多層パーセプトロンの原理

すうがくぶんか

　　　　　　　　　　　　　　　　【目次】

[１　単純パーセプトロン 3](#_Toc466998105)

[２　単純パーセプトロンにおける誤差関数 7](#_Toc466998106)

[３．XOR問題：単純パーセプトロンの限界 13](#_Toc466998107)

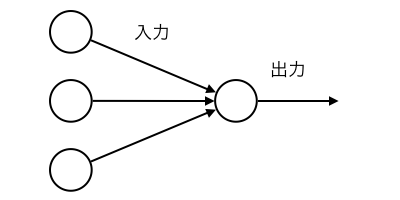
[４　活性化関数 20](#_Toc466998108)

[５　誤差逆伝播法 27](#_Toc466998109)

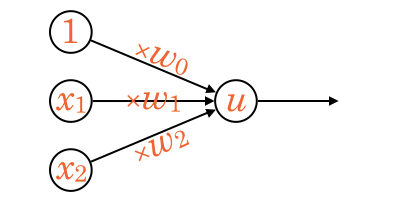
[６　デルタの定義 38](#_Toc466998110)

# １　単純パーセプトロン

動物の神経細胞（ニューロン）は、樹状突起という部位で他の細胞から複数の入力を受け取り、入力が一定値以上に達すると信号を出力する（これを「発火する」と言ったりします）とされており、それをモデル化したものとして**形式ニューロン**というものが提案され、さらに応用して**パーセプトロン**というモデルが発明されました。[1](http://hokuts.com/2015/11/25/ml2_perceptron/#fn-442-def)  
と、難しそうなことを書きましたが、ざっくり絵にするとこんな感じです。



絵では入力が3つになっていますが、実際にはいくつでも構いません。  
前回の の問題は、このモデルにあてはめることができます。

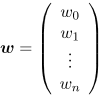


入力ノードが 、入力途中の矢印が に対応しています。  
を入力として受け取り、それぞれに を掛けた後、中心のノードですべて足し合わせます。  
この値を とします。



前回のSPAMの例だと、正しく学習された後であれば、SPAMの場合 、非SPAMの場合 となるはずですね。  
このようなモデルで表現できる学習機械を**単純パーセプトロン**と呼びます。  
これがいわゆる**ニューラルネットワーク**の基本単位となります。

**重みベクトルの更新**

こうして見ると、SPAMフィルタにおいて学習後の は相対的に各単語の「SPAMっぽさ」を表す値になっていることがわかるでしょうか。  
はメール中の注目した各単語の数でした。  
例えば と が同じ値（単語数）だったとしても、 より の絶対値の方が大きければ、 が に与える影響は大きくなります。 つまり、一般的に は各入力値 の**重み**を表していると言えます。[2](http://hokuts.com/2015/11/25/ml2_perceptron/#fn-442-weight)  
は入力と結びついていないバイアスと呼ばれる値なのですが、ここではこれもまとめて重みと考えてしまいましょう。  
というわけで、ベクトル を、**重みベクトル**と呼びます。

をパーセプトロンに入力したときに出力される値を教師データと見比べながら、 を更新していきます。  
では、実際にはどのようにを更新していけばいいのでしょうか？  
簡単に言うと、

1. にそれぞれランダムな値を設定する
2. 下記を繰り返す
   * 教師データを読み込ませ、出力が正しくなければ、重みの値を「正しそうな方向」に少しずらす
   * 全教師データについて正しい出力が行われたら繰り返し終了

という手順なのですが、単純パーセプトロンの場合、繰り返しの中で重みを「正しそうな方向」に更新する式は非常に簡単です。  
ただしここでは、その式を記載する前に、次回以降に学ぶ多層パーセプトロンでも活用できるよう、更新式を導き出す一般的な考え方についてまずは説明します。

**勾配降下法**

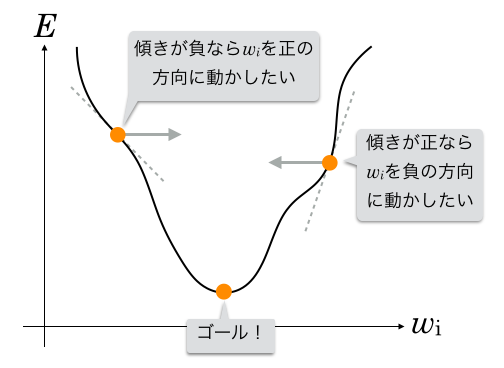
まず、「ある教師データを読み込ませたときの出力がどれくらい期待外れだったか？」を数値で返す関数 があるとします。  
これを**誤差関数**、あるいは**損失関数**と言ったりします。



