CS231n Convolutional Neural Networks for Visual Recognition

4.バックプロパゲーション(誤差逆伝播法)

序論

このセクションの学習にあたって．このセクションでは，連鎖法則の再帰的適用によって式の勾配を計算する方法であるバックプロパゲーション(誤差逆伝播法)を直感的に理解できる知識を身につけます．この過程とその微妙な違いを理解することは，ニューラルネットワークを理解し，効果的に開発，設計，デバッグするために不可欠です．

問題提起．本セクションで研究する中核的な問題は以下の通り．ある関数$f(x)$が与えられ(xは入力ベクトル)，xにおけるfの勾配(すなわち$\nabla f(x)$)を計算することに重点を置きます．
学習動機．この問題が興味深い第一の理由は，ニューラルネットワークの場合，fは損失関数(L)に対応し，入力xは訓練データとニューラルネットワークの重みで構成されることです．例えば，損失はSVMの損失関数であり，入力は訓練データ$(x_i, y_i), i = 1...N$と重みおよびバイアスW, bの両方です．機械学習では通常のことですが，学習データを与えられた固定されたものとして考え，重みを制御できる変数として考えてください．したがって，バックプロパゲーションを使って入力例x_iの勾配を簡単に計算することができますが，実際には通常，パラメータ(W, bなど)の勾配を計算するだけで，パラメータの更新に使用することができます．しかし，この授業の後半で見るように，x_iに対する勾配は，例えばニューラルネットワークが何をしているかを可視化したり，解釈したりする目的で役に立つことがあります．
受講する方の中で連鎖律による勾配の導出に慣れている方でも，少なくともこのセクションには目を通していただきたいと思います．このセクションでは，バックプロパゲーションを実値回路のバックワード・フローとして捉える，あまり発展していない見解が示されており，そこから得られる洞察は，授業全体に役立つかもしれません．

勾配についての簡単な説明と解釈

より複雑な形状の方程式の表記や慣例を身につけるために，簡単なことから始めましょう．
2つの数字の単純な乗算関数$f(x, y) = xy$を考えてみましょう．どちらの入力に対しても，偏微分を導出することはただの微積分の問題です．

$$f(x, y) = xy \quad \rightarrow \quad \frac{\partial f}{\partial x} = y, \quad \frac{\partial f}{\partial y} = x$$

解釈．微分の意味を考えてみましょう．微分とは，ある特定の点に近い無限小の領域間の，その変数に対する関数の変化率を示すものです．

$$\frac{\partial f(x)}{\partial x} = \lim_{h \to 0} \frac{f(x + h) - f(x)}{h}$$

技術的な注意点として，左辺の分数は，右辺の分数とは異なり，除算の意味ではありません．この表記は，関数fに対して演算子$\frac{\partial}{\partial x}$が適用され，別の関数(導関数)を返すことを示しています．
上の式を考える上で良い方法は，hが非常に小さいとき，関数は直線で近似され導関数がその傾きであるという点です．例えば，$x=4, y=-3$であれば，$f(x,y)=-12$となり，xで偏微分すると$\frac{\partial f}{\partial x}=y=-3$となります．これは，変数の値をわずかに増やしても，式全体では(負の符号のために)減少し，その3倍の値になることを示しています．これは，上の式($f(x+h)=f(x)+h\frac{df(x)}{dx}$)を並べ替えるとわかります．同様に，$\frac{\partial f}{\partial x}=4$であるから，$y$の値をある非常に小さな量$h$だけ増加させれば，関数の出力も(正の符号のために)$4h$だけ増加すると予想されます．

☞各変数の微分は，その値に対する式全体の感度を教えてくれます．

前述のように，勾配∇fは偏導関数のベクトルなので，$\nabla f=[\frac{\partial}{\partial x},\frac{\partial}{\partial y}]=[y,x]$となります．勾配は技術的にはベクトルですが，簡単にするために，技術的に正しい表現である「x上の偏導関数」の代わりに「x上の勾配」などの用語を使うことがあります．また，足し算の導関数を導くこともできます．

$$f(x,y)=x+y \quad \rightarrow \quad \frac{\partial f}{\partial x}=1 \quad \frac{\partial f}{\partial y}=1$$

