も進められているのです。このことは、暗澹としていて悲観的であるように思えます。しかし、良い知らせもあります。私たちは、次の章でこれらの問題をひっくり返します。ある程度はこれらの難問を克服もしくは回避するような、ディープラーニングのアプローチを開発します。

In academic work, please cite this book as: Michael A. Nielsen, "Neural Networks and Deep Learning", Determination Press, 2015

Last update: Sun Dec 21 14:49:05 2014

CHAPTER 6

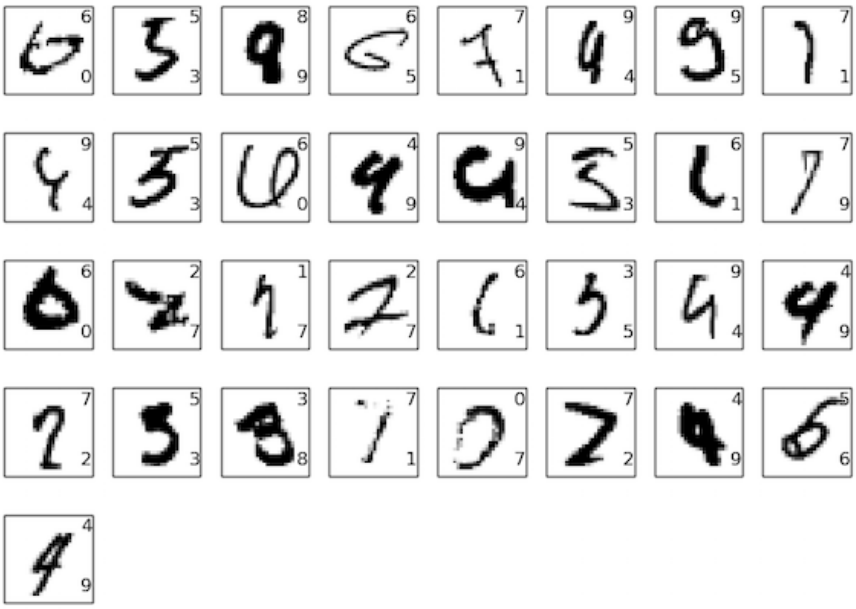
ディープラーニング

前章で、深いニューラルネットワークを訓練するのは、浅いネットワークを訓練する場合よりもずっと難しいことを学びました。これは悲しいことです。なぜなら、**もし**深いネットワークを上手く訓練できれば、浅いネットワークよりも遥かに強力になるからです。前章の知らせは残念ですが、私たちは歩みを止めません。この章では、深層ネットワークの訓練に使えるテクニックを発展させ、実践的な課題へ適用していきます。また、深層ネットワークの幅広い応用例として、画像認識や音声認識などに関する最新結果を簡単に紹介します。そして、ニューラルネットワークや人工知能の未来に何が待っているのか、についても予測していきます。

この章はとても長いです。章の全体を軽く案内しましょう。各セクション間の繋がりは強くはありません。ですので、ニューラルネットワークに既に少し馴染みがあるなら、興味のあるセクションへ先に飛ぶのもよいでしょう。

この章の主要なテーマは、幅広く使われている深層ネットワークの一種である、深層畳み込みネットワークの紹介です。MNISTの手書き数字の分類問題に対して、畳み込みネットワークを駆使して挑んで行く様子を、コードを交えながら詳細に追っていきます。

畳み込みネットワークの説明に際しては、本書で扱ってきた浅いネットワークと比較しながら進めます。試行錯誤により、ネットワークをいじりながら作り上げていきます。その過程でたくさんの強力なテクニックを学ぶでしょう。畳み込み、プーリング、(浅いネットワークの場合よりも効率的に訓練するための)GPUの使用、(過適合の抑制のための)訓練データの拡張、(過適合の抑制のための)ドロップアウト、ネットワークのアンサンブルなどを学んでいきます。これらを組み合わせると最終的に、人間に近いパフォーマンスを出せるシステムが出来上がります。10,000個のMNISTテスト画像(これらは訓練画像には含まれません！)から、9,967個を正しく分類できるようになります。誤って分類した33個の画像をちらっと見てみましょう。正しい分類が手書き数字の右上に表示されており、私たちのプログラムが分類した結果が右下に表示されています。



著:Michael Nielsen / 2014年9月-12月
訳:「ニューラルネットワークと深層学習」翻訳プロジェクト

人間でも正しく分類するのは難しい画像が多いです。例えば、上の行の3つ目の画像を見てください。正解は"8"ですが、私には"8"よりは"9"に見えます。私たちのネットワークも"9"と判断しています。この類の"間違い"は許容できるもので、むしろこの間違いを犯せるのは立派ではないかとさえ思えます。画像認識の議論のまとめとして、畳み込みネットワークを使用した最近の目まぐるしい研究成果を調査します。

この章の最後では、大局的にディープラーニングを概観します。具体的には、再帰型ニューラルネットワーク(RNN)や長期短期記憶(LSTM)ユニットなどの他のニューラルネットワークのモデルを簡単に調査して、そのようなモデルが音声認識や自然言語処理や他の分野の問題にどう適用可能であるのか考察します。そして、意思だけで使えるユーザインターフェイスから、人工知能におけるディープラーニングの役割まで幅広く扱い、ニューラルネットワークとディープラーニングの未来を予測します。

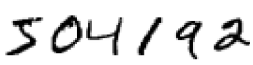
この章は、本書のこれまでの章の内容の集大成です。例えば、逆伝播、正規化、ソフトマックス関数などのアイデアを組み合わせてネットワークに利用します。ですがこの章を読むのに、前章までを詳細に読み切る必要はありません。ただし1章を読んで、ニューラルネットワークの基礎を抑えておくのは役に立つでしょう。2章から5章までのアイデアを利用するときには、必要であれば飛べるようにリンクを表示します。

この章では扱わないトピックを明確化しておきます。まず、ニューラルネットワークの素晴らしい最新ライブラリのチュートリアルではありません。また、数十の深層ネットワークを、巨大な計算機を使って訓練することで最先端の問題を解く、というようなトピックも扱いません。それよりむしろ、深層ニューラルネットワークの根源的な原理を理解することに焦点を当て、

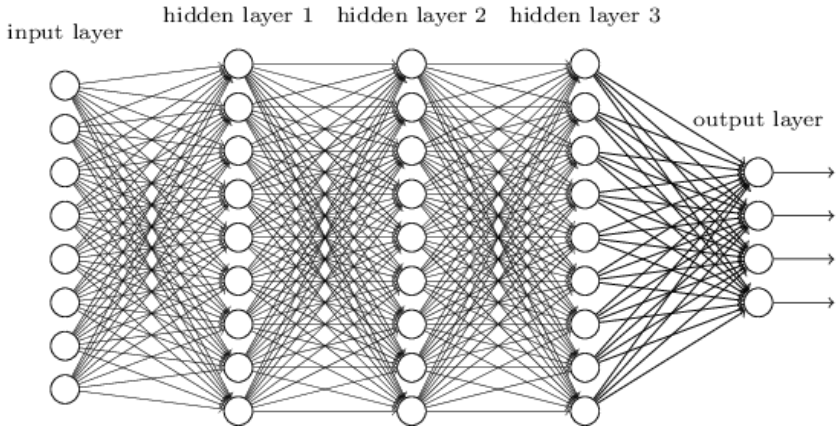
その原理をシンプルで理解しやすいMNISTの問題に適用します。繰り返すと、この章を読んでも最先端の流行には辿り着けません。しかし、この章と以前の全章を読むことであなたは深層ニューラルネットワークの本質に触れることができ、いずれは現在の流行を理解できるようになるでしょう。

畳み込みニューラルネットワークの導入

以前の章では、ニューラルネットワークの、手書き数字画像の認識精度を向上することを目的としていました。



この目的のために、隣接層がお互いに全結合するネットワークを使用していました。これは、ネットワーク内の全てのニューロンが、隣接する層の全てのニューロンと結合している状態です。



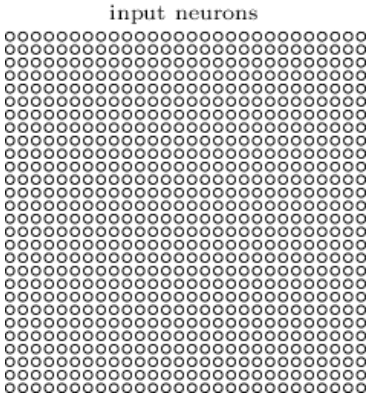
さらに、入力画像の各ピクセルの強度を、入力層内の対応するニューロンへエンコードしていました。これはつまり、使用した 28×28 ピクセルの画像の場合、ネットワークは 784 $(= 28 \times 28)$ の入力ニューロンを持つということです。そして、ネットワークの重みとバイアスを訓練することで、入力画像が'0', '1','2', ..., '8', '9'のいずれであるかを、ネットワークの出力から(望みどおり！)特定しようとしました。

これまで作ってきたネットワークは上手く動いていました。MNISTの手書き数字データセットから取り出した訓練データとテストデータを使って、98%以上の分類精度を達成していたのを覚えているでしょうか。しかしよく考えると、画像を分類するのに全結合層からなるネットワークを使うのは変です。なぜかというと、全結合層からなるネットワークは、画像の空間的な構造を考慮していないからです。たとえば全結合層では、入力ピクセルの中の離れたところにあるもの同士と、近いところにあるもの同士が全く同等に扱われます。そのような空間的な構造の概念自体、全結合層

のネットワークの場合、全て訓練データから推測されなければなりません。しかし、**真っさらな状態**のネットワーク構造からスタートする代わりに、空間構造を活用したネットワーク構造を使ったとしたらどうでしょう？このセクションでは、**畳み込みニューラルネットワーク**を扱います。畳み込みネットワークは、空間的構造性を考慮した設計となっており、画像分類に非常に適しています。この特性により、畳込みネットワークは効率的に学習できるのです。つまり、画像を分類するのに優れた、深くて多層のネットワークを訓練できると言えます。今日、深層の畳み込みネットワークもしくは類似のネットワークが、画像認識を目的とするニューラルネットワークの大半に使われています。

畳み込みニューラルネットワークは次の3つの重要なアイデアを使っています。それは、**局所受容野**、**重み共有**、**プーリング**です。各アイデアを順に見ていきましょう。

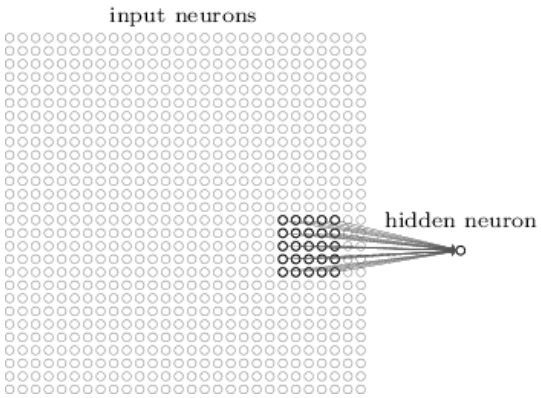
局所受容野: 上で確認した全結合層では、入力は一列のニューロンとして描かれていました。一方、畳み込みネットワークでは、入力を 28×28 の正方形のニューロンと考えます。各ニューロンの値は、入力として用いる 28×28 の入力ピクセルの強さに対応します。



いつも通り、入力のピクセルを隠れ層のニューロンに結合します。しかし、隠れ層の各ニューロンへ、各入力ピクセルを結びつけることはしません。代わりに、各ニューロンへ入力画像の中の小さい局所領域を結合します。

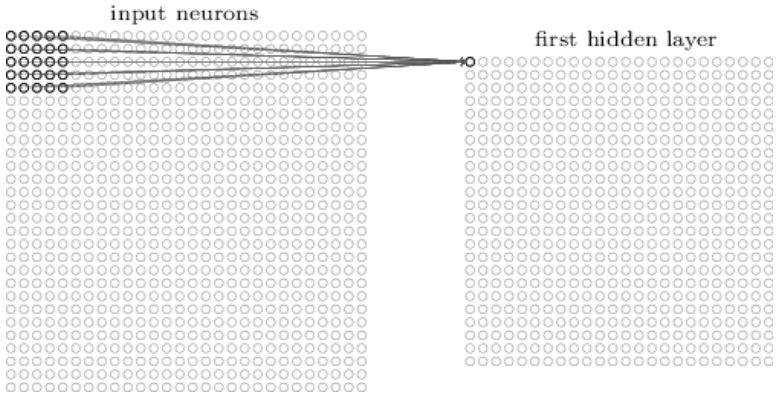
正確に言うと、1つ目の隠れ層の各ニューロンは、入力ニューロンの小さい領域と結合します。例えば、入力の 25 ピクセルに対応する 5×5 の領域に、隠れ層の各ニューロンが結合します。ある隠れニューロン結合を示すと次のようになります。

*畳み込みニューラルネットワークのはじまりは、1970年代まで遡ります。しかし、近年の畳み込みニューラルネットワークブームの発端の論文は1998年のby Yann LeCun, Léon Bottou, Yoshua Bengio, Patrick Haffnerによる"Gradient-based learning applied to document recognition"です。LeCunは畳み込みネットワークの用語定義に関して興味深い次のような発言を残しています。「畳み込みネットワークのモデルは、生物学の神経モデルからはあまり着想を得ていません。そのため私は、"畳み込みニューラルネットワーク"ではなく"畳み込みネットワーク"と呼んでいます。そして、ノードに対しても"ニューロン"ではなく"ユニット"と呼んでいます」と。この発言に反して、畳み込みネットワークは、私たちが学んできたニューラルネットワークと同じアイデアを多く利用しています。例えば、逆伝播や勾配降下、正規化、非線形な活性化関数などです。なので、慣例に従い、これからはニューラルネットワークの1種とみなします。"畳み込みニューラルネットワーク"という用語と"畳込みネットワーク"という用語を同じ意味で使っていきます。"(人工)ニューロン"や"ユニット"という用語も同義として使っていきます。

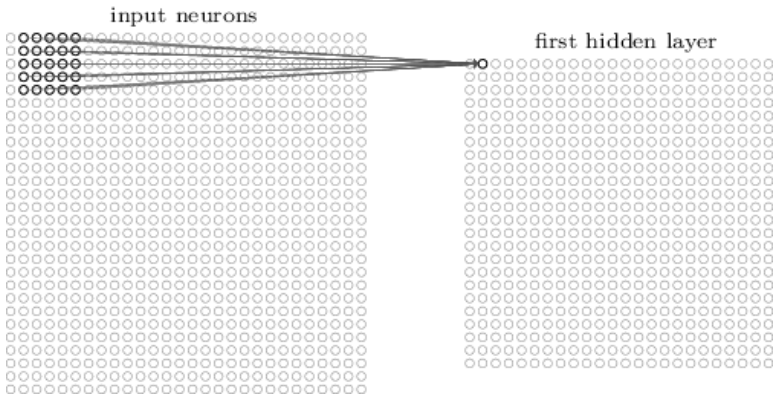


入力画像内のそのような領域は、隠れニューロンの**局所受容野**と呼ばれます。入力ピクセル上の小さな窓のようなものです。各結合は重みを学習します。隠れニューロンはバイアスも同じく学習します。特定の隠れニューロンは、特定の局所受容野を分析しているとみなせます。

そして、入力画像全体をカバーするように局所受容野をスライドさせます。局所受容野ごとに、1つ目の隠れ層の中で異なる隠れニューロンが割り当てられます。これを具体的に確認してみます。左上の角の局所受容野から始めてみましょう。



1ピクセル分（すなわち1ニューロン分）局所受容野を右へスライドして、次は2つ目の隠れニューロンの結合を考えます。



これを繰り返して、最初の隠れ層の全体に対して値を設定します。入力画像のサイズが 28×28 で、局所受容野のサイズが 5×5 の場合、1つ目の隠れ層のニューロンは 24×24 個となります。その理由は、入力画

像の右側（もしくは下側）にぶつかるまでに、23 個のニューロン分だけ局所受容野をスライドできるからです。

ここまで、1 ピクセルずつ局所受容野が移動する例を見てきました。実は、**ストライドの長さ**に 1 以外の値が使われることがしばしばあります。たとえば、2 ピクセルずつ局所受容野を右へ（もしくは下へ）動かすこともあるでしょう。これは、2 ピクセルのストライド長さが使われていると言えます。この章では大抵、ストライドの長さが 1 の場合しか扱いませんが、時には異なるストライド長さ*が使用されて実験が行われる場合もあることは知っておいてください。

*以前の章で触れたように、異なるストライド長さを試したい場合、最適なパフォーマンスを発揮するストライド長さを決めるには、検証データを使うのがよいでしょう。詳細は、ニューラルネットワークでハイパーパラメータを選ぶ方法についての以前の議論を確認してください。同じアプローチが局所受容野のサイズを選ぶ時にも使われるでしょう。もちろん、5×5 の局所受容野を使う特別な理由はないのです。一般的には、入力画像が 28×28 のMNIST画像よりもずっと大きい時には、大きいサイズの局所受容野を使って方が良い傾向があります。

重みとバイアスの共有: 各隠れニューロンはバイアスと局所受容野に結合された 5×5 の重みを持つことを上で述べました。しかし、24×24 の全ての隠れニューロンに対して、**同じ**重みとバイアスを適用することをまだ伝えていませんでした。これはつまり、j, k 個目の隠れニューロンへの出力は、下記のようになることを示しています。

$$\sigma\left(b + \sum_{l=0}^{4} \sum_{m=0}^{4} w_{l,m} a_{j+l,k+m}\right). \tag{125}$$

σ は活性化関数の一種であり、以前の章で使用したシグモイド関数です。b はバイアスの共有値です。$w_{l,m}$ は共有重みであり、そのサイズは 5×5 です。そして、$a_{x,y}$ は x, y における活性化された入力を示します。