などを含む式で表現され、0以上の値を返すようにするのが普通です。  
この関数は「出力が期待外れであるほど大きな値を返す」ものであれば何でも良いといえば良いのですが、定義次第で後の計算での扱いやすさが変わってきます。  
実際の定義は次項で行いますね。

誤差関数の出力値は「期待外れの度合い」ですから、それを最小にできれば学習完了したと見なせます。  
をランダムな値で初期化してから少しずつ動かしていき、誤差関数の出力値を最小にすることを考えます。

の要素の1つ に注目し、 と誤差関数の出力値の関係をグラフにすると下記のようになったとします。（ 以外の変数は固定されていると考えます。）



このグラフで「誤差が一番小さくなる」は曲線が下方向に突起している箇所なので、目指したいのはここです。[3](http://hokuts.com/2015/11/25/ml2_perceptron/#fn-442-minimum)  
グラフ中にも記載がありますが、

* 現在のでグラフの傾きが正の場合 →を負の方向に動かす
* 現在のでグラフの傾きが負の場合 →を正の方向に動かす

とすれば、誤差が一番小さくなるに辿り着くことができそうです。  
傾きとは、微分値 です。  
そこで、下記の更新式を考えてみます。[4](http://hokuts.com/2015/11/25/ml2_perceptron/#fn-442-gets)

**更新式（仮）**



微分値、つまり傾きが正の時はが負の方向に動き、負の時はが正の方向に動くようになりました。  
また、傾きの絶対値が大きいときは大きく変化し、小さいときは小さく変化します。  
ただ、これだと更新量が大きすぎてうまく収束しないので、小さな正の定数を決めて、微分値に掛けることで更新量を調整します。

**更新式（決定版）**



をベクトルとしてまとめて書いてしまうとこうです。

**更新式（決定版・改）**



このを**学習率**といいます。  
1より小さな値を設定するのが普通ですが、あまりにも小さいと更新量も小さくなり計算の回数が増えてしまうのでご注意ください。[5](http://hokuts.com/2015/11/25/ml2_perceptron/#fn-442-learningrate)

こうして誤差が最小になったと思われるまで繰り返しを変化させていくのです。  
この方式を**勾配降下法**といいます。[6](http://hokuts.com/2015/11/25/ml2_perceptron/#fn-442-decend)

この更新式からわかるように、の定義はで微分したときに計算が簡単な式になる形が望ましいです。

あとは、誤差関数が決まれば実際の更新式も決まりますね。

# ２　単純パーセプトロンにおける誤差関数

単純パーセプトロンにおいて、1組のパラメータに対する誤差関数は下記で定義できます。



は、 と のうち大きい方を出力する関数です。  
は教師データの正解ラベルで、次のように定義します。

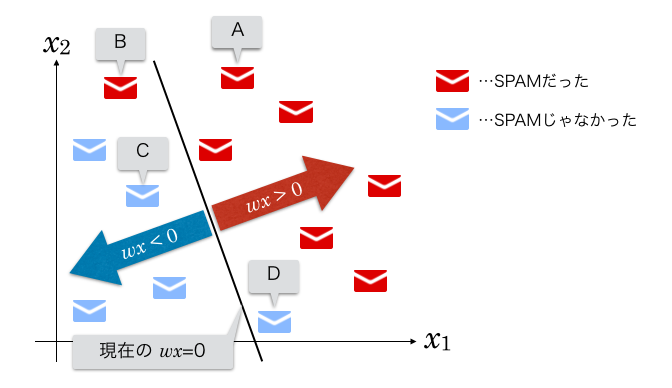


真偽値っぽく 0, 1 にしたい人もいるかも知れませんが、 -1, 1 なのが重要です。

なぜ が誤差関数として使えるのでしょうか？

の2次元で考えてみます。  
まず、繰り返しになりますが、（つまり ）の値は、 の直線上では当然 0、直線で分割された片方のエリアでは正の値、もう一方では負の値になります。 そして、ラベルが1の点はすべて正のエリアに、ラベルが-1の点はすべて負のエリアに入ると学習完了です。