つまり，x,yの両方での微分は，x,yの値に関わらず1となります．これは，x,yのいずれかを増加させることでfの出力が増加し，その増加率はx,yの実際の値に関係しないので，理にかなっています(上記の乗算の場合とは異なります)．
この授業でよく使う最後の関数は，max演算です．

$$f(x,y)=\max(x,y) \quad \rightarrow \quad \frac{\partial f}{\partial x}=\mathbb{1}(x>=y) \quad \frac{\partial f}{\partial y}=\mathbb{1}(y>=x)$$

つまり，(劣)勾配は大きくなった方の入力では1，もう一方の入力では0となります．直感的には，入力が$x=4, y=2$であれば，最大値は4であり，この関数はyの設定には敏感ではありません．つまり，もしわずかにhを増加させたとしても，関数は4を出力し続け，したがって勾配はゼロになります．もちろん，yを大きく(例えば2よりも大きく)変化させれば，fの値も変化しますが，導関数は，そのような大きな変化が関数の入力に与える影響については何も教えてくれません．微分は，$\lim_{h\to 0}$と定義されていることからもわかるように入力に対する極小，無限小の変化に対してのみ情報を提供します．

連鎖律による複合式

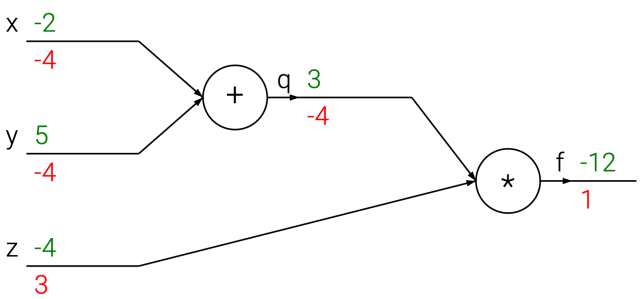
ここからは，$f(x,y,z)=(x+y)z$のような複数の関数を含む，より複雑な式を考えてみましょう．この式は，直接微分できるほど単純なものですが，バックプロパゲーションの直感を理解するのに役立つ特別なアプローチをとってみましょう．特に，この式が次の2式に分解できることに注意してください．$q=x+y$，$f=qz$．さらに，前節で見たように，両方の式の微分を別々に計算する方法もわかっています．fはqとzの乗算だけなので$\frac{\partial f}{\partial q}=z, \frac{\partial f}{\partial z}=q$，$q$はxとyの加算なので$\frac{\partial q}{\partial x}=1, \frac{\partial q}{\partial y}=1$となります．しかし，中間値$q$での勾配は必ずしも配慮することはない…$\frac{\partial f}{\partial q}$の値は役に立たないからです．その代わり，最終的に注目したいのはfのx,y,zに対する勾配です．連鎖律では，これらの勾配式を「連鎖」させるには，乗算を行うのが正しい方法です．例えば，$\frac{\partial f}{\partial x}=\frac{\partial f}{\partial q}\frac{\partial q}{\partial x}$です．
実用的には，2つの勾配を持つ2つの数値を掛け合わせるだけです．例を見てみましょう．

```
# set some inputs
x = -2; y = 5; z = -4

# perform the forward pass
q = x + y # q becomes 3
f = q * z # f becomes -12

# perform the backward pass (backpropagation) in reverse order:
# first backprop through f = q * z
dfdz = q # df/dz = q, so gradient on z becomes 3
dfdq = z # df/dq = z, so gradient on q becomes -4
# now backprop through q = x + y
dfdx = 1.0 * dfdq # dq/dx = 1. And the multiplication here is the chain rule!
dfdy = 1.0 * dfdq # dq/dy = 1
```

変数`[dfdx,dfdy,dfdz]`の勾配が残り，変数`x,y,z`の`f`に対する感度を知ることができます．これがバックプロパゲーションの最も単純な例です．今後は，`df`という接頭辞を省いた，より簡潔な表記法を使うことにします．例えば，`dfdq`ではなく`dq`と書き，常に最終出力で勾配が計算されていると仮定します．
この計算は，神経回路図を使ってきれいに視覚化することもできます．



左側の実数値の「回路」は，計算を視覚的に表したものです．前進パスでは，入力から出力までの値を計算します(緑色で表示)．後方パスでは，入力から出力に至るまでの勾配(赤で表示)を計算するために，最後から始まって連鎖法則を再帰的に適用するバックプロパゲーションを実行します．勾配は，回路を逆に流れると考えることができます．