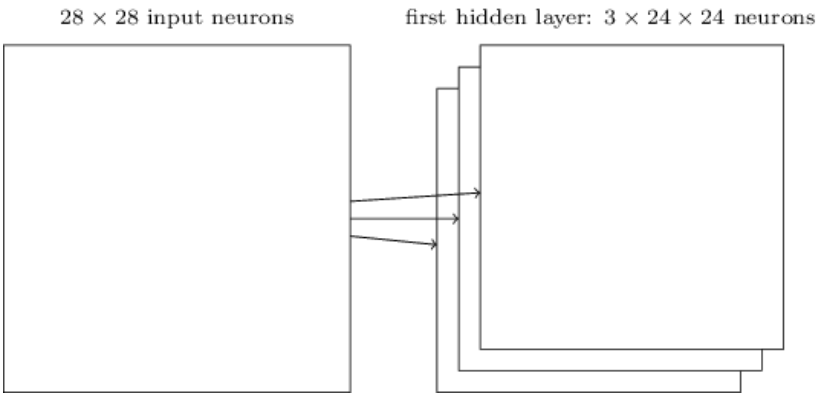
このことが意味するのは、入力画像の異なる位置の全く同じ特徴*を、1層目の隠れ層が検知するということです。なぜこれが成り立つのかを理解するために、隠れニューロンが縦のエッジを局所受容野に検知できるような、重みやバイアスを想定してみてください。この検知能力は、画像の他の位置でも有効に使えそうです。したがって、同じ特徴検出器を画像の全位置へ適用するのは有効と言えるのです。少し抽象的に表現すると、畳込みニューラルネットワークは画像に対して並進不変性があると言います。並進不変性とは例えば、猫の絵を少し並進移動しても、それはまだ猫の絵と言えるような性質のことです* 。

*まだ特徴の厳密な定義をしていませんでした。砕けた言い方をすると、隠れニューロンに検知される特徴とは、ニューロンを活性化する入力パターンの種類を意味します。そのパターンとは例えば、画像のエッジだったり他の形状だったりします。

*私たちが勉強してきたMNIST手書き数字の分類問題では、画像は中央に寄っており、大きさが正規化されていました。なので、MNISTは世の中にある画像よりも、並進不変性が小さいと言える。しかしそれでも、エッジや角といった特徴は入力空間全体において有効に使えるでしょう

この理由から、入力層から隠れ層への写像を**特徴マップ**と呼ぶこともあります。特徴マップを定義する重みを**共有重み**と呼びます。また、特徴マップを同じように定義するバイアスを、同じように**共有バイアス**と呼びます。共有重みと共有バイアスはしばしば、**カーネル**、もしくは**フィルタ**と呼ばれます。文献では、これらの用語を少し異なる使い方をする場合があります。そのため、私は用語の正確性は放棄しています。むしろ具体例で確認することのほうが本質的なので、そのように心がけていきます。

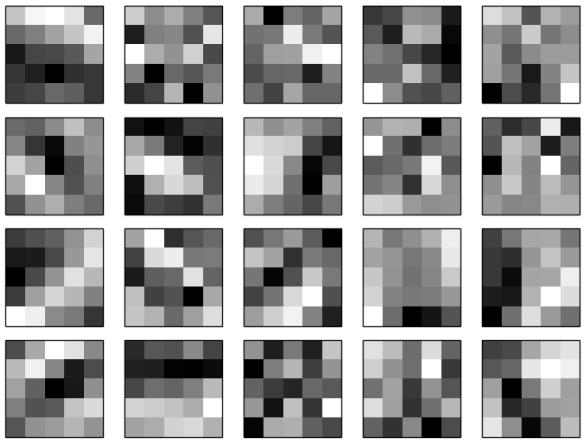
これまで扱ってきたネットワーク構造は、一種類の局所的特徴のみ検知できるものでした。画像認識のためには、二つ以上の特徴マップが必要となります。したがって、完全な畳込み層は幾つかの異なる特徴マップから構成されるのです。



上の例には 三つの特徴マップがあります。それぞれの特徴マップは 5×5 の共有重みと共有バイアスで定義されています。その結果、三種の異なる特徴を、画像全体に渡って検知できるのです。

図をシンプルにするために、3 つの特徴マップだけを見せました。しかし、実際の畳み込みネットワークは（たぶん遥かに）多くの特徴マップを使っているかもしれません。初期の畳み込みネットワークであるLeNet-5は、MNISTの手書き数字を認識するために 6 つの特徴マップを使っていました。各特徴マップは 5×5 の局所受容野と結合しています。したがって、上記の例はLeNet-5と実際かなり近いのです。章の後ろの方で開発するネットワークでは、20 と 40 の特徴マップをそれぞれ持つ畳み込み層を使っています。その特徴マップを少し覗き見してみましょう*

*図の特徴マップは私たちが訓練する最後の畳み込みネットワークに含まれるものです。ここを確認してください。



20 個の画像は 20 個の異なる特徴マップ（もしくはフィルタかカーネル）に対応しています。各マップは、5×5 ブロック画像で表され、局所受容野の 5×5 の重みに対応しています。白いブロックは小さい（概してマイナスの）重みを意味し、その特徴マップは入力ピクセルに反応しにくいという性質を持ちます。一方、黒いブロックは大きい重みを意味し、その特

徴マップは入力ピクセルによく反応します。大まかに言うと、畳み込み層がどのような種類の特徴に反応するかを、上の画像群は示しています。

これらの特徴マップからどんな結論を導けるでしょうか？ 何らかの空間的構造が特徴マップには現れているようです。特徴マップの多くは明るい領域と暗い領域の両方を含んでいます。これにより、空間的構造に関する特徴をネットワークが学習していることが分かります。しかし、これらの特徴検出器が何を学んでいるのかを、それ以上に深く把握するのは難しいです。明らかに、この特徴は画像認識の伝統的なアプローチでたくさん採用されてきたガボールフィルタではありません。畳み込みネットワークの学習した特徴を、突き詰めて理解しようとする研究が、現在多数進行中です。もし興味があるのなら、2013年のMatthew ZeilerとRob FergusによるVisualizing and Understanding Convolutional Networksを読むのを薦めます。

重みとバイアスを共有する大きな利点は、畳込みネットワークのパラメータ数を大きく減らせる点です。上記の畳み込みネットワークのパラメータを数えてみます。特徴マップごとに $25 = 5 \times 5$ の共有重みと、1つの共有バイアスを必要とします。つまり、各特徴マップは 26 のパラメータが必要です。20 個の特徴マップがある場合には、畳込み層を定義するための全パラメータは $20 \times 26 = 520$ 個となります。それと比較するために、全結合層のパラメータ数を数えてみます。これまで本書で使ってきた例と同様に $784 = 28 \times 28$ 個のニューロンからなる入力層と、30 個の隠れニューロンからなる全結合層で構成されるネットワークを仮定してみてください。その場合、784×30 個の重みとさらに 30 個のバイアスで、全部で $23,550$ 個のパラメータからなります。つまり、全結合層は畳み込み層と比べて 40 倍のパラメータを保持することになるのです。

もちろん、根本的に2つのモデルは異なっているため、パラメータ数を直接比較することは本当はできません。しかし直感的に考えてみても、畳み込み層には並進不変の性質があるので、全結合層のモデルと同じパフォーマンスを得るのに要するパラメータ数は少なくなると思えます。パラメータ数が小さいおかげで、畳込みモデルは高速に訓練でき、層を深くできるのです。

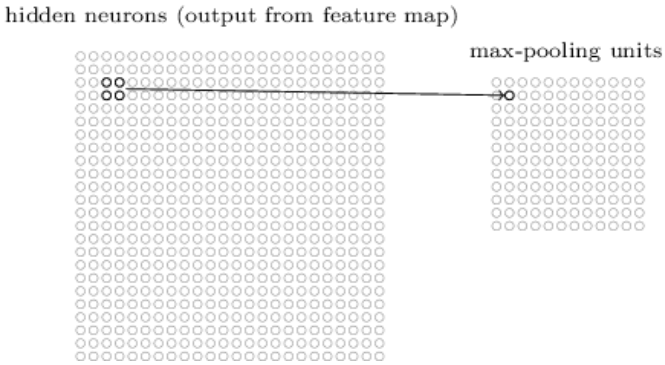
ところで、**畳み込み**という呼び名は(125)の、**畳み込み**という名で知られた操作に由来しています。正確に式を記述すると、$a^1 = \sigma(b + w * a^0)$ となります。ここで、a^1 はある特徴マップからの活性化された出力、a^0 は活性化された入力、$*$ は畳み込みと呼ばれる操作を示します。私たちは、いわゆる数学における畳込み操作を深く追い求めません。なので、

数学との結びつきを心配する必要はありません。しかし、由来が何なのかは少なくとも知っておいて損はありません。

プーリング層: 通常の畳み込みニューラルネットワークは、先ほどの畳み込み層に加えて、**プーリング層**も含みます。このプーリング層は通常、畳込み層の直後に置かれます。この層の役割は畳込み層の出力を単純化することです。

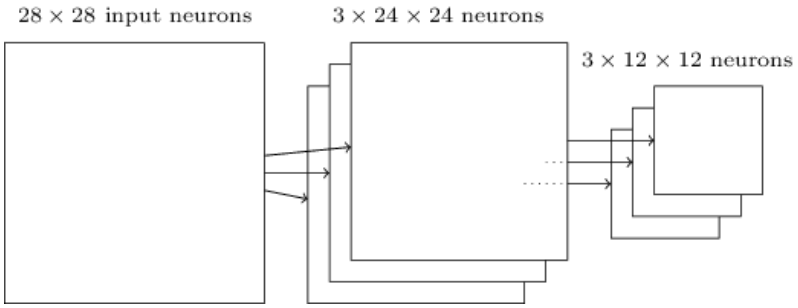
少し詳しく説明すると、プーリング層は畳み込み層から各特徴マップ*を取得し、特徴マップを濃縮させています。つまり、プーリング層の各ユニットは前層の 2×2 の領域のニューロンをまとめます。具体的な手法を紹介すると、プーリングのよく知られた例として**Maxプーリング**があります。Maxプーリングでは、プーリングのユニットは 2×2 の入力領域のうちで最大の値を単純に出力します。次の図を見てください。

*ここでの用語の定義ははっきりしていません。私は「特徴マップ」という言葉を、畳み込み層から算出される関数を指して使うのではなく、活性化された出力ニューロンのことを指して使います。この類の意味の揺れは、研究論文によくあることです。



畳み込み層の出力ニューロンは 24×24 なので、プーリング処理後は 12×12 のサイズのニューロンとなります。

上で述べた通り、畳み込み層は通常1つ以上の特徴マップを持ちます。それらの特徴マップに対して個別にMaxプーリングを適用します。したがって、3つ特徴マップがある場合には、畳込み層とMaxプーリング層の様子は次のようになります。

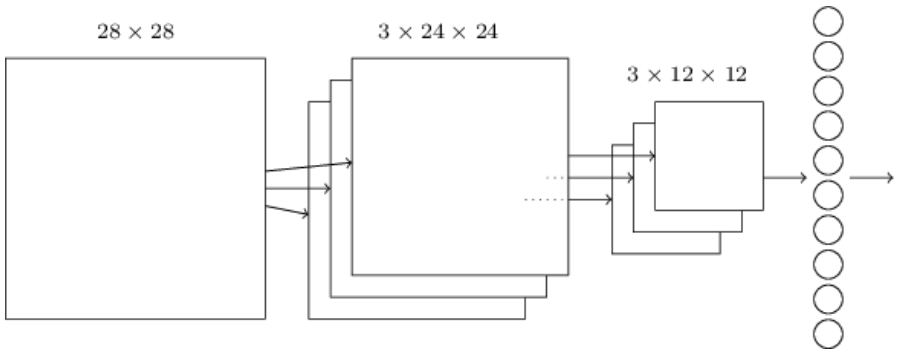


Maxプーリングは、画像のある領域内のどこかに指定の特徴があるかをネットワークが確認する手段とみなせます。つまり正確な位置の情報は棄てているのです。直感的に解釈すると、一度特徴が見つかればその正確な位置は重要でなく、他の特徴に対するおおよその位置さえ分かれ

ばよいということなのです。この手法の大きな利点は、特徴はプーリングされると少なくなるため、後方の層で必要なパラメータを減らすことができる点です。

プーリングの手法はMaxプーリングだけではありません。他のよく知られたアプローチとして**L2 プーリング**があります。L2プーリングの手法では、2×2 領域のニューロンの活性化出力の最大値をとるのではなく、2×2 領域の活性化出力の和の平方根をとります。L2プーリングは、畳み込み層からの情報を圧縮する方法とも言えます。詳細な手続きは異なるものの、直感的にはMaxプーリングに近いはたらきをします。実際、どちらの手法も広く使われてきました。そして、場合によってはさらに別のプーリング手法も使われることもあります。パフォーマンスを本気で良くしようと思ったら、検証データを使って異なるプーリング手法を試すのが良いでしょう。そして、一番良い手法を選択するのです。しかし私たちは、細かい最適化の種類を気にかけるつもりはありません。

全てを1つにまとめる: さあ、これまでのアイデアを全て使って、畳み込みニューラルネットワークを完成させましょう。これから作るものは、上で見てきた構造と似ていますが、10 のニューロンを持つ出力層が追加されています。この層の各ニューロンはMNISTの 10 種の手書き数字 ('0', '1', '2', **etc**) に対応するものです。



このネットワークは、MNIST画像のピクセル強度を符号化するのに使われる 28×28 の入力ニューロンから始まります。そして、5×5 の局所受容野と 3 の特徴マップを使う畳み込み層が続きます。この畳み込み層は $3 \times 24 \times 24$ の隠れ特徴ニューロンから構成されます。次にMaxプーリング層が続きます。この層では 2×2 の領域を 3 の特徴マップごとに処理します。つまりプーリング層は $3 \times 12 \times 12$ の隠れ特徴ニューロンからなります。

ネットワークの最後の層は全結合層です。Maxプーリング層の**全ての**ニューロンとこの層の 10 の出力ニューロンが個別に結合します。この全結合の構造は以前の章で扱ったものと同じです。しかし、上図では表記を

シンプルにするため、全ての結合を表示する代わりに1つの矢印で表現しています。結合の様子は容易に想像できるでしょう。

この畳込み構造は、以前までの章で扱ってきた構造と大きく異なります。しかし、全体像は似ています。ネットワークは単純なユニットから構成され、各ユニットの振る舞いは重みとバイアスから決定されます。全体の目標も同じです。それは、訓練データによりネットワークの重みとバイアスを訓練して、ネットワークが入力画像を上手く分類できるようにすることです。

また、以前の章と同じようにネットワークの訓練には、確率的勾配降下法と逆伝播を用います。以前の章と殆ど同じです。しかし、逆伝播の手続きには、少し修正を加える必要があります。以前の章の逆伝播による偏微分導出では、全結合層を対象とした手続きを扱っていたためです。幸運なことに、畳込み層とMaxプーリング層の偏微分の式を導出するには、修正を少し加えるだけで済みます。もし詳細を理解したければ、次の問題に取り組んだほうがよいでしょう。ただし、逆伝播にる前方の層の偏微分を正しく理解していない限り、この問題を解くには少し時間がかかります。

問題

- **畳み込みネットワークにおける逆伝播:** 全結合層のネットワークにおける逆伝播の重要な式は (BP1)-(BP4) (link)でした。上述のネットワークのように、畳み込み層、Maxプーリング層、全結合の出力層から構成されるネットワークを想定してください。この時、逆伝播の式はどのように修正されるでしょうか？

畳み込みニューラルネットワークの実際

畳み込みニューラルネットワークの核となるアイデアをこれまで確認してきました。実際にそれらがどう作用するのかを、畳込みネットワークを実装し、MNISTの数字分類問題へ適用することで確認してみましょう。今回、私たちが使うプログラムは`network3.py`です。これは以前の章で使った`network.py`と`network2.py`の改良版です* 。コードを参照したい場合、GitHubからコードを取得できます。次のセクションでは`network3.py`を作り上げていきます。一方、このセクションでは、`network3.py`を畳み込みネットワークを構築するためのライブラリとして使います。

*`network3.py`はTheanoライブラリの畳み込みニューラルネットワークのドキュメントからアイデアを取り込んでいることにも注意してください (特にLeNet-5の実装部分) 。また、Misha Denilのドロップアウトの実装や、Chris Olahのアイデアも参照しています。

`network.py`と`network2.py`はPythonと行列ライブラリであるNumpyを使って実装されていました。その際、ニューラルネットの原理や逆伝播、確率的勾配降下法などを学ぶために、各実装をスクラッチから行いました。しか

し、これらの詳細を私たちは既に理解しているため、network3.pyではTheano*という機械学習ライブラリを利用します 。Theanoを使うことで、畳み込みニューラルネットワークでの逆伝播を簡単に実装できます。それは全ての結合での計算を自動的に行ってくれるためです。さらにTheanoは私たちの以前のコード(こちらは速度よりも理解しやすさを重視して記述されています)よりも高速であり 、複雑なネットワークを訓練するにはとても実用的です。Theanoのさらに別の利点は、1つのコードをCPU上でもGPU上でも実行できることです。GPU実行により高速化が実現されるため、複雑なネットワークの利用に実用性が生まれます。

*2010年のJames Bergstra, Olivier Breuleux, Frederic Bastien, Pascal Lamblin, Ravzan Pascanu, Guillaume Desjardins, Joseph Turian, David Warde-Farley, and Yoshua Bengioによる Theano: A CPU and GPU Math Expression Compiler in Pythonを確認してください。Theanoは、人気のあるPylearn2や Kerasなどのニューラルネットワークライブラリにも利用されています。この本の執筆中の現在、他の人気のニューラルネットライブラリには、CaffeとTorchがあります

コードを追いたい場合には、あなたのシステムでTheanoを実行する必要があります。Theanoのインストールは、プロジェクトページの指示に従い行ってください。以降のコードはTheano 0.6*で実行を確かめています。コードの一部はGPUなしのMac OS X Yosemiteで実行しました。また、一部はNVIDIAのGPUありのUbuntu 14.04の環境で実行しました。幾つかの実験はどちらの環境でも試しました。network3.pyの実行の際は、network3.pyのコード中のGPUフラグをTrueかFalseのどちらか適当な方に設定する必要があります。Theanoを起動してGPU上で動かす際には、このインストラクションが役立ちます。ウェブ上のチュートリアルもGoogle検索により簡単に見つかります。きっと、あなたの助けになるでしょう。ローカル環境でGPUを利用できない場合、Amazon Web ServicesのEC2インスタンスやG2インスタンスを試すとよいでしょう。ただしGPUを使ったとしても、処理に時間がかかるコードがあることに注意してください。実験の多くは数分から数時間かかります。一番複雑な実験は、CPUだと数日かかるでしょう。以前の章でお薦めしたように、コードを実行している間に本書を読み進めて、時たま出力結果を確認するのが良いと思います。CPU上で複雑な実験を行う際には、訓練のエポック数を小さくするか、実験そのものを諦めた方がよいでしょう。