下図をご覧ください。



これは学習が完了していない状態のグラフで、BとDの点が誤分類されている様子を表しています。  
まずAの点に注目してみます。  
このメールは SPAM ですが、現在の識別関数で正しく分類されており、 です。  
正解ラベル は 1 ですから、  
です。よって  
となり、誤差が0という結果になります。

同じように、A〜D各点について表にまとめてみます。  
（「正解？」というのは、「現状で正しく分類されているか？」という意味で書いてます。）



この誤差関数は、ある点 が正しく分類されていれば 0 を返し、正しく分類されていなければ を返すことがわかると思います。[7](http://hokuts.com/2015/11/25/ml2_perceptron/#fn-442-abs)

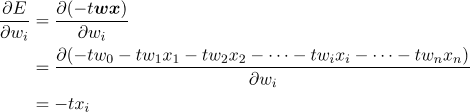
では、この というのは何を表しているのでしょうか？  
実は、この値はグラフ上の点から直線への距離に比例しているのです。[8](http://hokuts.com/2015/11/25/ml2_perceptron/#fn-442-distance)  
感覚的にも、 の直線に近い点ほど の値も 0 に近いというのは腑に落ちやすいのではないでしょうか。  
この誤差関数は、ある点が期待と逆側に分類されてしまっているときに、直線からの距離が大きければ誤差も大きいと見なし、距離が小さければ誤差も小さいと見なすということを意味します。

**単純パーセプトロンの学習アルゴリズム**

誤差関数が定まったので、重みの更新式を作れるようになりました。  
一般に、重みの更新式は下記でした。



は、誤差がない場合は 0 を、誤差がある場合は を返します。  
誤差がない場合を無視し、 の1要素 に注目すると、



なので、更新式は下記です。



ベクトルでまとめて表現するとこうです。

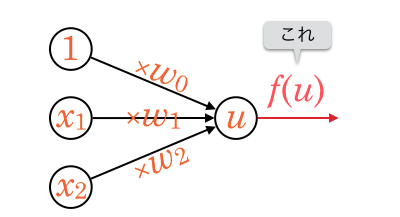


以上を踏まえると、単純パーセプトロンの学習アルゴリズムは下記のようになります。

* をランダムで設定
* each 全教師データ
  + 教師データのを入力し、出力が正解と合っているかチェック
    - 合っていた場合、何もしない
    - 合っていなかった場合、 
* 上記のループ内での更新があったか？
  + あった場合、もう一度上記のループを最初から
  + なかった（すべて正解だった）場合、終了

**活性化関数**

単純パーセプトロンにおいてはあまり意識しなくても実装できるのですが、パーセプトロンのユニットは、 の値に応じて何らかの値を出力します。



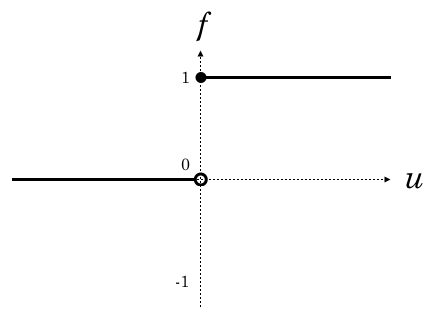
この出力値を決定する関数を活性化関数といいます。  
問題に応じて適切な関数を使用します。  
最もシンプルな活性化関数は、 をそのまま出力する関数です。



今回のSPAM識別のような二値分類では、単位ステップ関数という下記がシンプルです。



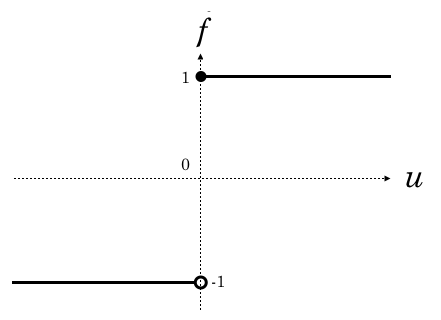
グラフにするとこんな感じです。



ただ、正解ラベルは1と-1だったので、それに合わせてこういう関数にした方がわかりやすい実装になるかも知れません。[9](http://hokuts.com/2015/11/25/ml2_perceptron/#fn-442-gte)