バックプロパゲーションの直観的な理解

バックプロパゲーションはとても局所的なプロセスであることに注目してください．回路図のすべてのゲートは，いくつかの入力を得て，すぐに2つのことを計算することができます．1.その出力値，2.入力に対する出力の局所的な勾配です．ゲートは，自身が組み込まれている回路全体の詳細を意識することなく，完全に独立してこの処理を行うことができることに注目してください．しかし，フォワードパスが終わると，バックプロパゲーションの際に，ゲートは最終的に回路全体の最終出力に対する出力値の勾配を知ることになる．チェーンルールでは，ゲートはその勾配を，通常，すべての入力に対して計算するすべての勾配に乗じるべきだとしている．

☞この連鎖法則による余分な乗算(各入力に対して)は，単一の比較的役に立たないゲートを，ニューラルネットワーク全体のような複雑な回路の「歯車」に変えてしまいます．

これがどのように機能するか，もう一度例を見て直感的に理解してみましょう．加算ゲートは入力[-2, 5]を受け取り，出力3を計算しました．このゲートは加算演算を行っているため，両方の入力に対する局所勾配は+1となります．回路の残りの部分では，最終的な値として-12が計算されます．チェーンルールが回路を再帰的に遡って適用されるバックワードパスの間に，(乗算ゲートの入力である)加算ゲートは，その出力の勾配が-4であったことを学習します．回路を「より高い値を出力したい」と擬人化すると(これは直感を助けることになる)，回路は加算ゲートの出力を(負の符号のために)より低く，力を4にすることを「望んでいる」と考えることができる． 再帰を続けて勾配を連鎖させるために，加算ゲートはその勾配を取り，入力の局所勾配すべてに乗算する(xとyの両方の勾配を1 * -4 = -4にする)
これが望ましい効果であることに注目してください：x, yが減少すると(負の勾配に反応して)，加算ゲートの出力が減少し，その結果，乗算ゲートの出力が増加します．つまりバックプロパゲーションとは，最終的な出力値を高くするために，ゲートがお互いに(勾配信号を通して)出力の増減(およびその強さ)を伝え合っていると考えることができます．

モジュラリティ(分割することによる効果)：シグモイドの例

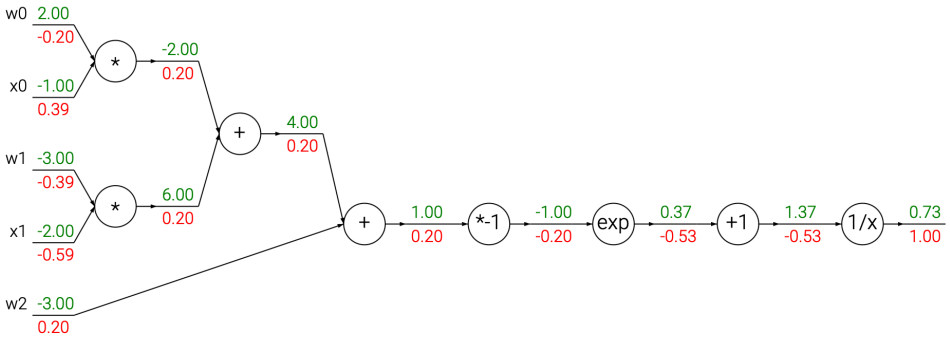
先に紹介したゲートは比較的自由に設定することができます．微分可能な関数であればどのようなものでもゲートとして働くことができますし，複数のゲートを1つのゲートにまとめたり，関数を複数のゲートに分解したりすることも都合の良いときにできます．この点を説明するために，別の式を見てみましょう．

$$f(w,x) = \frac{1}{1+e^{-(w_0x_0+w_1x_1+w_2)}}$$

後述するように，この式はシグモイド活性化関数を使用する2次元のニューロン(入力xと重みwを持つ)を表しています．しかし今は，入力w, xから1つの数値への関数として，非常にシンプルに考えることができます．この関数は複数のゲートで構成されています．上で説明したゲート(add, mul, max)に加えて，4つのゲートがあります．

$$
\begin{aligned}
f(x) = \frac{1}{x} \quad &\rightarrow \quad \frac{df}{dx} = -1/x^2 \\
f_c(x) = c + x \quad &\rightarrow \quad \frac{df}{dx} = 1 \\
f(x) = e^x \quad &\rightarrow \quad \frac{df}{dx} = e^x \\
f_a(x) = ax \quad &\rightarrow \quad \frac{df}{dx} = a
\end{aligned}
$$