*この章を公開時には、Theanoの最新バージョンは0.7に更新されていました。Theano 0.7でも同じコードを実行してみたところ、本書の記載結果とほぼ同等の結果が得られました。

準備運動として、100のニューロンを含む隠れ層を1つだけを持つ浅いネットワークから始めてみましょう。訓練のエポック数は 60 、学習率は $\eta = 0.1$、ミニバッチサイズは 10、正規化なしの条件で実行してみます*。

```
>>> import network3
>>> from network3 import Network
>>> from network3 import ConvPoolLayer, FullyConnectedLayer, SoftmaxLayer
>>> training_data, validation_data, test_data = network3.load_data_shared()
>>> mini_batch_size = 10
>>> net = Network([
        FullyConnectedLayer(n_in=784, n_out=100),
        SoftmaxLayer(n_in=100, n_out=10)], mini_batch_size)
>>> net.SGD(training_data, 60, mini_batch_size, 0.1,
            validation_data, test_data)
```

*このセクションの実験用コードは このスクリプトの中にあります。スクリプト中のコードは、このセクションの議論に単純に沿っていることに注意してください。

セクションの中では、訓練のエポック数を指定していることにも注意してください。これは、訓練の様子を明らかにするために行っています。実際には、早期打ち切りのテクニックが有効です。早期打ち切りは、検証データごとに精度を調査して、精度がそれ以上向上しなくなった段階で訓練を打ち切る方法でした。

$97.80%の分類精度を得ました。この結果は、**検証データ**を使って、最高の分類精度を発揮する訓練エポック数を探し、その訓練エポック数を**テストデータ**に適用した時の精度です。検証データを使って精度評価のタイミングを決定することで、テストデータへ過適合を防ぎます。(検証データの使用に関する前章での議論を確認してください) このやり方を今後も踏襲します。なお、ネットワークの重みとバイアスはランダムに初期化される*ため、あなたの結果は少し異なるかもしれません。

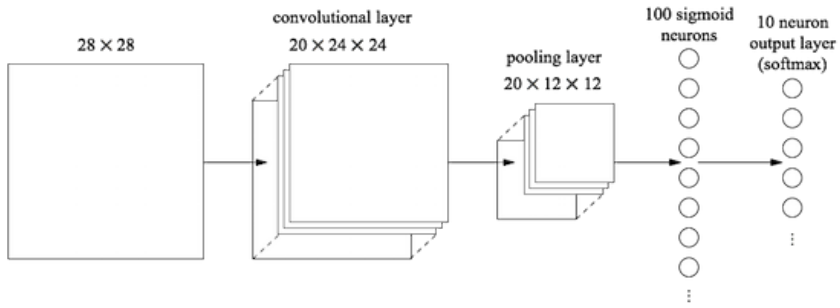
*私は、この実験で同じネットワークに対し3回実行しました。検証データによる精度が3つのうちで最も良かった条件下での、テストデータによる結果を報告しました。複数回実行することで結果のばらつきを小さくできます。私たちが行っているように、多くの構造を比較する際にはこの方法は便利です。以降では明記していない限り、この手続きをとっています。実際には、この手続きを行っても、あまり結果に違いは生まれません。

この 97.80 %という分類精度は、3章で得た 98.04 という結果に近いです。3章でも似たネットワーク構造を使って、ハイパーパラメータを学んでいました。どちらの例も、100 個のニューロンからなる隠れ層を1つ持つ浅いネットワークを使っています。さらにどちらも、訓練エポック数 60、ミニバッチサイズ 10、学習率 $\eta = 0.1$ の条件で訓練を行っています。

しかしネットワーク前方において、異なる点が2つあります。1つ目は、3章のネットワークでは、前方において正規化を行っていたことです。これにより過適合を防いでいました。一方、この章のネットワークに対して同様に正規化を施すと、精度は向上しますが、その上がり幅はわずかです。そのため、私たちは正規化に関しては終盤まで気にしないこととします。2つ目は、ネットワーク前方の最後の層は、活性化関数としてシグモイドと、誤差関数として交差エントロピー関数を用いていたのに対して、現在のネットワークは活性化関数としてソフトマックス関数を、誤差関数として対数尤度関数を使っている点です。3章で説明したように、これは大きな差異ではありません。特に深い理由なくこのようにしています。実際の理由は、ソフトマックス関数と対数尤度誤差を同時に使うのが、画像分類のネットワークでは常套手段だからです。

もっと深いネットワークを使えば、より良い結果を得られるでしょうか？

さぁ、ネットワークの最初の層へ、畳み込み層を挿入するところから始めましょう。5×5 の局所受容野、1 のストライド長さ、20 個の特徴マップを使います。さらにMaxプーリング層を挿入します。このプーリング層は 2×2 のプーリングウィンドウを用いて特徴を結合します。したがって、ネットワーク構造全体は、追加の全結合層以外は上のセクションで議論したものに近いです。



この構造の場合、畳込み層とプーリング層が入力の訓練画像内の局所空間構造を学び、後方の層で全結合層が、もう少し抽象的なレベルで画像全体の統合情報を学ぶとみなせます。これは畳み込みニューラルネットワークの典型的なパターンです。

そのようなネットワークを訓練してみて、どう振る舞うか見てみましょう*。

*10 のサイズのミニバッチをここでも使い続けています。実際は、以前議論したように、ミニバッチのサイズを大きくすれば訓練は高速化できるでしょう。同じミニバッチサイズを使っているのは、以前の章で行った実験と一貫性を保つためです。

```
>>> net = Network([
        ConvPoolLayer(image_shape=(mini_batch_size, 1, 28, 28),
                      filter_shape=(20, 1, 5, 5),
                      poolsize=(2, 2)),
        FullyConnectedLayer(n_in=20*12*12, n_out=100),
        SoftmaxLayer(n_in=100, n_out=10)], mini_batch_size)
>>> net.SGD(training_data, 60, mini_batch_size, 0.1,
            validation_data, test_data)
```

今回の分類精度は 98.78 となり、これまでで最高の結果でした。実際、誤差率を3分の1以上減らしました。これは素晴らしい改良です。

ネットワーク構造を確認すると、畳み込み層とプーリング層をまとめて1つの層と扱っています。別々の層としてみなすか、1つの層とみなすかは好みの問題です。network3.pyではまとめて1つの層とみなしています。なぜかというと、network3.pyのコードを少しコンパクトにできるからです。しかし、お望みであれば、各層を別々に扱うようnetwork3.pyを修正することも簡単にできます。

練習問題

- 全結合層を除外して、畳み込み-プーリング層とソフトマックス層のみ使うと、分類の精度はどうなるでしょうか？ 全結合層の存在が寄与しているのでしょうか？

98.78 % の分類精度をさらに向上できるでしょうか？

2つ目の畳み込み-プーリング層を挿入してみましょう。既存の畳み込み-プーリング層と全結合層の間に入れます。今回の畳み込み-プーリング層にも、5 × 5 の局所受容野と 2 × 2 のプーリング領域という設定を適用

します。訓練時のハイパーパラメータは以前と同じ条件として、何が起きるか見てみましょう。

```
>>> net = Network([
        ConvPoolLayer(image_shape=(mini_batch_size, 1, 28, 28),
                      filter_shape=(20, 1, 5, 5),
                      poolsize=(2, 2)),
        ConvPoolLayer(image_shape=(mini_batch_size, 20, 12, 12),
                      filter_shape=(40, 20, 5, 5),
                      poolsize=(2, 2)),
        FullyConnectedLayer(n_in=40*4*4, n_out=100),
        SoftmaxLayer(n_in=100, n_out=10)], mini_batch_size)
>>> net.SGD(training_data, 60, mini_batch_size, 0.1,
            validation_data, test_data)
```

前回よりも良い結果です！ 99.06 %の分類精度に到達しました！

ここで2つの疑問が湧きます。1つ目の疑問は、2層目の畳込み-プーリング層を適用した意味とは何なのか？というものです。2層目の畳み込み-プーリング層の入力は、12×12 の入力"画像"とみなせます。その各"ピクセル"は、もとの入力画像がある局所的な特徴を保持するか否かを示します。したがって、この層はもとの入力画像の別バージョンの"画像"を入力として持つと考えられます。そのバージョンの画像は抽象化されており情報が縮約されていますが、空間的構造は保持しています。したがって、2層目の畳み込み-プーリング層には存在意義があると言えるのです。

これは魅力的な見方です。でも、そうなると2つ目の疑問が湧きます。1層目の出力は 20 の個別の特徴マップです。したがって、2層目の畳み込み-プーリング層への入力は $20 \times 12 \times 12$ となります。これは、1層目の畳み込み-プーリング層の場合のように1つの画像が入力されるというよりも、まるで 20 の異なる入力画像が畳み込み-プーリング層に入力されるかのようです。2層目の畳み込み-プーリング層のニューロンは、これらの多数の入力画像にどのような反応を見せるのでしょうか？ 実際、2層目の自身の局所受容野中の入力ニューロン $20 \times 5 \times 5$ の**全て**から、2層目の各ニューロンは学習します。ざっくり言い換えると、2層目の畳込み-プーリング層の特徴検出器は前層の特徴**全て**にアクセスします。ただし、特定の局所受容野*の範囲においてのみですが。

入力がカラー画像の場合、1層目のこの問題は発生するでしょう。その場合、入力画像の赤・緑・青のチャネルに対応する3つの特徴を各ピクセルが保持します。したがって特徴検出器は、局所受容野の範囲内においては全ての色情報にアクセスできるのです。

問題

- **活性化関数として tanh の使用** 本書の前半の章でtanh関数はシグモイド関数よりも良い活性化関数であると述べました。これまでは主にシグモイド関数を使って議論を進めてきましたが、tanhを活性化関数として用いて少し実験をしてみましょう。畳み込み層と全結合

*activation_fn=tanhをパラメータとして

層でtanhを試しに使い、ネットワークを訓練してみてください*。シグモイドのネットワークの時と同じパラメータで始めてみましょう。ただし、エポック数は 60 ではなく 20 にします。ネットワークはどのように振る舞うでしょうか？ 60 エポックまで続けたらどうなるでしょう？ tanhの場合とシグモイドの場合で、検証データに対する精度を 60 エポックまでプロットしてみてください。あなたの結果が私の結果と近ければ、tanhのネットワークの方が少し高速ですが、最終的な精度はほぼ同じになったはずです。なぜtanhのネットワークの方が高速なのか説明できますか？ 学習率やスケール*などを変更することで、シグモイドと同じ速度で学習させることができるでしょうか？ tanhがシグモイドよりも優れている点を探すために、ハイパーパラメータやネットワーク構造を変更するなど、試行錯誤してください。**これは自由回答の問題であることに注意してください。 すべての設定を網羅して試せていませんが、個人的には、tanhへ活性化関数を変更する利点はないように思います。 もしかしたら利点を見つけられるかもしれません。 どちらにせよ、活性化関数をReLUへ変更する利点がこの後、見つかってしまいます。ですので、これ以上tanhを使用することは考えません。**

ConvPoolLayerとFullyConnectedLayerのクラスへ渡せます。

*$\sigma(z) = (1+\tanh(z/2))/2$ の式を思い出すことで、何かを思いつくでしょうか？

ReLUの使用: これまで開発してきたネットワークは、1998年の先駆的な論文*で使われたネットワークの亜種です。この論文では、MNISTの問題とLeNet-5というネットワークが紹介されています。私たちの理解と直感を促進する上で、この論文は有用です。特に、結果を良くするために何をすればよいかを示す、ネットワークの改善指針がたくさん載っています。

*その論文とは1998年のYann LeCun, Léon Bottou, Yoshua Bengio, Patrick Haffnerによる"Gradient-based learning applied to document recognition"です。細かいところに違いはたくさんあるのですが、ネットワーク全体で見ると私たちのネットワークに非常に似ています。

まず始めに、活性化関数をシグモイドではなく、ReLUに変更しましょう。すなわち $f(z) \equiv \max(0, z)$ という活性化関数を使うようにします。訓練のエポック数 60 、学習率 $\eta = 0.03$ で訓練します。パラメータ $\lambda = 0.1$ としてL2 正規化を使うと少し良くなるということも知っています。

```
>>> from network3 import ReLU
>>> net = Network([
        ConvPoolLayer(image_shape=(mini_batch_size, 1, 28, 28),
                      filter_shape=(20, 1, 5, 5),
                      poolsize=(2, 2),
                      activation_fn=ReLU),
        ConvPoolLayer(image_shape=(mini_batch_size, 20, 12, 12),
                      filter_shape=(40, 20, 5, 5),
                      poolsize=(2, 2),
                      activation_fn=ReLU),
        FullyConnectedLayer(n_in=40*4*4, n_out=100, activation_fn=ReLU),
        SoftmaxLayer(n_in=100, n_out=10)], mini_batch_size)
>>> net.SGD(training_data, 60, mini_batch_size, 0.03,
            validation_data, test_data, lmbda=0.1)
```

今回、99.23 %の分類精度を得ました。シグモイドの場合（99.06 %）と比べてほんの少し改善されました。しかし、同一の問題に対し様々な条件で試した結果、私が発見したのは、ReLUを使うとシグモイドを使ったネットワークよりも、一貫して高精度となることです。この問題に関しては、ReLUへ移行するのが得策のようです。

ReLUの活性化関数はシグモイドやtanh関数より何が優れているのでしょう？ この問いに対する答えを今はまだ持ち合わせていません。実際のところ、ReLUはここ数年で使われ始めたのです。その採用の理由は、試しに使ってみた実験の結果が良かったことにあります。最初の何人かが直感やヒューリスティックな議論*に従い、ReLUを試してみたのです。ReLUはベンチマークのデータセットを上手く分類しました。その実例は、さらに広がりつつあります。理想としては、応用方法に応じて使用する活性化関数を選ぶための理論が欲しいと思います。でも現実にそんな理論はありません。もし仮に活性化関数をさらに上手く選ぶことができれば、結果が良くなるでしょう。なので数十年後には、活性化関数に関する強力な理論が生まれていることを期待しています。でも今は、大ざっぱな理論にすがり、地道な実験を繰り返して、活性化関数を選ぶしかないのです。

*理由付けとしてよくあるのが、$\max(0, z)$ は大きな z に対しても飽和しないので良い、というものです。シグモイドは飽和するのに対して、このReLUは飽和しないために学習し続けられるというのです。この主張は直感的には合っている気がします。しかし詳細な証明ではありません。この飽和の問題に関する議論は2章で既に触れました。

訓練データの拡張： 結果を改良する別の手法として、訓練データをアルゴリズムを使って拡張する手法があります。シンプルなやり方は、訓練データの各画像を1ピクセルずつ上下左右のいずれかの方向にずらすことです。expand_mnist.pyのプログラムをシェルプロンプトから動かして試せます*。

*expand_mnist.pyのコードはここで取得できます。

```
$ python expand_mnist.py
```

このプログラムを実行すると、入力の 50,000 のMNISTの訓練画像を、250,000 と増やします。新たに生成した画像もネットワークを訓練するのに使えます。今回も上のネットワークと同じようにReLUを使います。最初の実験では、訓練のエポック数を減らしました。しかし、訓練データを増やすことで、過適合を防ぐ効果が期待できます。そのため幾つかの実験を経た後で、私はエポック数を 60 へ戻しました。さあ、訓練してみましょう。

```
>>> expanded_training_data, _, _ = network3.load_data_shared(
        "../data/mnist_expanded.pkl.gz")
>>> net = Network([
        ConvPoolLayer(image_shape=(mini_batch_size, 1, 28, 28),
                      filter_shape=(20, 1, 5, 5),
                      poolsize=(2, 2),
                      activation_fn=ReLU),
        ConvPoolLayer(image_shape=(mini_batch_size, 20, 12, 12),
```

```
                        filter_shape=(40, 20, 5, 5),
                        poolsize=(2, 2),
                        activation_fn=ReLU),
        FullyConnectedLayer(n_in=40*4*4, n_out=100, activation_fn=ReLU),
        SoftmaxLayer(n_in=100, n_out=10)], mini_batch_size)
>>> net.SGD(expanded_training_data, 60, mini_batch_size, 0.03,
            validation_data, test_data, lmbda=0.1)
```

拡張した訓練データを使った結果、99.37 %の分類精度を得ました。データに僅かな変化を施したことで、精度が向上したのです。データ拡張により結果が良くなることは以前にも議論しました。この時の議論の雰囲気を思い出してもらいましょう。2003年にSimard, Steinkraus, Platt*はMNISTに対する分類精度を 99.6 まで向上しました。その時には、2つの畳み込み-プーリング層とそれに続く 100 のニューロンを持つ全結合層からなるネットワークを使っていました。これは私たちのネットワークに非常に近いです。しかし、私たちのネットワークと彼らのネットワークの構造には細かい違いが幾つかあります。例えば、ReLUを使っていなかったことなどです。しかし、パフォーマンスが非常に良かった大きな理由は訓練データの拡張にあります。彼らはMNISTの訓練画像に対して、回転・並進移動・せん断の操作を行いました。さらに"elastic distortion"の操作も加えました。これは、人が数字を書く時に手の筋肉がランダムに振動する様子を模擬する方法です。これらの操作を組み合わせて、訓練データを実質的に増やし、99.6 %の分類精度を達成しました。

*2003年のPatrice Simard, Dave Steinkraus, John Plattによる Best Practices for Convolutional Neural Networks Applied to Visual Document Analysis。

問題

- 畳み込み層のアイデアとは、画像に対して並進方向の不変性を持つことです。となると、入力画像に対して並進移動の操作を加えて訓練データを増やしたときに、ネットワークの精度が向上するのは一見不思議です。なぜこれがとても合理的なのか説明できますか？

全結合層の追加: さらに改良できるでしょうか？ 1つの可能性としては、上記と同じ手続きを行いながらも、サイズの大きな全結合層を追加してみることです。300 と 1,000 のニューロンを持つ全結合層を追加して、それぞれ試したところ、それぞれ 99.46 %と 99.43 %の結果を得ました。これは興味深いですが、前回の結果(99.37 %)とあまり変わりませんでした。