グラフにするとこうなりますね。



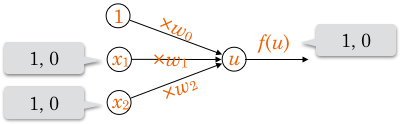
これら2つのグラフの形は、「入力がある値を超えたら急激に発火する」という動物の神経細胞の動きに似ています。  
次回以降で学ぶ多層パーセプトロンでは、この活性化関数をどのように選ぶかが重要になってきます。

1. 両者の違いを簡単に述べると、形式ニューロンは入出力が0または1のみであるというのと、バイアス（この記事でとしているもの）の扱いが少し違います。また、「パーセプトロン」に関しては、1958年に発表されたそうですが、その後様々な応用が提案される中で厳密な定義は曖昧になってきている印象です。あまりこだわらないのがよさそうです。
2. 例えばSPAM識別において「出会い」という単語の重みはとても大きくなるでしょう。  
   SPAMか否かに影響を与えない単語の重みは0に近くなるはずです（その単語はパラメータとして無意味ということでもあります）。  
   また、「この単語があればSPAMの可能性はむしろ減る…」というような単語がもし存在するなら、その重みは負の値になるかも知れません。
3. 厳密には勾配降下法で辿り着けるのは極小値なので、必ずしもそれが最小値になるとは限りません。ただし、単純パーセプトロンで線形分離可能な問題を解く場合はそこの心配は要りません。より複雑な問題では、極小値が複数ある場合を考慮する必要が出てきたりもします。
4. 左向きの矢印は代入を表します。
5. 実際の問題ではうまく収束させるために学習率を動的に変化させたりします。  
   また、ここで挙げた更新式が基本ではありますが、より確実に最小値を目指すためや、計算量を減らすためなどの、様々な改善テクニックがあります。

1. 勾配降下法を適用するアルゴリズムによって、さらに下記のように呼び分けられます。  
   **最急降下法**  
   一般に勾配降下法によって関数の極小値を目指す手法を言いますが、機械学習においては、まず全教師データを読み込ませ、各々の誤差の合計値（または平均値）を算出してから、それを最小化していく手法を指すようです。  
   このような学習法を一括（バッチ）学習と呼びます。  
   **確率的勾配降下法**  
   一部の教師データのみ（典型的には1件のみ）を読み込ませ、その誤差をもとに1回だけ更新を行い、次の教師データを読み込ませ…ということを繰り返す手法です。  
   逐次（オンライン）学習またはミニバッチ学習に分類される学習法です。
2. は の絶対値を表します。
3. 点 から直線 への距離は  
   で表せます。  
   この公式の証明は検索するといろいろ出てきます。
4. は ≧と同じです。

# ３．XOR問題：単純パーセプトロンの限界

単純パーセプトロンの学習によって論理演算（ANDやORなど）の役割を果たす識別器を作ることを考えます。  
真 = 1, 偽 = 0 とおき、, それぞれでいずれかを入力します。  
パーセプトロンを通過した結果、それらを論理演算した値（0または1）が出力されるようにします。



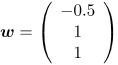
0か1の出力なので、活性化関数 として単位ステップ関数を利用します。

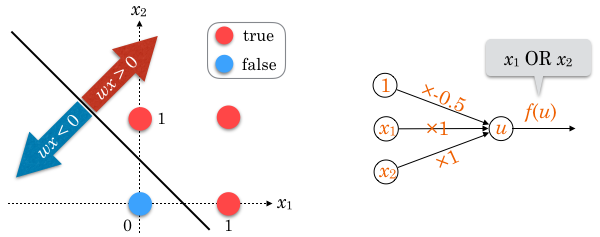


単純パーセプトロンにおける重みベクトルの学習手順は前回説明したとおりですので、以下、学習によって具体的にどのような重みになってほしいか？という観点で話を進めます。  
以下のようなアプローチで考察ができるのは2次元（入力が2つ）ならではです。