ここで，関数f_c, f_aは，それぞれ入力を定数cで変換し，定数aでスケーリングします．これらは技術的には加算と乗算の特殊なケースですが，定数c, aの勾配が不要なので，ここでは(新しい)単項ゲートとして導入します．回路全体の構成は次のようになります．



シグモイド活性化関数を持つ2次元ニューロンの回路例．入力は[x0,x1]，ニューロンの(学習可能な)重みは[w0,w1,w2]です．後で説明しますが，ニューロンは入力とのドット積を計算し，その活性化をシグモイド関数でソフト的に潰して，0から1の範囲にします．

上の例では，w, x間のドットプロダクトの結果を操作する関数アプリケーションの長いチェーンが見られます．この関数はシグモイド関数$\sigma(x)$と呼ばれています．入力に対するシグモイド関数の導関数を実行すると，(分子に1を足したり引いたりする色々トリッキーな部分を経て)単純化されることが判明しました．

$$
\begin{aligned}
\sigma(x) &= \frac{1}{1+e^{-x}} \\
\rightarrow \frac{d\sigma(x)}{dx} &= \frac{e^{-x}}{(1+e^{-x})^2} = \left(\frac{1+e^{-x}-1}{1+e^{-x}}\right)\left(\frac{1}{1+e^{-x}}\right) = (1-\sigma(x))\sigma(x)
\end{aligned}
$$

ご覧のように，勾配は単純化され，驚くほどシンプルになることがわかりました．例えば，シグモイド式は入力1.0を受け取り，フォワードパスで出力0.73を計算します．上の導出では，局所的な勾配は単純に(1 - 0.73) * 0.73 ~= 0.2となり，回路が以前に計算したようになります(上の画像を参照)．ただし，この方法では単一のシンプルで効率的な式で行われます(数値的な問題も少ない)．したがって，実際のアプリケーションでは，これらの演算を1つのゲートにまとめることが非常に有効です．では，このニューロンのバックプロップをコードで見てみましょう．

(次ページに続きます)

```
w = [2,-3,-3] # assume some random weights and data
x = [-1, -2]

# forward pass
dot = w[0]*x[0] + w[1]*x[1] + w[2]
f = 1.0 / (1 + math.exp(-dot)) # sigmoid function

# backward pass through the neuron (backpropagation)
ddot = (1 - f) * f # gradient on dot variable, using the sigmoid gradient
derivation
dx = [w[0] * ddot, w[1] * ddot] # backprop into x
dw = [x[0] * ddot, x[1] * ddot, 1.0 * ddot] # backprop into w
# we're done! we have the gradients on the inputs to the circuit
```

実装上の注意：段階的なバックプロパゲーション．上のコードに示されているように，実際にはフォワードパスを簡単にバックプロップできる段階に分割することが常に有用です．例えばここでは，`w`と`x`の間のドット積の出力を保持する中間変数`dot`を作成しました．バックワードパスの間に，これらの変数の勾配を保持する対応する変数(例えば`ddot`，最終的には`dw,dx`)を(逆順に)連続して計算します．

このセクションのポイントは，バックプロパゲーションがどのように実行されるかの詳細や，フォワード関数のどの部分をゲートと考えるかは，利便性の問題であるということです．式のどの部分が簡単に局所的な勾配を持つかを認識しておくと，最小限のコードと労力で連鎖させることができるようになります．

バックプロップの実践：段階計算

別の例で見てみましょう．次のような形式の関数があるとします．

$$f(x,y) = \frac{x + \sigma(y)}{\sigma(x) + (x+y)^2}$$

はっきり言って，この関数は全く役に立たず，なぜその勾配を計算したいのかもわからないのですが，バックプロパゲーションの実践的な良い例であることは事実です．重要なのは，もしxまたはyのどちらかに対して微分を実行し始めると，非常に大きくて複雑な式になってしまいます．しかし，そのようなことをする必要は全くありません．なぜなら，勾配を評価する明示的な関数を書く必要はないからです．勾配を計算する方法さえ知っていればいいのです．このような式のフォワードパスをどのように構成するかを示します．

(次ページに続きます)

```
x = 3 # example values
y = -4

# forward pass
sigy = 1.0 / (1 + math.exp(-y)) # sigmoid in numerator   #(1)
num = x + sigy # numerator                               #(2)
sigx = 1.0 / (1 + math.exp(-x)) # sigmoid in denominator #(3)
xpy = x + y                                              #(4)
xpysqr = xpy**2                                          #(5)
den = sigx + xpysqr # denominator                        #(6)
invden = 1.0 / den                                       #(7)
f = num * invden # done!                                 #(8)
```