さらに全結合層を追加してみたらどうでしょうか？ さあ、全結合層を1層追加して、100 ニューロンからなる全結合層を2つ持つネットワークとしましょう。

```
>>> net = Network([
        ConvPoolLayer(image_shape=(mini_batch_size, 1, 28, 28),
                        filter_shape=(20, 1, 5, 5),
                        poolsize=(2, 2),
                        activation_fn=ReLU),
```

```
        ConvPoolLayer(image_shape=(mini_batch_size, 20, 12, 12),
                      filter_shape=(40, 20, 5, 5),
                      poolsize=(2, 2),
                      activation_fn=ReLU),
        FullyConnectedLayer(n_in=40*4*4, n_out=100, activation_fn=ReLU),
        FullyConnectedLayer(n_in=100, n_out=100, activation_fn=ReLU),
        SoftmaxLayer(n_in=100, n_out=10)], mini_batch_size)
>>> net.SGD(expanded_training_data, 60, mini_batch_size, 0.03,
            validation_data, test_data, lmbda=0.1)
```

これにより、99.43 の分類精度を得ました。全結合層を追加しただけでは、またしても、精度が向上しませんでした。ニューロン数を 300 と 1,000 とした全結合層の場合でも同じ実験をしてみましたが、それぞれ 99.48 % と 99.47 %という結果でした。悪くないですが、格段には向上していません。

何が起きているのでしょうか？ 全結合層を増やすのは、MNIST問題には有効でないのでしょうか？ もしくはネットワークは改良されているのに、私たちが間違ったやり方で訓練しているのでしょうか？ 例えば、過適合を回避するために、より強力な正規化のテクニックを使うのもよいでしょう。別の例としては、3章で紹介したドロップアウトのテクニックがあります。ドロップアウトの基礎的なアイデアを思い出してください。訓練時に各活性化出力をランダムに0とするのです。これによりモデルが、各入力の有無の違いにロバストになるため、各訓練データの特異性に依存しなくなります。このドロップアウトを最後の全結合層に適用してみましょう。

```
>>> net = Network([
        ConvPoolLayer(image_shape=(mini_batch_size, 1, 28, 28),
                      filter_shape=(20, 1, 5, 5),
                      poolsize=(2, 2),
                      activation_fn=ReLU),
        ConvPoolLayer(image_shape=(mini_batch_size, 20, 12, 12),
                      filter_shape=(40, 20, 5, 5),
                      poolsize=(2, 2),
                      activation_fn=ReLU),
        FullyConnectedLayer(
            n_in=40*4*4, n_out=1000, activation_fn=ReLU, p_dropout=0.5),
        FullyConnectedLayer(
            n_in=1000, n_out=1000, activation_fn=ReLU, p_dropout=0.5),
        SoftmaxLayer(n_in=1000, n_out=10, p_dropout=0.5)],
        mini_batch_size)
>>> net.SGD(expanded_training_data, 40, mini_batch_size, 0.03,
            validation_data, test_data)
```

これにより 99.60 %の分類精度を得ました。やっと 100 のニューロンからなるネットワークのベンチマーク結果である 99.37 を大きく更新しました。

2つの注目すべき変化があります。

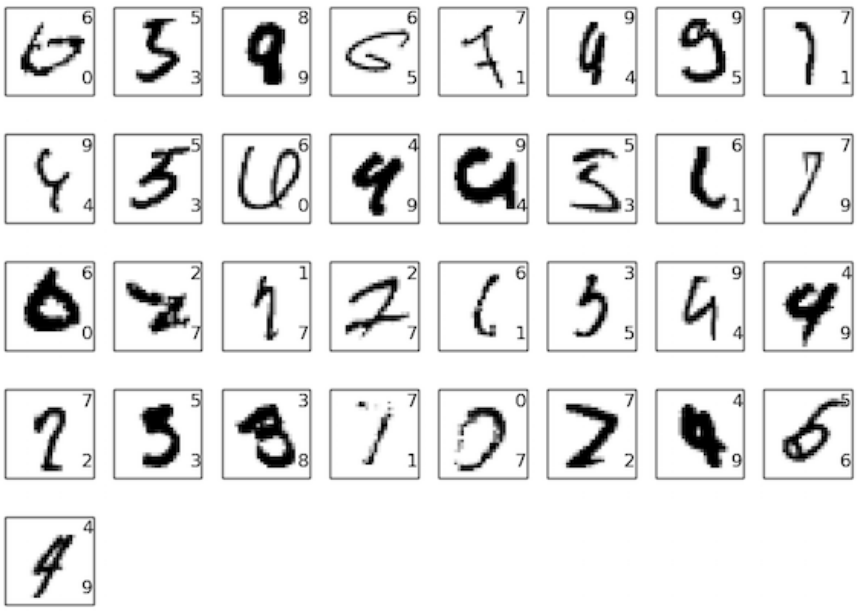
1つ目は、訓練するエポック数を 40 に減らしたことです。ドロップアウトが過適合を抑制するため、高速に学習できるのです。

2つ目は、全結合層内のニューロン数が以前と同じ 100 個ではなく、1,000 個であることです。ドロップアウトはニューロンの多くを効率的に省いているので、ニューロン数の増量が必要なのです。実際、300 と 1,000 の隠れニューロンを使って実験もしてみましたが、1,000 ニューロンの場合の方が（僅かですが）結果が良かったです。

ネットワークのアンサンブルの活用：パフォーマンスをさらに向上させる簡単な方法は、複数のニューラルネットワークを作成し、それらに多数決で分類を決めさせることです。例えば、上述の条件を満たす 5 つの異なるニューラルネットワークを訓練して、それぞれ 99.6 %の精度を得たと想定します。各ネットワークは精度は同等だったとしても、ランダムな初期化がされているため誤差の出し方が異なります。つまり、これら 5 つのニューラルネットワーク間で多数決を取れば、個々のニューラルネットワークだけの場合よりも良い分類が可能となるはずです。

あまりに上手く行きそうなので、胡散臭いですね。でも、この種のアンサンブル手法はニューラルネットワークや他の機械学習での常套手段です。そして実際、精度は向上し、99.67 %となりました。この結果を言い換えると、ネットワークのアンサンブルにより 10,000 のテスト画像のうち、33 以外の全てを正しく分類できたのです。

テストセットで間違えたものを下に示します。右上のラベルはMNISTによる正しい分類を示し、右下はネットワークのアンサンブルによる出力を示します。



この結果は細かく確認する価値があります。最初の2つの数字、"6"と"5"はアンサンブルが判断した間違いです。しかし、それらはいずれも理解できる間違いです。人間でも間違えそうです。その"6"は"0"のように見えますし、"5"は"3"のように見えます。3つめの画像は"8"のはずですが、実際には"9"に近く見えます。なので、私はネットワークのアンサンブルの判断の方が正しいように思えます。つまり、その数字を書いた人よりもネットワークのほうが良い仕事をしていると思います。一方、4つ目の画像の"6"はネットワークによる分類は不思議に感じます。

大抵のケースでは私たちのネットワークの選択は、少なくともっともらしいように見えます。幾つかのケースでは、数字を書いた人よりも良いはたらきをしています。上記に示さなかった9,967の画像を考慮すると、全体として私たちのネットワークは素晴らしいパフォーマンスを発揮していると言えます。そのような文脈を考えると、僅かにある分類の明らかな誤りも許容できます。現実にはどんなに注意深い人間でさえも、時には間違いを犯します。したがって、とんでもなく注意深くて理論的な人間だけがこのネットワークよりもよい精度を出せると思います。私たちのネットワークは人間の最高精度に到達しつつあるのです。

なぜ全結合層だけにドロップアウトを適用したのか： 上記のコードを注意深く見ると、ドロップアウトを全結合層にのみ適用しており、畳み込み層へは適用していないことに気づくでしょう。原理的には、畳み込み層に対しても同じ手続きを適用できます。しかし、実際その必要はありません。畳み込み層は過適合に対するもともとの抵抗力が強いのです。その理由は、重みを共有することにより、畳込みフィルタが画像全体から学習することなるからです。これにより訓練データの局所的な特異性による影響を受けにくくなっています。したがって、ドロップアウトなどの他の正規化手法を適用する必要性が薄いのです。

さらなる性能向上をめざして： MNIST問題に対する性能をもっと上げることはできます。Rodrigo Benensonが 有益なまとめページを作っています。このページでは年々の進化を、論文にリンクした形で確認できます。これらの論文では、私たちが使ってきたものと、近しい深層畳み込みネットワークを使用しています。論文を漁れば、面白いテクニックがたくさん見つかるでしょう。それを実装するのは楽しいはずです。もし実装する場合には、高速に訓練ができる単純なネットワークを使うのが賢いと思います。そういったネットワークでは、何が起きているのかをすぐに理解しやすいからです。

私は最近の大抵の論文を調査しようとしません。しかし、1つだけ例外があります。2010年のCireşan, Meier, Gambardella, Schmidhuberによる論文です*。この論文はシンプルさが好きです。ネットワークは多層で、全結合層のみを使っています（畳み込み層はなし）。最も成功したネットワークは 2500, 2000, 1500, 1000, 500 のニューロンをそれぞれ含む全結合層からなるネットワークです。訓練データを拡張するために Simard **et al**に似たアイデアを使っています。しかし、それ以外にも幾つかのトリックを用いています。その1つは畳み込み層を使わないことです。簡素で平凡ですが、1980年代（もしMNISTデータセットがあったとしたら）であっても計算機の能力さえあれば同じことが可能です。彼らは私たちと同等程度の 99.65 %の分類精度を達成しました。ポイントはとても深いネットワークを使い、GPUで高速に訓練したことです。エポック数を大きく取って訓練しています。訓練時間を長く取ることで、学習率を 10^{-3} から 10^{-6} へ徐々に減少させています。論文のネットワーク構造を使って、結果が合うかどうか試してみるのは楽しい演習になるでしょう。

*2010年のDan Claudiu Cireşan, Ueli Meier, Luca Maria Gambardella, Jürgen Schmidhuberによる Deep, Big, Simple Neural Nets Excel on Handwritten Digit Recognition。

なぜ訓練できるのでしょうか？ 前章で、深くて多層のニューラルネットワークの訓練における本質的な障壁について 学びました。特に、勾配がとても不安定になる問題を確認しました。出力層から前方の層に遡るにしたがって、勾配は消失（勾配消失問題）するか、爆発（勾配爆発問題）してしまう傾向があります。勾配は訓練のきっかけとなるものなので、勾配が不安定になると問題となります。

では、どうやってこの問題を避けているのでしょうか？

もちろん答えは、この問題を回避していない、というものです。代わりに、結果を良くするための操作を幾つか行っています。(1)畳み込み層を使うことで、層のパラメータ数が劇的に減っているため、学習に伴う問題が起きにくくなっています。(2)強力な正規化テクニック（特にドロップアウトと畳込み層）を使うことで、他の複雑なネットワークでは問題となる過適合を防いでいます。(3)シグモイドの代わりにReLUを使うことで、訓練を高速に行っています。実験では 3-5 倍速くなっています。(4) GPUを使って訓練時間を長くしています。特に、最後の実験では、もとのMNISTの訓練データの 5 倍のデータを用いて 40 エポック分訓練しています。本書の前半ではもとの訓練データを用いて、30 エポック分訓練していました。(3) と (4) の要素を組み合わせると以前の 30 倍長く時間がかかります。

あなたはきっと、「それだけ？深いネットワークの訓練に必要な工夫は本当にそれだけ？（これまでさんざん苦労してきたのに）一体どうなってるの？」と思うでしょう。

もちろん、他の工夫も加えています。過適合を防ぐために十分大きなデータセットを利用したり、学習が遅くなるのを防ぐために正しいコスト関数を使ったり、ニューロンの飽和による学習の遅延を防ぐために重みを上手く初期化したり、訓練データをアルゴリズムで拡張したりしました。上記の工夫についてはこれまでの章で解説してきたので、この章では説明が少なくても理解できるでしょう。

これらの工夫はシンプルなものです。シンプルですが、同時に使用すると強力です。ディープラーニングがとても容易になります！

ところで、これらのネットワークの深さはどのくらい？ 畳み込み-プーリング層を1つの層として数えると、私たちの最後のネットワークは隠れ層を 4 つ持つこととなります。このネットワークは本当に **深層** ネットワークと呼ばれるに値するのでしょうか？ もちろん、これまで学んできた他の浅いネットワークと比べると 4 つの隠れ層というのは深いです。これまでの浅いネットワークは 1 つ、もしくは 2 つだけ隠れ層を持っていました。一方、2015年の最新の深層ネットワークは 10 以上の隠れ層を持ちます。最近はとにかく層を深くする傾向があります。聞くところによると、隠れ層の数で周囲に遅れを取っているのでは、ディープラーニングと呼べないというのです。しかし、一時的な結果に依存する何かをディープラーニングと呼ぶことになってしまうため、私はこの態度には共感しません。ディープラーニングの本当のブレークスルーは、2000年代中盤まで支配的だった浅い 1 層や 2 層のネットワーク以外でも実用的な結果が得られることが判明したことだと思っています。それは、はるかに表現力豊かなモデルを探索する機会を得るという意味で真のブレークスルーでした。しかしそれを除いても、層の深さは本質的な議論ではありません。むしろ、他の目的に深いネットワークをいかに応用できるかの方が重要です。

注意点： このセクションでは、単一の隠れ層を持つ浅いネットワークから多層の畳み込みネットワークへ順調に移行しました。畳み込みネットワークは簡単そうです！ ネットワークに変更を加えることで、すぐに結果が良くなりました。しかしあなたが実験を始めても、その直後は上手く行くとは限らないことを私は保証します。その理由は、実際には多くの実験を行っているにも関わらず、本書ではそれらを省略した無駄のない文脈で、畳み込みネットワークを紹介したからです。失敗に終わった実験を裏でたくさん行っています。無駄のない文脈で説明してきたので、あなたの基礎理解は深まったと思っています。しかし、誤解を与えた恐れもありますね。上手く行くネットワークに辿り着くには、イライラしながら試行錯誤をする必要があります。実際、多くの実験をこなさなければならないはずです。その時には、3章の ニューラルネットワークのハイパーパラメータ

をどう選ぶかの議論が作業効率化の助けになります。そして、この章の残りを読むことも重要です。

畳み込みネットワークのコード

さあ、私たちのコードnetwork3.pyを見てみましょう。3章で開発したnetwork2.pyに構造的に似ています。ただし、Theanoを導入したため、詳細は異なっています。まずFullyConnectedLayerのクラスから確認を始めましょう。これは本書で既に扱ってきた層に似ています。コードはこのようになっています*。

```
class FullyConnectedLayer(object):
    def __init__(self, n_in, n_out, activation_fn=sigmoid, p_dropout=0.0):
        self.n_in = n_in
        self.n_out = n_out
        self.activation_fn = activation_fn
        self.p_dropout = p_dropout
        # Initialize weights and biases
        self.w = theano.shared(
            np.asarray(
                np.random.normal(
                    loc=0.0, scale=np.sqrt(1.0/n_out), size=(n_in, n_out)),
                dtype=theano.config.floatX),
            name='w', borrow=True)
        self.b = theano.shared(
            np.asarray(np.random.normal(loc=0.0, scale=1.0, size=(n_out,)),
                       dtype=theano.config.floatX),
            name='b', borrow=True)
        self.params = [self.w, self.b]

    def set_inpt(self, inpt, inpt_dropout, mini_batch_size):
        self.inpt = inpt.reshape((mini_batch_size, self.n_in))
        self.output = self.activation_fn(
            (1-self.p_dropout)*T.dot(self.inpt, self.w) + self.b)
        self.y_out = T.argmax(self.output, axis=1)
        self.inpt_dropout = dropout_layer(
            inpt_dropout.reshape((mini_batch_size, self.n_in)), self.p_dropout)
        self.output_dropout = self.activation_fn(
            T.dot(self.inpt_dropout, self.w) + self.b)

    def accuracy(self, y):
        "Return the accuracy for the mini-batch."
        return T.mean(T.eq(y, self.y_out))
```

self.wを初期化する行で、scale=np.sqrt(1.0/n_out)としているのに気づいた人もいるでしょう。3章の議論ではscale=np.sqrt(1.0/n_in)を使うように推していたのに、何故この方法を取っているのかと不思議に感じる方もいると思います。実を言うと、これは単純に私のミスです。本当は、この章のコードを修正しなくてはいけないのですが、現在私は他のプロジェクトにかかりきりになっているので、しばらくはそのままにしておきます。

__init__の大部分は自明ですが、少し解説しておくとコードの意味が明快になるでしょう。通常通り、適切な標準偏差を設定して重みとバイアスをランダムにばらつかせて初期化しています。この操作を行っている行は少し馴染みが薄いかもしれません。でも、複雑そうに見える処理は実は、重みとバイアスをTheanoが共有変数と呼ぶ変数へ渡しているだけです。この操作は、GPU実行可能であれば、この変数がGPU上で処理されることを保証するというものです。詳細にはこれ以上踏み込みません。興味

があれば、Theanoのドキュメントを参照してください。この重みとバイアスの初期化は、以前議論したようにシグモイドの活性化関数を考慮して行われていることにも注意してください。理想的には、重みとバイアスの初期化をtanhやReLU向けに、少し異なる方法で行うのがよいでしょう。これは後々議論します。__init__関数はself.params = [self.w, self.b]を行い終了します。この処理は、層に関連する学習可能なパラメータをまとめる手軽な方法です。後々、Network.SGD関数がparamsの属性を使うときに、Networkのインスタンスが学習するパラメータを明らかにしているのです。

set_inpt関数は層への入力を設定し、対応する出力を計算するために使われます。inputではなくinptという名前を使っているのは、Pythonにinputというビルトイン関数があるためです。ビルトイン関数と混在すると、予測不能な振る舞いが起きる恐れがあったので避けました。さて、入力を2つの異なる方法で設定していることに注意してください。それぞれself.inptとself.inpt_dropoutです。訓練時にはドロップアウトを使いたいと思うかもしれないので、こうしました。その時には、self.p_dropoutのニューロンの一部を取り除く必要があります。それが、関数dropout_layerの最後から2行目のset_inpt関数が行っていることです。したがって、self.inpt_dropoutとself.output_dropoutは訓練時に使用されます。一方、self.inptとself.outputは、例えば、検証データとテストデータの精度を評価する場合など、どのような場合にも使われます。