**OR（論理和）**

まず、ORの役割を果たす単純パーセプトロンについて考えます。  

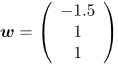

入力値 をグラフにプロットし、出力値の境界に適当な直線を引いてみると、例えば傾きが-1で切片が0.5、つまり の直線で分割できることがわかります（下図参照）。[1](http://hokuts.com/2015/12/04/ml3-mlp/#fn-638-example)  
移項して の形に直すと、 です。  
左辺の , に図中の赤い点の値を代入すると 、青い点の値を代入すると となり、この が識別関数として機能していることがわかります。  
つまり、学習の結果 重みベクトルが例えば になれば、この単純パーセプトロンは OR の役割を果たします。  
なお、 : : の比率がだいたいそれに近ければ同じ出力をしたりします。

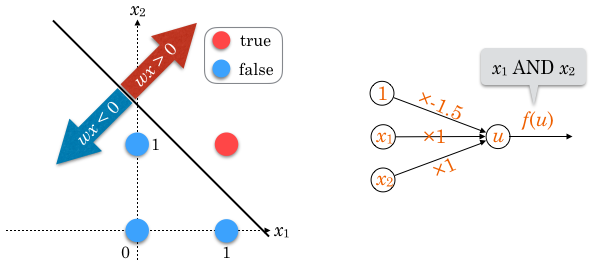


**AND（論理積）**

AND は下表です。



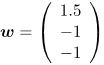
入力値 をグラフにプロットし、出力値の境界に直線を引いてみると、例えば で分割でき、これを OR と同じように整理すると です。

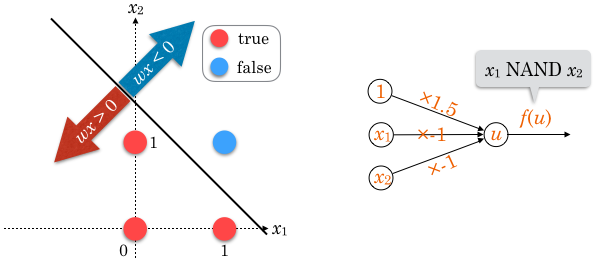


**NAND（否定論理積）**

NAND（つまり NOT AND）は下表です。



当然ですが、演算結果がANDと反転しています。  
AND と同じ直線で分離できるのですが、直線で分離された両エリアの不等号が逆になります。  
不等式の不等号を逆にするためには、両辺に -1 をかければいいですね。  
よって、AND の重みの正負を逆転させた が適切な重みの一例です。

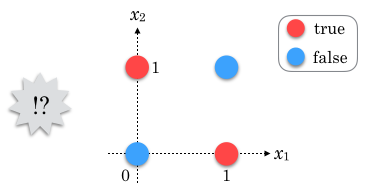


**XOR（排他的論理和）**

XORは下表です。

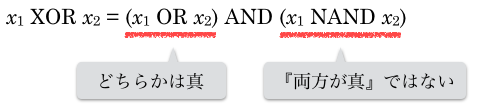


グラフを見るとわかるように、4点を直線で分割することができません。  
つまりこれは線形識別不可能な問題であり、単純パーセプトロンでは解決することができないのです。

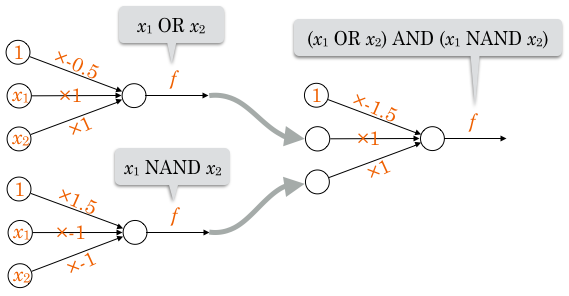


**多層パーセプトロン**

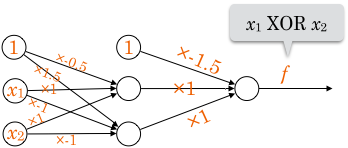
XORは、「,どちらかは真だが、『両方が真』ではない」時に真となります。  
つまりこのように表現できます。



OR、AND、NAND を使って表現できました。  
この3つの演算子は単純パーセプトロンで作成できるので、それらをつなぎ合わせることで結果的に XOR の役割を果たす識別器が作れます。