フーッ．この式の終わりまでに，フォワードパスを計算しました．複数の中間変数を含むようにコードを構成していることに注意してください．それぞれの中間変数は，すでに局所勾配がわかっている単純な式にすぎません．したがって，バックプロップパスの計算は簡単です．フォワードパスの途中にあるすべての変数(`sigy, num, sigx, xpy, xpysqr, den, invden`)に対して，同じ変数(`d`で始まる変数)を用意し，その変数に対する回路の出力の勾配を保持します．さらに，バックプロップのすべてのピースは，その式の局所勾配を計算し，その式の勾配と乗算を連鎖させることに注意してください．また，各行について，フォワードパスのどの部分を参照しているかを強調しています．

```
# backprop f = num * invden
dnum = invden # gradient on numerator                          #(8)
dinvden = num                                                  #(8)
# backprop invden = 1.0 / den
dden = (-1.0 / (den**2)) * dinvden                             #(7)
# backprop den = sigx + xpysqr
dsigx = (1) * dden                                             #(6)
dxpysqr = (1) * dden                                           #(6)
# backprop xpysqr = xpy**2
dxpy = (2 * xpy) * dxpysqr                                     #(5)
# backprop xpy = x + y
dx = (1) * dxpy                                                #(4)
dy = (1) * dxpy                                                #(4)
# backprop sigx = 1.0 / (1 + math.exp(-x))
dx += ((1 - sigx) * sigx) * dsigx # Notice += !! See notes below  #(3)
# backprop num = x + sigy
dx += (1) * dnum                                               #(2)
dsigy = (1) * dnum                                             #(2)
# backprop sigy = 1.0 / (1 + math.exp(-y))
dy += ((1 - sigy) * sigy) * dsigy                              #(1)
# done! phew
```

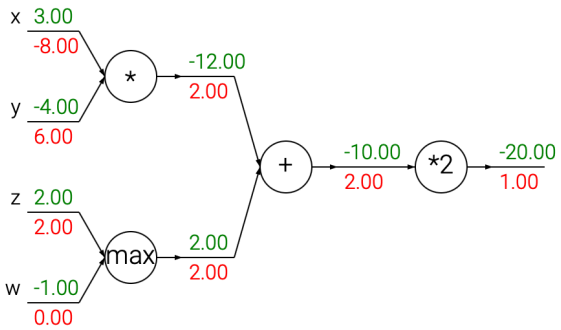
注意点:

フォワードパスの変数をキャッシュする．バックワードパスを計算する際に，フォワードパスで使用された変数の一部があると非常に便利です．実際には，これらの変数をキャッシュし，バックプロパゲーション中に利用できるようにコードを構成する必要があります．それが難しい場合は，再計算することも可能です(ただし，無駄になります)．

勾配変数は分岐で加算されます．順方向の式は変数x,yを複数回含むので，バックプロパゲーションを行う際には，これらの変数の勾配変数を蓄積するために，=ではなく+=を使用するように注意しなければなりません(そうしないと上書きしてしまいます)．これは微積分学の多変数連鎖法則に従ったもので，ある変数が回路の異なる部分に分岐している場合，その変数に戻ってくる勾配は加算されるというものです．

バックワードフローのパターン

興味深いのは，多くの場合，バックワードフローの勾配は直感的に解釈できるということです．例えば，ニューラルネットワークで最もよく使われる3つのゲート（add,mul,max）は，バックプロパゲーションの際にどのように作用するかという点で，非常にシンプルな解釈ができます．次のような回路を考えてみましょう．



バックプロパゲーションが入力の勾配を計算するのにバックワードパスで実行する演算の背後にある神経回路の例．Sum演算は，勾配をすべての入力に均等に分配します．Max演算では，勾配を高い方の入力に分配します．Multiplyゲートは，入力の活性化関数を受け取り，それらを交換し，その勾配を乗算する．