ConvPoolLayerとSoftmaxLayerクラスの定義はFullyConnectedLayerの定義と似ています。本当にそっくりなので、ここではコードを引用しません。興味があれば、このセクションの後にあるnetwork3.pyの全コードを参照してください。

しかし、いくつかの細かい違いに着目するのは悪くありません。ConvPoolLayerとSoftmaxLayerは、層の種類に応じて適切な出力の活性化を行っています。幸運なことに、Theanoの提供するビルトイン演算操作を使えば、畳み込みやMaxプーリング、ソフトマックス関数の計算を簡単に行なえます。

さらに、些細な事ですが、ソフトマックス層を導入した際、重みとバイアスの初期化の仕方を議論しませんでした。一方、シグモイドの層では、重みを適切なランダム値に初期化するべきであることは既に述べました。しかし、そのヒューリスティックな議論はシグモイドニューロン(と少し修正を加えればtanh)に特有なものです。同じ議論をソフトマックス層に適用すべき理由は特にありません。したがって、その初期化法を適用するア・プリオリな理由はないのです。むしろ、0 で全ての重みとバイアスを初

期化した方がよいと思います。これはとても**場当たり的な**やり方に見えますが、実践では十分上手くいきます。

よし、これで層の全種類のクラス定義を確認したことになります。Networkクラスはどうでしょう？ __init__関数を見るところから始めましょう。

```
class Network(object):

    def __init__(self, layers, mini_batch_size):
        """Takes a list of `layers`, describing the network architecture, and
        a value for the `mini_batch_size` to be used during training
        by stochastic gradient descent.

        """
        self.layers = layers
        self.mini_batch_size = mini_batch_size
        self.params = [param for layer in self.layers for param in layer.params]
        self.x = T.matrix("x")
        self.y = T.ivector("y")
        init_layer = self.layers[0]
        init_layer.set_inpt(self.x, self.x, self.mini_batch_size)
        for j in xrange(1, len(self.layers)):
            prev_layer, layer  = self.layers[j-1], self.layers[j]
            layer.set_inpt(
                prev_layer.output, prev_layer.output_dropout, self.mini_batch_size)
        self.output = self.layers[-1].output
        self.output_dropout = self.layers[-1].output_dropout
```

コードの大部分は見ればわかると思います。self.params = [param for layer in ...]の行では、各層のパラメータを1つのリストにまとめています。上で触れたように、Network.SGD関数がself.paramsを観て、Networkの中のどの変数が学習するのかを把握します。self.x = T.matrix("x")とself.y = T.ivector("y")の行は、xとyと名付けたTheanoのシンボリック変数を定義する部分に当たります。これらの変数は、入力とネットワークの望みの出力を表現するのに使われます。

さて本書はTheanoのチュートリアルではないので、シンボリック変数*の意味については深く踏み込みません。しかし簡単に説明しておくと、シンボリック変数とは数学的な変数であり、値を表すものでは**ありません**。加算、減算、乗算や関数の適用などの操作をシンボリック変数へ施すことができます。他にも、畳み込みやMaxプーリングなどの、シンボリック変数を操作する方法をTheanoは多数提供しています。シンボリック変数を使う大きな利点は、逆伝播のアルゴリズムで必要な微分を高速なシンボリック微分として行える点です。確率的勾配降下法を様々なネットワーク構造に対して実行するにあたって、これはとても強力に感じます。次の数行では、ネットワークの出力を定義しています。その際、最初の層へ入力を設定するところから始まっています。

*Theanoの導入にはTheanoのドキュメントを読むべきです。もし詰まったら、オンラインにある他のチュートリアルを参照すると良いでしょう。例えば、このチュートリアルは基礎を広く押さえています。

```
        init_layer.set_inpt(self.x, self.x, self.mini_batch_size)
```

入力が1つのミニバッチに一度に設定されることに注意してください。入力self.xを2つの引数として渡していることにも注意してください。これは、ネットワークを(ドロップアウトの有無で)2つの異なる方法で使うためです。forループでは、Networkの層間をシンボリック変数self.xが順伝播していきます。これにより、最後のoutputとoutput_dropoutの中身を定義することができます。これらはNetworkの出力を表現します。

これで、Networkがどのように初期化されるかがわかりました。さあSGD関数による訓練方法を見ていきましょう。コードは長いですが、構造は実にシンプルです。コードの後に説明のコメントを記します。

```
    def SGD(self, training_data, epochs, mini_batch_size, eta,
            validation_data, test_data, lmbda=0.0):
        """Train the network using mini-batch stochastic gradient descent."""
        training_x, training_y = training_data
        validation_x, validation_y = validation_data
        test_x, test_y = test_data

        # compute number of minibatches for training, validation and testing
        num_training_batches = size(training_data)/mini_batch_size
        num_validation_batches = size(validation_data)/mini_batch_size
        num_test_batches = size(test_data)/mini_batch_size

        # define the (regularized) cost function, symbolic gradients, and updates
        l2_norm_squared = sum([(layer.w**2).sum() for layer in self.layers])
        cost = self.layers[-1].cost(self)+¥
               0.5*lmbda*l2_norm_squared/num_training_batches
        grads = T.grad(cost, self.params)
        updates = [(param, param-eta*grad)
                   for param, grad in zip(self.params, grads)]

        # define functions to train a mini-batch, and to compute the
        # accuracy in validation and test mini-batches.
        i = T.lscalar() # mini-batch index
        train_mb = theano.function(
            [i], cost, updates=updates,
            givens={
                self.x:
                training_x[i*self.mini_batch_size: (i+1)*self.mini_batch_size],
                self.y:
                training_y[i*self.mini_batch_size: (i+1)*self.mini_batch_size]
            })
        validate_mb_accuracy = theano.function(
            [i], self.layers[-1].accuracy(self.y),
            givens={
                self.x:
                validation_x[i*self.mini_batch_size: (i+1)*self.mini_batch_size],
                self.y:
                validation_y[i*self.mini_batch_size: (i+1)*self.mini_batch_size]
            })
        test_mb_accuracy = theano.function(
            [i], self.layers[-1].accuracy(self.y),
            givens={
                self.x:
```

```
                test_x[i*self.mini_batch_size: (i+1)*self.mini_batch_size],
                self.y:
                test_y[i*self.mini_batch_size: (i+1)*self.mini_batch_size]
            })
        self.test_mb_predictions = theano.function(
            [i], self.layers[-1].y_out,
            givens={
                self.x:
                test_x[i*self.mini_batch_size: (i+1)*self.mini_batch_size]
            })
        # Do the actual training
        best_validation_accuracy = 0.0
        for epoch in xrange(epochs):
            for minibatch_index in xrange(num_training_batches):
                iteration = num_training_batches*epoch+minibatch_index
                if iteration
                    print("Training mini-batch number {0}".format(iteration))
                cost_ij = train_mb(minibatch_index)
                if (iteration+1)
                    validation_accuracy = np.mean(
                        [validate_mb_accuracy(j) for j in xrange(num_validation_batches)])
                    print("Epoch {0}: validation accuracy {1:.2
                        epoch, validation_accuracy))
                    if validation_accuracy >= best_validation_accuracy:
                        print("This is the best validation accuracy to date.")
                        best_validation_accuracy = validation_accuracy
                        best_iteration = iteration
                        if test_data:
                            test_accuracy = np.mean(
                                [test_mb_accuracy(j) for j in xrange(num_test_batches)])
                            print('The corresponding test accuracy is {0:.2
                                test_accuracy))
        print("Finished training network.")
        print("Best validation accuracy of {0:.2
            best_validation_accuracy, best_iteration))
        print("Corresponding test accuracy of {0:.2
```

最初の数行は単純です。データセットを x と y の要素に分けて、各データセットで使われるミニバッチの数を計算しています。次の数行は興味深く、Theanoの醍醐味となる部分です。その行をここに引用してみましょう。

```
    # define the (regularized) cost function, symbolic gradients, and updates
      l2_norm_squared = sum([(layer.w**2).sum() for layer in self.layers])
      cost = self.layers[-1].cost(self)+¥
             0.5*lmbda*l2_norm_squared/num_training_batches
      grads = T.grad(cost, self.params)
      updates = [(param, param-eta*grad)
                 for param, grad in zip(self.params, grads)]
```

これらの行では、正規化した対数尤度の誤差関数を用意しています。また、パラメータの更新に応じて、勾配関数の中の対応する微分を計算します。Theanoを使うと数行でこれら全てを実装できます。隠されていて唯一分かりにくいのは、costの計算の時に出力層のcost関数を呼ぶことです。このcost関数はnetwork3.py内の別の箇所にあります。しかし、そのコ

ードは短くシンプルです。さて、これら全ての設定が終わると、train_mb関数を定義する段階へ移ります。このTheanoのシンボリック関数はupdatesを用い、ミニバッチのインデックスをもとにNetworkのパラメータを更新します。同様に、validate_mb_accuracyとtest_mb_accuracyは、検証データやテストデータのミニバッチに基づいて、Networkの精度を計算します。これらの関数の結果の平均をとって、検証データやテストデータの全体精度を計算できるのです。

SGD関数の残りの部分は自明だと思います。単純にエポック数分だけ反復し、訓練データのミニバッチに基づいてネットワークを訓練し、検証データとテストデータの精度を計算します。

よし、これでnetwork3.pyの重要な部分は理解したことになります。プログラム全体を見てみましょう。コードを詳細に読み解く必要はありません。きっと、コードを眺めるだけで楽しいはずです。あなたが気になった箇所を深掘りしてみるのも良いと思います。もちろん、コードを深く理解するための一番良い方法は、コードに修正を加えたり、何か特徴を追加したり、もしくはエレガントになるようリファクタリングしてみることです。コードの後ろに、いくつか修正すべき項目を載せています*

*Theanoを使ってコードをGPU実行する方法は少しトリッキーです。特に、GPUからデータを取得するところは間違えやすく、間違えるとかなり低速になってしまいます。私はこれを避けようと試行錯誤してきました。このコードではTheanoの最適化設定を注意深く行っているため、かなり高速に動作するはずです。詳細はTheanoのドキュメントを見て確認してください。

```
"""network3.py
~~~~~~~~~~~~~~

A Theano-based program for training and running simple neural
networks.

Supports several layer types (fully connected, convolutional, max
pooling, softmax), and activation functions (sigmoid, tanh, and
rectified linear units, with more easily added).

When run on a CPU, this program is much faster than network.py and
network2.py.  However, unlike network.py and network2.py it can also
be run on a GPU, which makes it faster still.

Because the code is based on Theano, the code is different in many
ways from network.py and network2.py.  However, where possible I have
tried to maintain consistency with the earlier programs.  In
particular, the API is similar to network2.py.  Note that I have
focused on making the code simple, easily readable, and easily
modifiable.  It is not optimized, and omits many desirable features.

This program incorporates ideas from the Theano documentation on
convolutional neural nets (notably,
http://deeplearning.net/tutorial/lenet.html ), from Misha Denil's
implementation of dropout (https://github.com/mdenil/dropout ), and
from Chris Olah (http://colah.github.io ).

Written for Theano 0.6 and 0.7, needs some changes for more recent
versions of Theano.

"""

#### Libraries
```

```
# Standard library
import cPickle
import gzip

# Third-party libraries
import numpy as np
import theano
import theano.tensor as T
from theano.tensor.nnet import conv
from theano.tensor.nnet import softmax
from theano.tensor import shared_randomstreams
from theano.tensor.signal import downsample

# Activation functions for neurons
def linear(z): return z
def ReLU(z): return T.maximum(0.0, z)
from theano.tensor.nnet import sigmoid
from theano.tensor import tanh


#### Constants
GPU = True
if GPU:
    print "Trying to run under a GPU.  If this is not desired, then modify "+¥
        "network3.py¥nto set the GPU flag to False."
    try: theano.config.device = 'gpu'
    except: pass # it's already set
    theano.config.floatX = 'float32'
else:
    print "Running with a CPU.  If this is not desired, then the modify "+¥
        "network3.py to set¥nthe GPU flag to True."

#### Load the MNIST data
def load_data_shared(filename="../data/mnist.pkl.gz"):
    f = gzip.open(filename, 'rb')
    training_data, validation_data, test_data = cPickle.load(f)
    f.close()
    def shared(data):
        """Place the data into shared variables.  This allows Theano to copy
        the data to the GPU, if one is available.

        """
        shared_x = theano.shared(
            np.asarray(data[0], dtype=theano.config.floatX), borrow=True)
        shared_y = theano.shared(
            np.asarray(data[1], dtype=theano.config.floatX), borrow=True)
        return shared_x, T.cast(shared_y, "int32")
    return [shared(training_data), shared(validation_data), shared(test_data)]

#### Main class used to construct and train networks
class Network(object):

    def __init__(self, layers, mini_batch_size):
        """Takes a list of `layers`, describing the network architecture, and
        a value for the `mini_batch_size` to be used during training
        by stochastic gradient descent.

        """
        self.layers = layers
        self.mini_batch_size = mini_batch_size
        self.params = [param for layer in self.layers for param in layer.params]
        self.x = T.matrix("x")
```

```
        self.y = T.ivector("y")
        init_layer = self.layers[0]
        init_layer.set_inpt(self.x, self.x, self.mini_batch_size)
        for j in xrange(1, len(self.layers)):
            prev_layer, layer  = self.layers[j-1], self.layers[j]
            layer.set_inpt(
                prev_layer.output, prev_layer.output_dropout, self.mini_batch_size)
        self.output = self.layers[-1].output
        self.output_dropout = self.layers[-1].output_dropout

    def SGD(self, training_data, epochs, mini_batch_size, eta,
            validation_data, test_data, lmbda=0.0):
        """Train the network using mini-batch stochastic gradient descent."""
        training_x, training_y = training_data
        validation_x, validation_y = validation_data
        test_x, test_y = test_data

        # compute number of minibatches for training, validation and testing
        num_training_batches = size(training_data)/mini_batch_size
        num_validation_batches = size(validation_data)/mini_batch_size
        num_test_batches = size(test_data)/mini_batch_size

        # define the (regularized) cost function, symbolic gradients, and updates
        l2_norm_squared = sum([(layer.w**2).sum() for layer in self.layers])
        cost = self.layers[-1].cost(self)+¥
               0.5*lmbda*l2_norm_squared/num_training_batches
        grads = T.grad(cost, self.params)
        updates = [(param, param-eta*grad)
                   for param, grad in zip(self.params, grads)]

        # define functions to train a mini-batch, and to compute the
        # accuracy in validation and test mini-batches.
        i = T.lscalar() # mini-batch index
        train_mb = theano.function(
            [i], cost, updates=updates,
            givens={
                self.x:
                training_x[i*self.mini_batch_size: (i+1)*self.mini_batch_size],
                self.y:
                training_y[i*self.mini_batch_size: (i+1)*self.mini_batch_size]
            })
        validate_mb_accuracy = theano.function(
            [i], self.layers[-1].accuracy(self.y),
            givens={
                self.x:
                validation_x[i*self.mini_batch_size: (i+1)*self.mini_batch_size],
                self.y:
                validation_y[i*self.mini_batch_size: (i+1)*self.mini_batch_size]
            })
        test_mb_accuracy = theano.function(
            [i], self.layers[-1].accuracy(self.y),
            givens={
                self.x:
                test_x[i*self.mini_batch_size: (i+1)*self.mini_batch_size],
                self.y:
                test_y[i*self.mini_batch_size: (i+1)*self.mini_batch_size]
            })
        self.test_mb_predictions = theano.function(
            [i], self.layers[-1].y_out,
            givens={
                self.x:
                test_x[i*self.mini_batch_size: (i+1)*self.mini_batch_size]
```

```
                })
            # Do the actual training
            best_validation_accuracy = 0.0
            for epoch in xrange(epochs):
                for minibatch_index in xrange(num_training_batches):
                    iteration = num_training_batches*epoch+minibatch_index
                    if iteration % 1000 == 0:
                        print("Training mini-batch number {0}".format(iteration))
                    cost_ij = train_mb(minibatch_index)
                    if (iteration+1) % num_training_batches == 0:
                        validation_accuracy = np.mean(
                            [validate_mb_accuracy(j) for j in xrange(num_validation_batches)])
                        print("Epoch {0}: validation accuracy {1:.2%}".format(
                            epoch, validation_accuracy))
                        if validation_accuracy >= best_validation_accuracy:
                            print("This is the best validation accuracy to date.")
                            best_validation_accuracy = validation_accuracy
                            best_iteration = iteration
                            if test_data:
                                test_accuracy = np.mean(
                                    [test_mb_accuracy(j) for j in xrange(num_test_batches)])
                                print('The corresponding test accuracy is {0:.2%}'.format(
                                    test_accuracy))
            print("Finished training network.")
            print("Best validation accuracy of {0:.2%} obtained at iteration {1}".format(
                best_validation_accuracy, best_iteration))
            print("Corresponding test accuracy of {0:.2%}".format(test_accuracy))

    #### Define layer types

    class ConvPoolLayer(object):
        """Used to create a combination of a convolutional and a max-pooling
        layer.  A more sophisticated implementation would separate the
        two, but for our purposes we'll always use them together, and it
        simplifies the code, so it makes sense to combine them.

        """

        def __init__(self, filter_shape, image_shape, poolsize=(2, 2),
                     activation_fn=sigmoid):
            """`filter_shape` is a tuple of length 4, whose entries are the number
            of filters, the number of input feature maps, the filter height, and the
            filter width.

            `image_shape` is a tuple of length 4, whose entries are the
            mini-batch size, the number of input feature maps, the image
            height, and the image width.

            `poolsize` is a tuple of length 2, whose entries are the y and
            x pooling sizes.