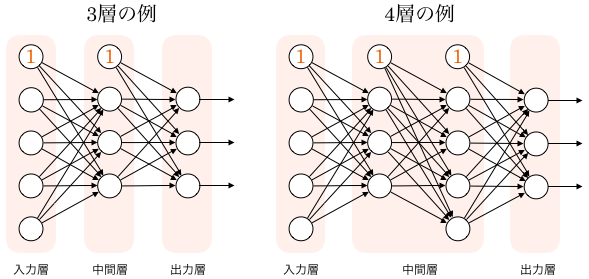
まとめた絵にするとこうです。



「あるユニットの出力を別のユニットの入力として使う」という構造を作ったことで、線形識別不可能な問題を解くことができました。[2](http://hokuts.com/2015/12/04/ml3-mlp/#fn-638-unseparable)  
これが**多層パーセプトロン**です。[3](http://hokuts.com/2015/12/04/ml3-mlp/#fn-638-neural)

この例では、真ん中の2ユニットの出力が右のユニットの入力として使われています。

一般に、つなぎ合わせるユニットの数は問題によって様々ですが、非常に多くのユニットを結合させて利用することも多いです。



このように、ユニットが層状になっていることから多層パーセプトロンと言います。  
上図で、一番左の層を**入力層**、一番右の層を**出力層**、それ以外の層を**中間層**または**隠れ層**と呼びます。  
この図では出力層にユニットが複数ありますね。その意義の一例は後の項で述べます。

XORの例は、「多層にすることによって、【重みの値を適切に学習できさえすれば】線形識別不可能な問題も解くことができる」ことを示すために挙げました。  
ただ、多層パーセプトロンの学習において、通常は単純パーセプトロンと同じ手順で重みを求めることはできません。  
XOR問題は非常に単純な問題なので「中間層のユニットがどのような値を出力してほしいか？」や「各重みがどういった値になってほしいか？」が明らかでしたが、一般にそういったことはほぼあり得ないからです。（教師データが与えるのは「入力層に入力するパラメータ」と「出力層から出力される値に対応する正解ラベル」のみです。中間層ユニットの各出力に対応する正解ラベルはありません。）  
というわけで、実際に重みを求める方法は次回解説します。

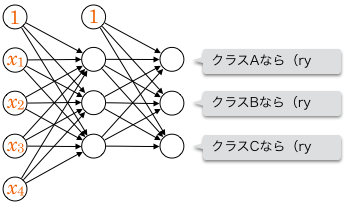
**多クラス分類**

ここまでの単純パーセプトロンの例では、SPAM識別や論理演算を扱いました。  
これらはパラメータ（グラフの各点）を2つに分類する行為（SPAM/非SPAM、真/偽 など）なので、2クラス分類や二値分類などと呼びます。  
これに対し、3つ以上に分類することを**多クラス分類**と呼んで区別したりします。

前回学んだ単純パーセプトロンで多クラス分類を行うには、分類すべきクラス数と同じ数だけ単純パーセプトロンを準備し、各入力に対してそのうち1つだけが発火（1を出力）するよう学習すればよいです。  
入力に対し、単純パーセプトロンAは「クラスAに属するか否か」を判定、単純パーセプトロンBは「クラスBに属するか否か」を判定…という具合です。  
4次元の入力をもとに3クラスに分類する例を図示するとこうなります。



ユニットの出力が別のユニットの入力になっていないという意味で、上記は多層パーセプトロンとは言えないことに注意してください（そのことがわかりやすいよう、上図では単純パーセプトロンの1つに色づけをしてあります）。  
単純パーセプトロンが複数あるだけなので、各クラスに属するか否かがそれぞれ線形識別可能な問題でなければ学習できません。  
一方、多層パーセプトロンで多クラス分類を行えば、線形識別不可能であっても分類できることがあります。



正解ラベルは、ベクトルを使って のように表現することができます。  
この場合は出力層の1つめのユニットだけが1を出力すれば正解という意味です。

このようにして、例えばメール中の各単語の数をもとに自動でメールをカテゴリ分類する仕組みなどが作れます。  
カテゴリ分けではなくタグ付けにするのであれば、出力層の複数ユニットが同時に1を出力することを許容してもいいですね。  
多層パーセプトロンによってどのような問題が解けるかを考えると楽しいと思います。