ひとつの例として上図を見ると，以下のことが解ります．
加算ゲートは，フォワードパスでの値に関わらず，常に出力の勾配を取り，それをすべての入力に均等に分配します．これは，足し算の局所的な勾配が単純に+1.0であることから，すべての入力の勾配は出力の勾配と完全に等しくなり，出力はx1.0倍される（変化しない）からです．上の回路例では，+ゲートが2.00の勾配をその両方の入力に等しく，かつ変化しないようにルーティングしていることに注目してください．
maxゲートは勾配をルーティングします．勾配をすべての入力に変更せずに分配したaddゲートとは異なり，maxゲートは勾配を（変更せずに）正確に1つの入力（フォワードパス中に最も高い値を持っていた入力）に分配します．これは，maxゲートの局所的な勾配が，最も高い値では1.0，それ以外の値では0.0であるためです．上の回路例では，max演算によって2.00の勾配がwよりも高い値を持つz変数にルーティングされ，wの勾配は0のままとなっています．
乗算ゲートは少し解釈が難しくなります．その局所勾配は入力値（スイッチを除く）であり，これにチェーンルール中の出力の勾配が掛けられます．上の例では，xの勾配は-8.00で，-4.00×2.00となります．
直感的でない効果とその結果．乗算ゲートへの入力の一方が非常に小さく，他方が非常に大きい場合，乗算ゲートは少し直感的でない動作をすることに注意してください．重みが入力とドット積$w^T x_i$（乗算）される線形分類器では，データの大きさが重みの勾配の大きさに影響します．

例えば，すべての入力データ例x_iを掛け合わせた場合x_iを1000倍にしたとすると，重みの勾配は1000倍になり，その分だけ学習率を下げなければなりません．このように，前処理は非常に重要であり，時には微妙な方法で行われます．そして，勾配の流れを直感的に理解することは，このようなケースのデバッグに役立ちます．

ベクトル演算時の勾配

上記のセクションでは，単一の変数について説明しましたが，すべての概念は，行列やベクトルの演算に簡単に拡張できます．ただし，次元と転置演算には細心の注意を払う必要があります．

行列同士の乗算による勾配．おそらく最も厄介な演算は，行列-行列乗算（一般的な行列-ベクトルおよびベクトル-ベクトル）です．

```
# forward pass
W = np.random.randn(5, 10)
X = np.random.randn(10, 3)
D = W.dot(X)

# now suppose we had the gradient on D from above in the circuit
dD = np.random.randn(*D.shape) # same shape as D
dW = dD.dot(X.T) #.T gives the transpose of the matrix
dX = W.T.dot(dD)
```

ヒント：次元を考慮しましょう．`dW`と`dX`の式は，次元に基づいて簡単に再導出できるので，覚えておく必要はないことに注意してください．例えば，重みの勾配`dW`は計算後の`W`と同じ大きさでなければならず，`X`と`dD`の行列乗算に依存することがわかっています（`X`,`W`の両方が行列ではなく単一の数値である場合がそうです）．正しい仕方でこの次元で実装する方法は一つです．
例えば，`X`はサイズが[10 x 3]で，`dD`はサイズが[5 x 3]なので，`dW`を求め，`W`が[5 x 10]の形をしている場合，これを実現する唯一の方法は，上に示したように，`dD.dot(X.T)`です．

小さくて明確な例を使う．ベクトル化された式の中には，勾配の更新を導き出すのが最初は難しいと感じる人もいるでしょう．私たちがお勧めするのは，最小限のベクトル化された例を明示的に書き出し，紙の上で勾配を導き出し，そのパターンを効率的なベクトル化された形に一般化することです．
Erik Learned-Millerは，行列/ベクトルの導関数の取り方について，より長い関連文書を書いています．それはこちらです．

まとめ

・勾配が何を意味するのか，回路内をどのように逆流するのか，最終的な出力を高くするために回路のどの部分をどのような力で増減させればよいのかをどのように伝えるのか，直感的に理解することができました．

・バックプロパゲーションの実用的な実装には，**段階的な計算**が重要であることを説明しました．局所的な勾配を簡単に導けるようなモジュールに関数を分割し，それらを連鎖律で連鎖させたいと常に考えています．重要なことは，これらの式を紙に書き出し，記号的に微分することはほとんどないということです．なぜなら，入力変数の勾配を表す明示的な数学的方程式は必要ないからです．したがって，各ステージを独立して微分できるように式をステージに分解し（ステージは行列ベクトルの乗算や最大値の演算，和の演算などになります），変数を1ステップずつバックプロップしていきます．

次のセクションでは，ニューラルネットワークの定義に入りますが，バックプロパゲーションによって，損失関数のパラメータに対する勾配を効率的に計算することができます．これでニューラルネットを学習する準備が整い，この授業で最も概念的に難しい部分が終わったことになります．ConvNetsもあと少しで完成です．