            """
            self.filter_shape = filter_shape
            self.image_shape = image_shape
            self.poolsize = poolsize
            self.activation_fn=activation_fn
            # initialize weights and biases
            n_out = (filter_shape[0]*np.prod(filter_shape[2:])/np.prod(poolsize))
            self.w = theano.shared(
                np.asarray(
                    np.random.normal(loc=0, scale=np.sqrt(1.0/n_out), size=filter_shape),
                    dtype=theano.config.floatX),
```

```
                borrow=True)
            self.b = theano.shared(
                np.asarray(
                    np.random.normal(loc=0, scale=1.0, size=(filter_shape[0],)),
                    dtype=theano.config.floatX),
                borrow=True)
            self.params = [self.w, self.b]

        def set_inpt(self, inpt, inpt_dropout, mini_batch_size):
            self.inpt = inpt.reshape(self.image_shape)
            conv_out = conv.conv2d(
                input=self.inpt, filters=self.w, filter_shape=self.filter_shape,
                image_shape=self.image_shape)
            pooled_out = downsample.max_pool_2d(
                input=conv_out, ds=self.poolsize, ignore_border=True)
            self.output = self.activation_fn(
                pooled_out + self.b.dimshuffle('x', 0, 'x', 'x'))
            self.output_dropout = self.output # no dropout in the convolutional layers

    class FullyConnectedLayer(object):

        def __init__(self, n_in, n_out, activation_fn=sigmoid, p_dropout=0.0):
            self.n_in = n_in
            self.n_out = n_out
            self.activation_fn = activation_fn
            self.p_dropout = p_dropout
            # Initialize weights and biases
            self.w = theano.shared(
                np.asarray(
                    np.random.normal(
                        loc=0.0, scale=np.sqrt(1.0/n_out), size=(n_in, n_out)),
                    dtype=theano.config.floatX),
                name='w', borrow=True)
            self.b = theano.shared(
                np.asarray(np.random.normal(loc=0.0, scale=1.0, size=(n_out,)),
                           dtype=theano.config.floatX),
                name='b', borrow=True)
            self.params = [self.w, self.b]

        def set_inpt(self, inpt, inpt_dropout, mini_batch_size):
            self.inpt = inpt.reshape((mini_batch_size, self.n_in))
            self.output = self.activation_fn(
                (1-self.p_dropout)*T.dot(self.inpt, self.w) + self.b)
            self.y_out = T.argmax(self.output, axis=1)
            self.inpt_dropout = dropout_layer(
                inpt_dropout.reshape((mini_batch_size, self.n_in)), self.p_dropout)
            self.output_dropout = self.activation_fn(
                T.dot(self.inpt_dropout, self.w) + self.b)

        def accuracy(self, y):
            "Return the accuracy for the mini-batch."
            return T.mean(T.eq(y, self.y_out))

    class SoftmaxLayer(object):

        def __init__(self, n_in, n_out, p_dropout=0.0):
            self.n_in = n_in
            self.n_out = n_out
            self.p_dropout = p_dropout
            # Initialize weights and biases
            self.w = theano.shared(
                np.zeros((n_in, n_out), dtype=theano.config.floatX),
```

```
            name='w', borrow=True)
        self.b = theano.shared(
            np.zeros((n_out,), dtype=theano.config.floatX),
            name='b', borrow=True)
        self.params = [self.w, self.b]

    def set_inpt(self, inpt, inpt_dropout, mini_batch_size):
        self.inpt = inpt.reshape((mini_batch_size, self.n_in))
        self.output = softmax((1-self.p_dropout)*T.dot(self.inpt, self.w) + self.b)
        self.y_out = T.argmax(self.output, axis=1)
        self.inpt_dropout = dropout_layer(
            inpt_dropout.reshape((mini_batch_size, self.n_in)), self.p_dropout)
        self.output_dropout = softmax(T.dot(self.inpt_dropout, self.w) + self.b)

    def cost(self, net):
        "Return the log-likelihood cost."
        return -T.mean(T.log(self.output_dropout)[T.arange(net.y.shape[0]), net.y])

    def accuracy(self, y):
        "Return the accuracy for the mini-batch."
        return T.mean(T.eq(y, self.y_out))


#### Miscellanea
def size(data):
    "Return the size of the dataset `data`."
    return data[0].get_value(borrow=True).shape[0]