次回は、実際に多層パーセプトロンでどのように重みを更新していくのか、その手順を書きます。

1. 「例えば」と書いたのは、実際にはグラフ上の点を分割する直線の式は無数に存在するからです。傾きや位置が多少ずれても分割できたりしますし、同じ直線になるような重みも無限に考えられます。
2. なぜ線形分離不可能な問題が解けるかというと、「結果的に中間層で線形分離可能な形に変換されているから」と言うこともできます。
3. より広い意味の言葉として「ニューラルネットワーク」がありますが、多層パーセプトロンはその中でも  
   ・情報の伝達が一方向にのみ行われる  
   ・ユニットが層状に配置される  
   ・再帰的な接続を持たない  
   といった特徴を持ちます。

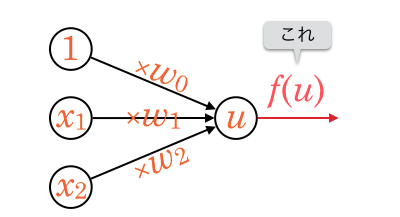
# ４　活性化関数

先にこの項のまとめを書いておくとこうです。

* 多層パーセプトロンの活性化関数は微分できることが重要
* 有効な活性化関数にはいろいろある
* ひとまずこの記事では活性化関数を 、その導関数を と一般化して記述する

それだけなのですが、せっかくなのでそのあたりの現状に軽く触れておこうと思います。

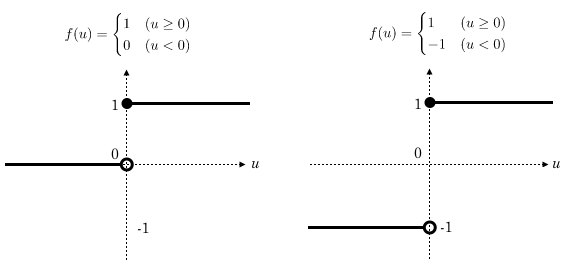
活性化関数とは、ユニットの出力値を決める関数のことでした。





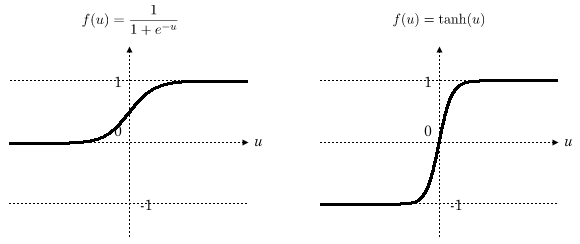


単純パーセプトロンの説明では、次のようなステップ関数を活性化関数として挙げました。



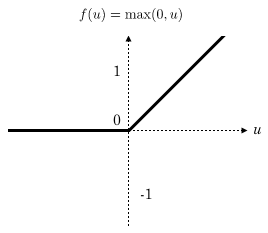
単純パーセプトロンの説明で、1つのユニットは （= ）が 0 より大きいか小さいかで入力を評価するものとしました。これらのステップ関数の定義はそれにマッチしています。  
そしてこれらは動物の神経細胞の発火状態と非発火状態を大幅に簡略化して表現したものとみなすことができます。

しかし実は、多層パーセプトロンではこの活性化関数が使えません。  
詳細は次回解説しますが、学習時に活性化関数の“微分値”が重要になるからです。  
ステップ関数だと有効な微分値が導けません。[1](http://hokuts.com/2016/05/27/pre-bp/#fn-985-nondifferentiable)  
そこで、多層パーセプトロンでは下記のような活性化関数がよく使われてきました。  
左はシグモイド関数、右は双曲線正接関数と呼ばれるものです。



式自体にはあまり意味がないと思っていいです。グラフの形に注目してみましょう。[2](http://hokuts.com/2016/05/27/pre-bp/#fn-985-sigmoid)  
どちらも  
・0を境に急激に変化している  
・出力値の範囲  
という点は前述のステップ関数と似ていますが、グラフが滑らかな曲線になっています。  
グラフが滑らかなのは微分可能な関数の特徴ですね。  
また、出力が0と1の二値ではなく連続値[3](http://hokuts.com/2016/05/27/pre-bp/#fn-985-continuous)をとることに注意してください。