def dropout_layer(layer, p_dropout):
    srng = shared_randomstreams.RandomStreams(
        np.random.RandomState(0).randint(999999))
    mask = srng.binomial(n=1, p=1-p_dropout, size=layer.shape)
    return layer*T.cast(mask, theano.config.floatX)
```

問題

- 現在、SGDの関数では訓練のエポック数をユーザが手入力するようになっています。しかし以前議論したように、訓練のエポック数を自動的に決める方法として早期打ち切りが知られています。そこで、network3.pyを修正して、この早期打ち切りを実装してください。

- 任意のデータセットに対して、精度を出力する関数をNetworkに追加してください。

- SGD関数を修正して、学習率 η がエポック数の関数になるようにしてください。**ヒント: この問題にしばらく取り組んだら、このリンクの議論を見るとよいでしょう。**

- この章の前半で、訓練データに(小さい)回転やせん断、並進移動を加えることで、訓練データを拡張するテクニックを紹介しました。network3.pyを修正して、上記のテクニックを取り入れてください。**巨大なメモリを持っていない場合には、拡張データセット全体を**

生成するのは実用的ではありません。 そのときは別のアプローチを考えてください

- ネットワークを記録、再生する機能をnetwork3.pyに加えてください。

- 現在のコードの欠点は、診断ツールが少ないことです。ネットワークの過適合の度合いを簡単に把握できる診断ツールを考えて、実装してください。

- ReLUの初期化手続きは、シグモイド（とtanh）のニューロンの場合と同じ手続きを使っています。以前の初期化の議論はシグモイド関数に特有のものでした。（出力も含め）全体がReLUから構成されるネットワークを考えてみてください。ネットワーク中の全ての重みを定数 $c > 0$ でスケーリングすると、単に出力が c^{L-1} 倍されることを示してください。ただし、L は層の数とします。最終層がソフトマックス関数になると、これはどのように変化するでしょうか？ シグモイドの初期化方法をReLUに適用するのはどう思いますか？ もっと良い初期化方法を思いつきますか？ **注意：これは自由回答の問題です。答えは決まっていません。 しかし、この問題を考えてみることで、ReLUを含むネットワークに対する理解が深まるでしょう。**

- 勾配が不安定になる問題への分析 は、以前の章でシグモイドニューロンに対して実施しました。ReLUから構成されるネットワークになると、この分析はどう変化するでしょうか？ 勾配が不安定となる問題を回避するための、ネットワークを修正する良い方法を思いつきますか？ **注意： 「良い」方法を探すのは研究課題です。 目的を達成する修正は実際には簡単に思いつきます。 しかし、本当に「良い」テクニックかどうかは私は深く調べ切っていません。**

画像認識の近年の進展

1998年、MNISTが生まれた年には、当時の最新のワークステーションを使ったとしても、ネットワークの訓練に数週間かかっていました。その時の精度を、現在GPUを使うと一時間以内に達成してしまいます。したがってMNISTは現在の技術の前では、もはや問題として物足りなくなってしまったと言えます。現在では、より難しい画像認識問題を研究の課題とするようになりました。このセクションでは、近年のニューラルネットワークを使った画像認識の研究を概観します。

このセクションは本書の大部分と趣向が異なります。これまで本書では、長く通用するアイデアをテーマとして扱ってきました。例えば、逆伝播、

正規化、畳込みネットワークなどです。今から記述していくような、長期的には価値が不明な流行っている知見を避けようとしてきたのです。科学の世界では、流行りというのはすぐに移り変わり、影響力が薄いものです。このことから考えると、懐疑的な人は次のように述べるでしょう。「ええとつまり、近年の画像認識の成果は結局、流行りものですよね？ 2、3年後には、物事は移り変わっているはずです。したがって最新の結果というものは、最先端で競争する専門家のような限られた人にとってのみ、意味のあるものですね？ だとしたら、私たちがなぜ議論する必要があるのですか？」

最新論文の細かい結果の重要性は徐々に薄れていく、ということに関しては、懐疑論者は正しいです。とは言うものの、ここ数年、途轍もなく難しい画像認識問題に深層ネットワークが挑み、その素晴らしい結果が立て続けに発表されています。2100年の、コンピュータビジョンの歴史を綴る歴史家のことを想像してください。彼らはきっと、2011年から2015年（とさらに数年）を、深層畳込みネットワークによるブレークスルーの時代と位置づけるでしょう。それは2100年になっても、深層畳込みネットワークが通用するか否かとは無関係です。もちろん、ドロップアウトやReLUなどのアイデアが使用されているかどうかも関係ありません。それは、歴史の中でまさに今、重大な進化が起きているということを意味するのです。原子の発見や抗生物質の発明を目撃しているようなものだと思います。これは、歴史的な規模の発明と発見と私は信じています。したがって、詳細には踏み込みませんが、この瞬間も新たに発見されているアイデアを確認しておくことは重要なのです。

2010年のLRMD論文：2012年のStanfordとGoogleの研究者による論文からまず始めましょう*。この論文を、最初の4人の著者の苗字からLRMDと呼びます。LRMDはニューラルネットワークを用いてImageNetという画像分類問題を解いています。ImageNetは画像認識の難問です。彼らが使用した2011年のImageNetのデータは、2万カテゴリに分類された1600万枚のフルカラー画像でした。これらの画像はウェブ上で集められ、Amazon Mechanical Turkのサービスにより分類されたものです。これがImageNetの画像の例です* 。

これらの画像の分類クラスはそれぞれ、玉縁装飾かんな、キュウリ科の植物の褐色根腐れを引き起こす菌類、加熱された牛乳、線虫です。問題を試したいのであれば、ImageNetの手工具のページをお薦めします。

*2012年のQuoc Le, Marc'Aurelio Ranzato, Rajat Monga, Matthieu Devin, Kai Chen, Greg Corrado, Jeff Dean, Andrew NgによるBuilding high-level features using large scale unsupervised learning。この論文で扱われているネットワークの詳細な構造は、これまで学んできた深層畳み込みネットワークと多くの点で異なります。しかし、大まかな視点で見ると、同じアイデアに基づいていることが分かります。

*これらは2014のデータセットのものです。2011年のデータとは少し異なります。しかし質的にはほとんど同じです。ImageNetの詳細は、Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, Li Fei-Feiによる2009年のImageNetの論文、ImageNet: a large-scale hierarchical image databaseを参照してください。

上記サイトでは、玉縁装飾かんな、木口用かんな、面取りかんなを始め、10種類程度のかんなの分類があります。あなたがどうかは分からないですが、私はこれらの道具を自信を持って区別できません。これは明らかにMNISTよりも難しい画像認識問題です！ LRMDのネットワークはImageNet画像に対して、15.8 %の分類精度を得ています。あまり精度が良くないように聞こえるでしょう。でも、LRMDより以前の最高結果は9.3 %だったのです。そこから考えると大きな進展です。この進展は、ImageNetのような難しい画像認識問題に対して、ニューラルネットが強力なアプローチであることを示しています。

2012年のKSH論文：2012年に Krizhevsky, Sutskever, Hinton(KSH)がLRMDの研究を追って論文*を出しました 。KSHは、ImageNetデータのサブセットを使って、深層畳み込みニューラルネットワークの訓練とテストをしました。このサブセットは、機械学習の人気の大会であるILSVRC(the ImageNet Large-Scale Visual Recognition Challenge)から引用したものです。この大会で使用されるデータセットを用いると、他の最先端アプローチと比較することができます。ILSVRC-2012の訓練データ・セットは、1000の分類クラスからなる120万のImageNet画像です。検証とテストのデータは、1000の分類クラスからなるそれぞれ5万と15万の画像です。

*2012年のAlex Krizhevsky, Ilya Sutskever, Geoffrey E. HintonによるImageNet classification with deep convolutional neural networks

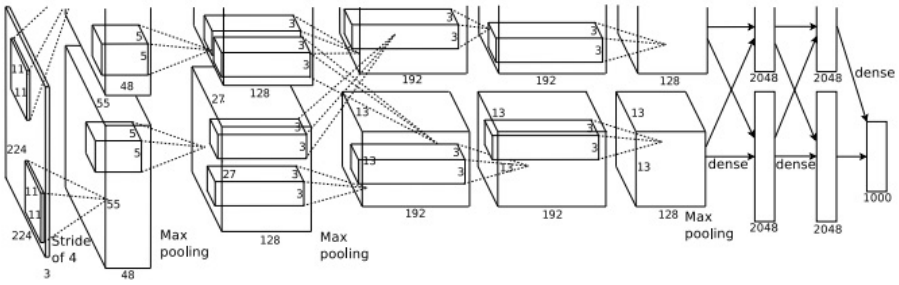
ILSVRCの難しい点はImageNetの画像が複数の物体を含んでいることです。ラブラドールレトリバーがサッカーボールを追いかけている画像を思い浮かべてください。ImageNetによる「正しい」分類クラスは、きっとラブラドールレトリバーでしょう。ここで、この画像をサッカーボールと分類した時に、このアルゴリズムはペナルティを受けるべきでしょうか？ この曖昧性を考慮して、実際のImageNetの分類は 5 つ選んだ分類の中に正解があれば、アルゴリズムは正しいと判定することになっています。この上位 5 の分類を使う基準に則ると、KSHの深層畳み込みネットワークは84.7 %の精度を達成しています。ちなみに次点のネットワークの精度は73.8 %でした。分類の正確性の基準をもう少し厳しいものを適用すると、KSHのネットワークの分類精度は 63.3 %となります。

KSHのネットワークは後続の研究に大きな影響を与えたネットワークであるので、今後の参考のために簡単に描写してみようと思います。KSHの方が手が込んでいますが、本章でこれまで私たちが訓練してきたネットワークにとてもよく似ています。KSHは深層畳み込みニューラルネットワークであり、2台のGPU上で訓練を行っています。2台のGPUを使用した理由は、使用しているGPU（NVIDIA GeForce GTX580)に原因があります。ネットワーク全体を保持するためには、このGPUのオンチップメモリ

が足りないのです。そのため、ネットワークを二分割して2台のGPUに分けて搭載しています。

KSHのネットワークには7つの隠れ層があります。前方の 5 つの隠れ層は畳み込み層（幾つかはMaxプーリング付き）で、次の 2 層は全結合層です。出力層は 1,000 ユニットからなるソフトマックス層で、1,000 の分類クラスに対応しています。下図がKSHの論文*から引用したネットワークのスケッチです 。詳細は下に記述します。多くの層が 2 つのGPUに対応するために 2 部分に分割されていることに注意してください。

*Ilya Sutskeverに感謝します。



入力層は、3 × 224 × 224 ニューロンを含み、224 × 224 の画像のRGB値を表します。以前述べたように、ImageNetの画像は解像度が異なることを思い出してください。ニューラルネットワークの入力層のサイズは固定なので、このままでは問題が起きます。そこで、KSHは各画像を拡大縮小して、短辺が長さ 256 となるように調整しています。次に、その画像の中央 256 × 256 の領域を切り取ります。最後に、その 256 × 256 領域の中から、ランダムに 224 × 224 の部分画像（水平反転も含む）を抜き出します。このランダムに抜き出す操作により、訓練データを拡張し、過適合を防いでいます。この一連の操作はKSHのような巨大なネットワークの場合、有効です。224 × 224 の画像がネットワークの入力に使われています。大抵の場合、抜き出された画像は目的の物体を含んでいるはずです。

KSHの隠れ層の話に移ります。1つ目の隠れ層はMaxプーリング付きの畳み込み層です。この層はサイズが 11 × 11 の局所受容野を、ストライド長さ 4 ピクセルで使います。全体で 96 の特徴マップとなります。特徴マップは各 48 の2グループに分割され、始めの 48 の特徴マップは片方のGPUに置かれ、後半の 48 の特徴マップはもう片方のGPUに置かれます。この層含めて後層でも、Maxプーリングは 3 × 3 の領域で行われます。しかしプーリング領域は重複が許されており、実際 2 ピクセルしか離れていません。

2つ目の隠れ層もMaxプーリング付きの畳み込み層です。5 × 5 の局所受容野を使い、全体で 256 の特徴マップを持ち、各GPUに 128 ずつ分割され置かれます。ここで特徴マップは、前層の出力の 96 チャネル全

てを利用するのではなく、48 の入力チャネルのみ使うことに注意してください（これは通常の操作ではありません）。なぜかと言うと、同じGPUからしか入力を受け取れないからです。この点で、これまで私たちが学んできた畳み込み層とは異なります。ただし、根底に流れる基礎的なアイデアはやはり同じです。

3、4、5層目の隠れ層も畳込み層ですが、前層までと異なりMaxプーリングは行いません。各パラメータは次のようになっています。(3) 特徴マップ 384 個、局所受容野のサイズ 3×3 、入力チャネル 256 、(4) 特徴マップ 384 個、局所受容野のサイズ 3×3 、入力チャネル 192 、(5) 特徴マップ 256 個、局所受容野のサイズ 3×3 、入力チャネル 192 、3層目は、特徴マップが全ての入力チャネルを使うために、（図に示すように）GPU間通信を行うことに注意してください。

6、7層目の隠れ層は 4,096 のニューロンからなる全結合層です。

出力層は 1,000 ユニットのソフトマックス層です。

KSHのネットワークは多くのテクニックを利用しています。シグモイドやtanhを活性化関数に使う代わりに、ReLUを使って訓練を高速化しています。もともと巨大な訓練データセットを使っているとはいえ、KSHのネットワークには約6000万のパラメータがあるので、過適合しやすいです。これを克服するために、上述の通りランダムに抜き出す戦略により、訓練データを拡張したのです。さらにL2正規化の一種や、ドロップアウトを用いて過適合を抑制しています。ネットワークの訓練は、モメンタムを用いたミニバッチ確率的勾配降下法で行います。

以上がKSH論文の重要アイデアの概要です。いくつかの詳細は省きました。論文で確認してみてください。またAlex Krizhevskyによるcuda-convnet（とその後継情報）を見るのもよいでしょう。コード実装に関するたくさんのアイデアが載っています。Theanoベースの実装は発表*されており、ここでそのコードが入手できます。そのコードは複数GPUの使用するため少し複雑ですが、この章で見てきたものにそっくりです。Caffeフレームワークの中でもKSHネットワークが実装されています。Model Zooを見てください。

*2014年のWeiguang Ding, Ruoyan Wang, Fei Mao, and Graham TaylorによるTheano-based large-scale visual recognition with multiple GPUs

2014のILSVRC: 2012年から急激な発展が続きました。2014年のILSVRCコンペティションを概観しましょう。2012年の場合と同じく、訓練データは 1,000 の分類クラスからなる 120 万の画像です。正解の基準は、画像に対して分類した上位 5 カテゴリの中に正しいラベルが含まれていることです。勝者はGoogle*を主体としたチームで、22 層の深い畳み込みネットワークを使用していました。彼らは自身のネットワークを、

*2014年のChristian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke,

LeNet-5に対するオマージュとしてGoogLeNetと名付けました。GoogLeNetは上位5分類基準の精度で評価すると 93.33 %でした。これは2013年の勝者の記録(Clarifaiは 88.3 %)と2012年の勝者の記録(KSHは 84.7 %)を大幅に上回っています。

Andrew RabinovichによるGoing deeper with convolutions

GoogLeNetの 93.33 %という精度はどのくらい良い結果なのでしょうか？2014年に研究者がILSVRCに関するサーベイ論文*を書いています 。彼らは、ILSVRCに人間が挑むとどのような精度になるか、という問題を提起しました。これを調べるために、彼らは人間がILSVRC画像を分類するためのシステムを作りました。著者の1人であるAndrej Karpathyが有益なブログの投稿を行っています。その内容は、人間がGoogLeNetのパフォーマンスにたどり着くのは難しいというものでした。

*2014年のOlga Russakovsky, Jia Deng, Hao Su, Jonathan Krause, Sanjeev Satheesh, Sean Ma, Zhiheng Huang, Andrej Karpathy, Aditya Khosla, Michael Bernstein, Alexander C. Berg, Li Fei-FeiによるImageNet large scale visual recognition challenge。

> ...画像に対して1000のカテゴリの中から5カテゴリを選んでラベリングするタスクは極めて難しかったです。ILSVRCに日常的に取り組んでいて、分類に馴染みがある研究所の友人にとってさえ難しいものでした。一番初めは、Amazon Mechanical Turkのサービスを利用しようと考えていました。その後、学部生にバイトさせる方針に切り替えました。最後には、私たちの研究所の（ラベリングを専門とする）人たちの中で組織を作ることにしました。そこで、GoogLeNetの予測のために使ったインターフェースに修正を加えて、分類のカテゴリ数を1000から100へ減らしました。それでもまだまだ難しすぎました。皆、分類を間違えて 13-15% の誤差を出し続けたのです。結局、GoogLeNetの精度に接近するためには、私自身が長時間座り続けて長く辛い訓練を行った上で、注意深くラベリングするしかない...と気づきました。訓練し始めた当初は、分類作業を1分に1回しかできませんでした。しばらく経つと、次第に高速にできるようになり...画像によってはすぐに認識できる状態となりました。一方、画像（細かい犬種や鳥の種類、猿の種類を示す画像など）によっては、集中して数分取り組まないといけませんものもありました。しかし更に時間が経つと、訓練画像をもとにして...犬種の識別など、かつて難しかったタスクを容易に行えるようになりました。GooLeNetは分類誤差が 6.8% でしたが、...私の分類誤差は最終的には 5.1% となり、約 1.7% ポイント勝りました。

つまり、専門家の人間が苦痛を伴う努力をして初めて、深層ニューラルネットワークを僅かに上回れるということです。実際、2人目の専門家は、サンプル数の少ない訓練画像で訓練して挑んだものの、12.0 %の誤差までしか到達できなかったとKarpathyは報告しています。これは

GoogLeNetの性能に大きく劣ります。間違えた問題の半数は、選択肢にさえ真のラベルを選べなかったそうです。

この報告は驚異的です。この研究以降、実は幾つかのチームが 5.1% を**超える**結果を報告しています。これらの結果を受けて、「システムが人間を超える視覚を手に入れた」とメディアで報道がなされました。たしかに結果は本当に素晴らしいものですが、注意することとして、「人間を超える視覚を手に入れた」、というのは誤解です。ILSVRCはとても制約の大きい問題です。ウェブから集めた画像を分類しているので、様々な応用目的で使う画像とは必ずしも一致しません。もちろん評価指標である、上位 5 を基準にするというのも非常に恣意的です。画像認識、さらに広く言うとコンピュータビジョンの問題を完全に解いたとはまだとても言えないのです。ただし、難問に対して、たった数年でこの結果が得られたというのは、とても励みになります。

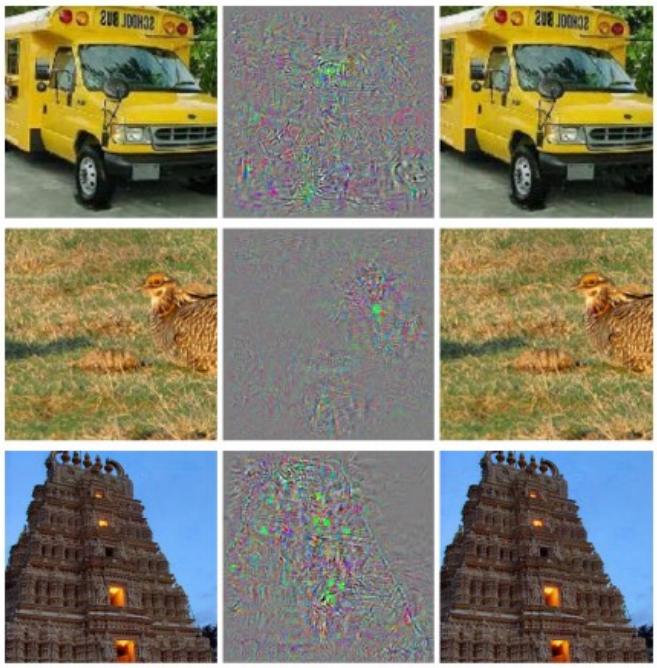
他の活動: これまでImageNetを見てきました。しかし、他にもニューラルネットワークを使用して画像認識する活動があります。興味深い近年の研究結果を簡単に紹介します。

Googleによって生まれた実践的な結果があります。彼らは深層畳み込みネットワークを、Google Street Viewの景色の中の数字認識の問題に適用*しました 。論文の中では、1億の路地番号を検知して自動的に文字に起こすタスクが、人間と同等の精度で行われたと報告されています。さらに、このシステムは非常に高速です。Street Viewのフランス国内の全ての画像において、路地番号全てを1時間以内に文字に起こしたのです！「生成した路地番号のデータセットを利用すると、Google Mapの地理情報の質が驚異的に向上しました。他の地理情報源を従来持たなかった幾つかの国では、特に影響が大きかったです」と彼らは述べています。また、「短文の視覚文字認識の問題をこのモデルでは解決したと思っています。このモデルは多くのアプリケーションに利用できます」とも述べています。

*2013年のIan J. Goodfellow, Yaroslav Bulatov, Julian Ibarz, Sacha Arnoud, Vinay Shetによる Multi-digit Number Recognition from Street View Imagery using Deep Convolutional Neural Networks

これらは素晴らしい結果だと私も思っています。しかし、別の研究では、私たちは本質をまだ理解できていないという指摘がなされています。例えば、2013年の論文* では、深層ネットワークが盲点を持ち、挙動が不安定となる様子が示されています。下図を見てください。左は、ネットワークによって正しく分類されたImageNet画像です。右は、少し外乱（中央の画像）が挿入された画像です。ネットワークは右の画像を**誤って**分類しました。特別な画像だけでなく、どの画像にもそのような"adversarial"な画像が存在すると著者は指摘しています。

*2013年のChristian Szegedy, Wojciech Zaremba, Ilya Sutskever, Joan Bruna, Dumitru Erhan, Ian Goodfellow, Rob FergusによるIntriguing properties of neural networks。



この結果には戸惑います。論文では、KSHのネットワークの場合をもとにネットワークを組み立てていました。KSHのネットワークの種類は、とても幅広く使われているのです。そのようなニューラルネットワークの計算する関数は原理的に連続的であるはずなのに、上記の結果は極端な非連続性を示すものでした。しかもこれは、私たちの直感に反する非連続性です。これは気がかりです。何が非連続性の要因なのかがまだ良くわかっていないのです。誤差関数の何かに関係しているのでしょうか？ 活性化関数？ ネットワークの構造？ それとも他の何かでしょうか？ まだ分かっていません。

実は、これらの結果に悲観する必要はありません。adversarialな画像は普遍的に存在しますが、実践では発生しにくいのです。

> adversarialな存在はネットワークの高い汎化性能に矛盾するように思えます。実際、ネットワークが汎用性を獲得できた場合、通常の画像と一見区別できないようなadversarialな存在によって騙されうるのでしょうか？ これに対する反論は、adversarialな画像はめったに発生しないため、テストセットの中には観測されないというものです。しかし、テストセットは有理数のように詰まっているため、本質的にはどんなテストケースにも存在するはずです。

このような類の結果が近年発見されるのは、本質的にニューラルネットワークを理解していないことの現れでしょう。もちろん、このような結果の発表されることで、追跡調査が盛んに行われ、研究が進みます。他にも例えば、最近の論文* では、人間にはホワイトノイズに見えるのに、ネットワークは確信を持って既知のカテゴリに分類するような画像を生成する結

*2014年のAnh Nguyen, Jason Yosinski, Jeff CluneによるDeep Neural Networks are Easily

果が報告されています。ニューラルネットワークとその画像認識法を理解するまでにはまだまだ時間がかかりそうです。

Fooled: High Confidence Predictions for Unrecognizable Images

しかし現状を俯瞰すると、励みになる結果が多いです。ImageNetのようにとても難易度の高いベンチマークに対する急速な進展がありました。また、StreetViewの路地番号を認識するという現実世界の問題に対処する例も確認しました。これらは励みになります。ただし、ベンチマークに対する良い結果や現実への応用を追うのみでは不十分だと思います。まだ、adversarialな画像の例など、私たちがほとんど理解できていない本質的な現象があります。そのような本質的な問題は、現在も研究されている途中であり、画像認識の問題を完全に解くには、まだまだ発展途上だと言えます。別の言い方をすると、今後の研究には余地がたくさん残っており、課題としてはとても魅力的なのです。

深層ニューラルネットワークに対する他のアプローチ

本書では、MNISTの数字分類というただ一つの問題に専念してきました。MNISTの問題は味わい深くて、重要なアイデアを理解することができました。それは確率的勾配降下法や、逆伝播、畳込みネットワーク、正規化などです。しかし、この問題の扱う領域は広くはありません。ニューラルネットワークの文献を読むと、これまで議論に登場しなかった多くのアイデアに出会うはずです。例えば、再帰型ニューラルネットワークやボルツマンマシン、生成モデル、転移学習、強化学習、. . . ！ ニューラルネットワークはとても広い分野なのです。しかし、その多くの重要なアイデアは、実を言うとこれまで既に議論してきたアイデアの応用でしかありません。なので、少しの努力で理解できるはずです。このセクションでは、あなたのまだ知らないニューラルネットワークについて少しお見せしましょう。本書の範囲を越えてしまうため、詳しい議論や網羅的な議論は行いません。むしろ、各概念を直感的に理解できるように、これまで学んできたことに関連させて学んでいきます。セクションを通じて、他のソースへのリンクを示しておきます。このリンクを辿れば、さらに学ぶことができます。もちろん、リンクの多くは今後すぐに古びてしまい、最先端の文献を常に探すようになるでしょう。それでも、根底のアイデアは永続的に残るはずです。

再帰型ニューラルネットワーク (RNN): これまで使ってきたフィードフォワードのネットワークでは、後方のニューロンの活性化を決める入力は1つでした。これはとても静的な構造です。ネットワークの全ては固定され、まるで凍結された結晶のようです。しかし、動的に変化し続けるネットワーク要素を想定することもできます。例えば、隠れ層の振る舞いが、1

つ前の隠れ層の活性化によってのみ決まるのではなく、さらに以前の活性化にも影響されるような場合も考えられます。実際、あるニューロンの活性化が、そのニューロン自身の以前の活性化により定義されることもありそうです。単なるフィードフォワードネットワークでは、そんなことは起きません。さらに別の場合を考えてみると、隠れ層や出力層の活性化がネットワークの現在の入力だけでなく、もっと以前の入力によっても影響されるパターンも想定できます。

このような時間経過を考慮に入れたニューラルネットワークは、**再帰型ニューラルネットワーク**もしくは**RNN**として知られています。この再帰型ネットワークの数学的な定式化は、色々なやり方があります。WikipediaのRNNのページを眺めると数学的モデルの雰囲気が掴めます。この執筆段階では、上記ページには13以上の異なるモデルが掲載されています。数学的詳細は置いておき、RNNを大雑把に表現すると、時間経過による動的変化の概念が含まれるニューラルネットです。このモデルは時間により変化するデータの分析や処理に役立ちます。そのようなデータや処理は、音声や自然言語などの問題に自然と含まれています。

現在のRNNの使われ方の1つは、アルゴリズムの概念やチューリングマシンやプログラミング言語などの概念を、ニューラルネットワークに学習させるというものです。2014の論文では、（とても、とてもシンプルな！）Pythonプログラムの文字列を入力としてRNNに渡して、プログラムの出力を予測するのにRNNが使われました。砕けた言い方をすると、そのネットワークはPythonプログラムを「理解する」ことを学んでいます。同じく2014年の論文では、ニューラルチューリングマシン（NTM）と呼ばれるものを開発するための足がかりとして、RNNを使っていました。このNTMは、勾配降下によって汎用計算機の構造を学習するものです。彼らは、NTMを訓練して、ソートやコピーなどの簡単な問題に対するアルゴリズムを推測させました。

現状では、これらはシンプルすぎるトイモデルにとどまっています。`print(398345+42598)`というPythonプログラムを実行することを学習しただけでは、一人前のPythonインタープリターとは言えません。このアイデアを推し進めると、何が実現できるようになるかは明らかではありません。しかし、結果は実に面白いです。歴史的には、ニューラルネットワークは既存のアルゴリズムによるアプローチが手を焼いていたパターン認識問題を上手く解決してきました。対照的に、既存のアルゴリズムによるアプローチは、ニューラルネットワークが得意でない問題を上手く対処できます。今日の誰も、ウェブサーバやデータベースのプログラムにニューラルネットワークを使いません！ ニューラルネットワークと伝統的なアルゴリズムによるアプローチの強みを両方取り入れたモデルが作れれば、素晴ら

しいと思います。RNNやRNNに端を発するアイデアはきっと、その目標に向けた良い足がかりとなるでしょう。

RNNは他にも多くの問題に利用されています。その中でも特に音声認識において、有効活用されています。例えば、RNNをもとにしたアプローチは 音素認識の精度向上に貢献しています。また、会話中の言語モデルの改善にも寄与しています。言語モデルが良ければ、発音の似ているフレーズ間でも区別できます。例えば、言語モデルにより"to infinity and beyond" の方が "two infinity and beyond"よりも起きやすいことが分かります。RNNは言語に関するベンチマークでも、貢献してきたのです。

これらは、深層ニューラルネットワークの音声認識における成果の一部です。例えば、他の深層ネットワークに基づくアプローチは 大語彙連続音声認識（LVCSR）で驚異的な結果 を残しました。また、深層ネットワークベースの別のシステムはGoogleの Androidオペレーティングシステムに採用されるほど精度が高いです (関連する技術動向は Vincent Vanhouckeによる2012から2015の論文を参照してください)。

RNNによって実現可能な具体例を述べてきましたが、どう実現するかについては触れてきませんでした。フィードフォワードのネットワークで学んできたアイデアがRNNでも同様に使われているので、きっとあなたに驚きはないでしょう。勾配降下法と逆伝播を単純に修正するだけで、RNNの訓練は可能となります。他の多くのアイデアも、RNNで有効です。例えば、正規化のテクニック、畳み込みや活性化関数、コスト関数などです。本書で見てきた多くのテクニックが、RNNに適用可能なのです。

長期短期記憶ユニット（LSTM）：RNNの難しさの1つは、モデルを訓練するのがとても大変なことです。深層フィードフォワードネットワークよりもさらに訓練が難しいのです。その理由は5章で議論した勾配の不安定性に起因します。思い出してください。これは、後ろの層に伝播するにつれ、勾配がどんどん小さくなっていくという問題でした。これにより、前方の層ほど学習が遅くなるのです。RNNではこの問題がさらに悪化します。なぜかと言うと、RNNでは勾配は単に前方の層へ伝播するだけではなく、時間経過に従い後方へも伝播するからです。ネットワークが長時間動いている場合、勾配は極めて不安定になり、学習は難しくなるでしょう。幸運なことに、長期短期記憶ユニットとして知られるアイデアをRNNへ取り入れることができます。このユニットは 1997年にHochreiterとSchmidhuberが、勾配の不安定性に取り組むために考案したものです。LSTMにより、RNNの訓練で良い結果を簡単に得ることができるよう

になったため、最近の論文の多く(私が上でリンクした論文も含め)では、LSTMもしくは関連するアイデアを利用しています。

Deep Belief Network、生成モデル、ボルツマンマシン: 近年のディープラーニングの流行の発端は2006年です。そのきっかけは、ニューラルネットワークの一種として知られる**Deep Belief Network** (DBN)*の訓練方法に関する論文でした。DBNはその後数年は影響力がありましたが、人気は徐々に衰えていきました。その間、フィードフォワードモデルと再帰型ニューラルネットワークが流行っていきました。現在は流行っていないのですが、DBN自体は興味深い性質を幾つか備えています。

*2006年のGeoffrey Hinton, Simon Osindero, Yee-Whye TehによるA fast learning algorithm for deep belief netsと合わせて、2006年のGeoffrey HintonとRuslan Salakhutdinovによる Reducing the dimensionality of data with neural networks を確認してください。

DBNが興味を引く一つの理由は、**生成モデル**と呼ばれるものの一例だからです。フィードフォワードネットワークでは、入力の活性化状態を決まることで、ネットワークの後方の層の特徴の活性化状態も決まります。DBNのような生成モデルも、同じように使うことができます。しかし別の使い方として、あるニューロンの特徴を特定したら、「ネットワークを逆伝播」させ、入力の活性化値を生成することができます。もっと具体的に言うと、手書き数字について学習したDBNは、人の手書き数字のような画像を生成可能であるということです。抽象的に言い換えると、DBNは書くことを学べるのです。この点において、生成モデルは人間の脳のようです。数字を読むだけでなく、数字を書くこともできるのです。Geoffrey Hintonはこのことを、「形状を認識するために、まず画像を生成するのです」という記憶に残るフレーズで表現しています。

DBNが好奇心をそそる2つ目の理由は、DBNで教師なし学習や半教師あり学習が可能であるからです。例えば、画像データを入力とする学習を行う時に、訓練画像にラベルがなくても、DBNは他の画像についての有効な特徴を学習できます。半教師あり学習は科学的にも、(もし十分上手く動作しているなら)実用的にもとても興味深いです。

こうした魅力的な性質を持っているのに、なぜDBNはディープラーニングと比べて人気がないのでしょうか？ 理由の1つは、フィードフォワードモデルや再帰型ネットワークモデルが、画像認識や音声認識のベンチマークで、目を見張るほどの成果を出したからです。これらのモデルへ注目が集まるのも驚きはないですし、納得できます。DBNに人気がないのは残念ですが、当然の帰結と言えます。アイデアの市場は、しばしば勝者総取り方式であり、その瞬間において全ての注目が一部に集まる性質があります。その時、流行っていないアイデアに取り組もうとするのは、とても難しいのでしょう。たとえそのアイデアが、長期的には利益が出ることが明らかだったとしても。私の個人的な意見では、DBNや他の生成モデル

は、潜在的には現在よりもさらに注目を集めてよいはずです。DBNや関連するモデルがいつの日か、現在流行のモデルよりも人気を得ても驚きません。DBNについて入門するには、この要約を見てください。この記事も有益です。**当然**、DBNがメインではないのですが、DBNの重要な要素である制約ボルツマンマシンに関する有益な情報が含まれています。

他のアイデア: ニューラルネットワークとディープラーニングには他にどんなアイデアがあるでしょう？ ええ、他にも魅力的な研究がたくさんあります。ニューラルネットワークの研究が盛んに行われている分野として、自然言語処理 (この有益なレビュー論文を確認してください)や、機械翻訳、そして音楽情報学などにも驚きの応用もあります。今挙げた以外の領域にもあります。背景知識のギャップを埋める必要はありますが、本書を読み終えたら、最近の研究を読み始めることができるはずです。

このセクションの締めに、特別楽しめる論文を紹介しましょう。この論文は深層畳み込みネットワークと強化学習を組み合わせて、ビデオゲームを上手くプレイ (こちらの追加情報も参照してください)するよう学習するものです。畳み込みネットワークを使って、ゲームスクリーンのピクセルデータを単純化し、特徴へ変換します。それをもとに「左へ移動する」「下へ移動する」「発泡する」などの行動を決定します。一番興味深いのは、単一のネットワークで、7つの異なるクラシックなビデオゲームを、とても上手くプレイするよう学習したことです。そのうち3つのゲームでは、人間の専門家のパフォーマンスで上回りました。これは離れ業のように聞こえるでしょう。この論文は "Playing Atari with reinforcement learning" というタイトルで注目を集めました。定性的に考えると、このシステムは生のピクセルデータを入力とするだけです。つまり、ゲームのルールすら知らないのです！ 複雑なルールで構成される様々な環境において、単なるピクセルデータのみを入力として、高品質な意思決定を行っているのです。極めて巧妙なのです。

ニューラルネットワークの未来

意図で駆動するユーザインタフェース: 気短な教授が、どうすればよいか分かっていない学生に次のように伝える古いジョークがあります。「私が言うことを聞かなくていい。私の**意図**を読み取ってくれ」と。昔からコンピュータは、ユーザの意図が分からず戸惑っている学生のような存在でした。しかし、現在変化しつつあります。私が初めてGoogle検索でスペルを間違えてしまった時に、Googleが "Did you mean [正しいクエリ]?"と伝えてきて、対応する検索結果を提示したときの驚きをまだ覚えています。GoogleのCEOのLarry Pageは、あなたの検索の意味を正確に

理解し、まさに望むものを提供する完璧な検索エンジンについて記述を残しています。

これは**意図駆動のユーザインタフェース**の理想像です。この理想像では、ユーザによる文字ベースのクエリの代わりに、ぼんやりとした入力からユーザの意図を機械学習で識別して、その意図に対して機能を提供するのです。

意図駆動のインターフェースのアイデアは、検索以外にも幅広く適用できます。数十年以内に、何千もの会社が機械学習を使って、ユーザの真の意図を識別・行動するインターフェースを作り上げるでしょう。私たちは意図駆動のインターフェースの早熟な例を既に目の当たりにしています。AppleのSiri、Wolfram Alpha、IBMのWatsonや、写真やビデオに注釈をつけるシステムなどです。

これらの製品の大半は失敗するでしょう。見事なユーザインターフェースデザインは難しいのです。しかし、強力な機械学習技術を使うことで、素晴らしいユーザインターフェースを作り上げることを期待しています。ユーザインターフェースのコンセプト自体がもともと悲惨だと、機械学習が素晴らしくても意味がないでしょう。しかし、それ以外の製品は成功すると思います。そのうち、私たちのコンピュータへの関わり方に大きな変化が起こるでしょう。つい最近まで、2005年くらいまででしょうか、ユーザはコンピュータとやり取りするのに正確な操作を必要としていました。実際、当時のコンピュータリテラシの意味というのは、コンピュータには想像力がないということを理解することでした。一文字分セミコロンの位置を間違えただけで、コンピュータは意図通りに動きません。しかし、数十年後には、上手く作動する意図駆動のユーザインターフェースが開発されることを私は期待します。その結果、コンピュータとの関わり方が劇的に変わるでしょう。

機械学習、データサイエンス、イノベーションの好循環: もちろん、機械学習は意図駆動のユーザインターフェースだけに使われるのではありません。注目の応用先はデータサイエンスです。そこでは、機械学習がデータの中の「既知の未知」を発見するのに使われます。この分野は既に流行っており、文献はたくさんあるので、私は多くを語りません。しかし、この流行りの行き着く先について、一つ述べておきます。長期的に見ると、機械学習でのブレークスルーはおそらく、アイデアや発明のブレークスルーではありません。一番大きなブレークスルーは、データサイエンスか他の分野かにおいて、機械学習研究が利益を出すことだと思います。ある会社が1ドルを機械学習研究に投資して、すぐに1ドルと10セントを回収したとすると、多くのお金が機械学習研究につぎ込まれるでしょう。言

い換えると、機械学習は新たな巨大市場を生成し、技術の成長分野となりうるのです。その結果、深い専門技術をもった多くのチームと、途轍もなく巨大なリソースが誕生します。最終的に、それらが機械学習を更に促進させ、さらなる市場と機会を作り、イノベーションの好循環の図となるはずです。

ニューラルネットワークとディープラーニングの役割：機械学習について、技術にとって新たな機会を創出するだろうとこれまで広く述べてきました。ニューラルネットワークとディープラーニングの役割は、特にどのようなものでしょうか？

この疑問に答えるには、歴史を振り返るのが助けになるかもしれません。1980年代は多くの実験がなされており、ニューラルネットワークに対する楽観的な見方が広がっていました。逆伝播が広く知られるようになってからは、特に楽観的でした。実験が減っていき、1990年代に入って、他のサポートベクターマシンなどのテクニックにバトンを渡しました。今日、ニューラルネットワークは再び、勢いに乗っています。様々な記録を打ち立て、多くの問題に対する他の解決法を負かしました。しかし、明日には別の新しいアプローチが登場して、ニューラルネットワークを再び淘汰する可能性を誰が否定できるでしょうか？ もしくは、代替技術がなくても、そのうちニューラルネットワークの進化は停滞するのではないでしょうか？

このような可能性を考えると、ニューラルネットワークだけを予想するよりも、広く機械学習の未来を考えるほうが簡単です。そもそも私たちは、ニューラルネットワークを深く理解できていません。なぜ、ニューラルネットワークは十分に汎化できるのでしょうか？ パラメータ数がとても多いときに、過適合を避けるにはどうすればよいのでしょうか？ 確率的勾配降下法はなぜ上手く動作するのでしょうか？ データセットのサイズが変化したとき、ニューラルネットワークはどの程度上手く動作するでしょうか？ 例えば、ImageNetが 10 倍に拡張された場合、他の機械学習技術と比較して、ニューラルネットワークの性能は向上するでしょうか？それとも低下するでしょうか？ これらは全てシンプルで本質的な疑問です。そして現在、私たちはこれらの疑問への答えを持ち合わせていません。ですので、機械学習の未来におけるニューラルネットワークの役割を述べるのは難しいのです。

1つ予測します。ディープラーニングは定着すると私は思います。階層的な概念を学習する能力や、層を組み立てて抽象化を行う能力は、世界を構成している本質そのものだと思います。これは、明日のディープラーニングのモデルが、今日のモデルと同じであることを意味しているのではありません。構造の中の構成要素や学習アルゴリズムに、大きな進展が

あるはずです。それはきっと、あまりに目覚ましい変化なので、その結果のシステムがニューラルネットかどうかをもはや気にしていないと思います。しかし、きっとディープラーニングは使われているでしょう。

ニューラルネットワークとディープラーニングはすぐに人工知能となるか? 本書ではニューラルネットワークを画像分類など特定の目的に使ってきました。私たちの野望を大きくして、尋ねてみましょう。汎用に考えるコンピュータになりえますか？ ニューラルネットワークとディープラーニングは、(汎用) 人工知能 (AI) の問題を解けますか？ そしてその場合は、ディープラーニングがさらに発展すれば、すぐに汎用AIを実現できますか？

これらの疑問に対して包括的に取り組むためには、別の本を書かなくてはいけません。代わりに、一つ考察してみましょう。この考察は、コンウェイの法則として知られるアイデアに基づいています。コンウェイの法則とは、次のことを主張しているものです。

> システムを設計する組織は、その構造をそっくり真似た構造の設計を生み出してしまう。

これは例えば、ボーイング747航空機の設計が、当時のボーイング社と業者の組織的な構造を反映していることを示唆します。もしくは、さらにシンプルな具体例として、複雑なソフトウェアアプリケーションを作る会社を考えてみます。アプリケーションのダッシュボードに、ある機械学習アルゴリズムを取り入れようとする場合、ダッシュボードの製作者は、その会社の機械学習の専門家と話をした方がよいでしょう。コンウェイの法則は、この例を大規模にしたものに相当します。

コンウェイの法則を初めて聞いた時、多くの人は「当たり前でしょ？」と反応するか、「間違ってない？」と反応します。コンウェイの法則が間違っている、という二つ目の指摘に対して意見を述べさせてください。この指摘の背景にはきっと、次のような質問が想定されているのでしょう。「ボーイングの経理部署が、747の設計のどこに現れていますか？」、「管理部署はどこですか？」、「ケータリングを行う部署は？」と。確かに、これらの組織部署は747のどこにも、現れていません。しかし、コンウェイの法則における組織とは、設計とエンジニアリングに明らかに関わる部署のみを意図していると理解すべきなのです。

コンウェイの法則が、平凡で当たり前のことであるという指摘はどうでしょうか？ この指摘は部分的には正しい気がします。しかし、人材配置に失敗している組織に対しては、コンウェイの法則は成り立たないと思っています。新製品を開発するチームに、年配者しかいなかったり、逆に若者

のみで専門性を持つものがいなかったりする状況があります。これは単純に、製品のターゲットと開発チームとのミスマッチです。時に人がこの法則を無視して人材配置する場合があるのです。

コンウェイの法則は、私たちが構成要素とその組立て方をよく理解している、システムの設計とエンジニアリングに適用されます。人工知能の開発に直接には適用できません。なぜかと言うと、AIはまだそのような問題ではないからです。どんな構成要素からなるのか知りません。実際、AIに関する基礎的な問いすら分かっていないのです。つまり、現時点でAIは工学の問題というよりは、科学の領域の問題と言えます。ジェットエンジンや空気力学の原理を知らない状態で、747を設計する状況を想像してください。あなたは、どんな種類の専門家を組織に雇えばいいか分からいないでしょう。Wernher von Braunは「何をしているのか分かっていない場合には、基礎研究こそ行うべきである」と述べています。さて、工学ではなく科学の問題に適用できるバージョンのコンウェイの法則というのはあるのでしょうか？

この問いに対する洞察を得るために、薬の歴史を考えてみてください。古代において、薬はガレノスやヒポクラテスら体全体を研究する専門家の領域でした。しかし、知識が蓄積されて膨大になるにつれ、徐々に領域が細分化されていきます。その過程で、私たちは多くの奥深い発見をしてきました *。例えば、胚種説のような理論や、抗生物質の作用や、心、肺、静脈や動脈による心臓血管の形成などを考えてみてください。そのような深い洞察はやがては、疫学、免疫学、心臓血管系の学問などの部分的な分野となりました。したがって、私たちの知識の構造は、薬学の社会的構造を形作ったのです。これは免疫学においては、とても画期的なことでした。免疫系の存在に気づき、免疫系を研究する価値を見出したのは深い洞察からでした。そのようにして今、薬学という分野があるのです。専門家がいて、会議があって、賞までもあります。

*「深い」と重ねてしまい申し訳ありません。「深い」の意味の正確な定義を私はしませんが、大ざっぱ言うと研究分野全体に必要な教養のようなものを意図しています。逆伝播のアルゴリズムと胚種説はどちらもその好例です。

これは科学が確立するときに繰り返される普遍的なパターンです。薬学だけでなく、物理学、数学、化学でも同じです。そんな分野でも初めは、深いアイデアを2、3個だけ伴っていました。黎明期の専門家は、そのアイデアの全てを使いこなします。しかし時間が経つと、アイデアは一枚岩でなくなります。深いアイデアが新たに発見され、1人ではとても扱いきれなくなります。結果的に、その分野の社会構造は再編され、アイデアごとに分割されます。一枚岩の代わりに、分野の中に分野がある構造となり、複雑かつ再帰的で自己言及的な社会的構造となります。その組織は、私たちの深い洞察を反映するのです。**つまり、私たちの知識は科学の社会的構造を形成します。 しかし一方で、社会的構造は、私**

たちの発見を邪魔したり促進したりします。 これは科学におけるコンウェイの法則のアナロジーです。

さて、この話がディープラーニングやAIと何の関係があるのでしょうか？

ええと、AIの黎明期には、人々は次のようなやり取りをよく行っていました。「そんなに難しくないよ、"超特別な武器"を俺たちは既に持ってるんだから」と一方が言うと、「"超特別な武器"じゃあまだ全然ダメだよ」と否定するというものです。ディープラーニングは、同様の話題に使われる"最新の超特別な武器"です* 。この主張の昔のバージョンでは、論理やProlog（非手続き型プログラミング言語）、エキスパートシステムなど、その時代の強力なテクニックが使われていました。そのような主張に伴う重大な欠陥は、その"超特別な武器"がいかに強力かを示すことができないことです。もちろん、ここまで1章分を費やして、ディープラーニングがとんでもない難問を解く様子を確かめました。非常に刺激的で将来有望に見えます。しかし、それはPrologや Eurisko、当時のエキスパートシステムにも同じことが言えます。つまり、有望そうなアイデアがあるだけでは十分ではないのです。ディープラーニングがこれらの昔のアイデアよりも遥かに強力であることをどう証明するのでしょうか？ アイデアがどれほど強力で有望かを定量化する方法はあるのでしょうか？ そこでコンウェイの法則の出番です。コンウェイの法則では、雑でヒューリスティックな近似指標として、アイデアに関係する社会構造の複雑さを評価することを提案しています。

*興味深いことに、否定されている人はしばしばディープラーニングの最先端の人ではありません。例えば、下記のYann LeCunによる 思慮に満ちた投稿を見てください。この内容は先ほどの主張とは全然異なります。

さあ、2つの質問を考えましょう。1つ目は、ディープラーニングに関連するアイデアはどれほど強力なのかについてです。これは、社会的複雑性の指標で評価します。2つ目は汎用人工知能を実現するために、どれほど強力な理論が必要なのかについてです。

一つ目の質問についてまず考えます。今日のディープラーニングを眺めると、興味深くて進展も速いですが、比較的一枚岩の分野ではあります。幾つかの深いアイデアがあり、幾つかの会議があり、それらの会議は実質的な重なりがあります。そして、どの論文も同じ基礎的なアイデアを使っています。どの論文でも、確率的勾配降下法（かその変種）を用いて、コスト関数を最適化しています。それらのアイデアが上手くいくのは素晴らしいことです。しかし、深いアイデアをそれぞれ探索し、ディープラーニングを各方向へ押し広げるような、学問分野の下位部分はまだないようです。したがって社会的複雑さの指標に従って、誤解を恐れずに言えば、ディープラーニングはまだまだ浅い分野であると断言できます。まだ一人の人間が、全ての深いアイデアを習得することが可能な分野なのです。

今度は二つ目の質問に取り組みます。AIに到達するには、どれだけの複雑さと強力さを備えたアイデアが必要なのでしょうか？ もちろん、この質問への答えは誰にも分かりません。しかし、付録では、この質問に対する実験を行っています。どれだけ楽観的に言ったとしても、AIを作るには深いアイデアがとてもたくさん必要です。そしてそこに到達するには、そのような深い洞察を反映するような、複雑で驚くべき社会的構造が現れる必要があることをコンウェイの法則が示唆しています。ニューラルネットワークとディープラーニングには、このような複雑な社会構造はまだありません。したがって、ディープラーニングを使って汎用AIを開発するのには、まだ数十年かかるはずです。

ここまで、暫定的な議論を構築するのに苦労しました。この議論の流れは明らかであるように見えるものの、結論はぼんやりとしています。きっと、厳密さを追い求める人々をイライラさせていると思います。オンラインの文献を読んでいると、とても確信をもって主張をしている人が、AIに対して強い意見をもっているのに、実はその論理が薄っぺらいものであったり、存在しない証拠に依存していたりするのをよく見ます。私の率直な意見として、時期尚早で何も言うことはありません。次のような古いジョークがあります。科学者に発見はどれだけ先かを尋ねると、「10年（以上）」と答えます。その意図は「私にはいい考えがない」というものです。AIは、制御核融合や他の技術のように、10年プラス60年以上、その実現には時間がかかるでしょう。反面、ディープラーニングの能力は非常に強力で、その限界は今のところ分かっていません。そして、本質的な問題もまだまだたくさんあります。これはとてもクリエイティブで夢のある話だと思いませんか。

In academic work, please cite this book as: Michael A. Nielsen, "Neural Networks and Deep Learning", Determination Press, 2015

Last update: Thu Jan 19 06:09:48 2017