さらに近年、中間層においては下記のような活性化関数を用いるとよいことが発見されました。  
ReLU（Rectified Linear Unit）と呼ばれるものです。[4](http://hokuts.com/2016/05/27/pre-bp/#fn-985-relu)



上記は数学的に言うと微分可能ではありませんが、プログラム上では下記のように微分値を定義することができます。



また、層内の出力値を全て足し合わせると1になるよう調整するソフトマックス関数や、層内の複数の出力値の中で最も大きい値を出力するMaxoutなど、同じ層にある他のユニットの影響も受けるような活性化関数もあります。

各々の解説は別記事に譲りますが、活性化関数についてはこのように様々な提案がされている状況ですので、今回の記事では活性化関数の具体的な定義をせず、 という形で統一して記述することにします。[5](http://hokuts.com/2016/05/27/pre-bp/#fn-985-various)  
また、の導関数を とします。

**各記号の再定義**

**各ユニットの重みと出力値**

多層パーセプトロンでは、大量のユニットと、それら各々に紐づく重みの値を扱う必要があります。  
これまで使ってきた などの、添え字が1つだけの記号では足りないので、ここで新たに記号を割り当てなおします。

新しい記号では添え字をたくさん使うので、これが式の中に現れると何だかとてつもなく難しく見えてしまいます。

ただし、添え字のせいで難しく見えるだけで、式自体はこれまでの知識で理解できるものなので、図を見ながら把握してみてください。

まず、入力層を除く層の数をとおき、入力層を第0層、その次を第1層…というように呼ぶことにします。出力層が第層ということになります。

そして、任意の第層のユニット数（固定で1を出力しているものを除く）を のように表記します。

第3層ならユニット 個です。

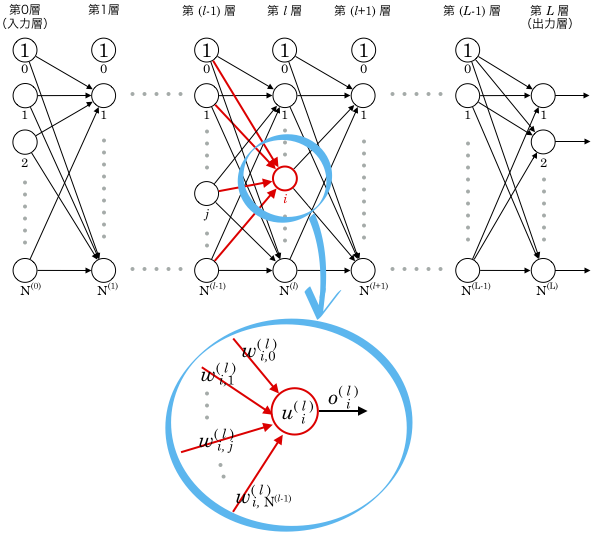
右上に数字がありますが累乗ではなくただの添え字です。累乗と区別するためにカッコをつけています。

次に、第層の番目のユニットに注目します。（下図参照）  
このユニットの出力値を とします。  
そしてこのユニットに紐付いている重みのうち、番目のものを と表現します。[6](http://hokuts.com/2016/05/27/pre-bp/#fn-985-subscript)

文字がたくさん組み合わさっていますが、1つの記号と見なしてください。添え字が3つあるだけです。

右下の2つの添え字の範囲は、1番目が 1〜、2番目が0〜となります。

このユニットへの入力は、1つ前の層の出力値なので、のようになります。  
これらの入力値に重みをかけて合計した値を とします。



添え字が多くて把握するのが大変ですが、せめて

* 右上の添え字は層番号
* 右下の添え字の1番目は層内でのユニット番号

というのは統一してみました。

**関係を式で表してみる**

1つのユニットは単純パーセプトロンと同じ動作をしますから、以上の記号の関係を式で表すと下記のようになります。





添え字が多すぎて書いてる方もイヤになってきました。最後の項なんかは添え字にも添え字がついています！

ただ、繰り返しますが、式自体の難しさはこれまでと変わりません。

上記の2行目の式を、今回だけ (A) 式と名付けます。  
この式の中に 層目の各ユニットの出力値が現れていますね。  
層目の 番目のユニットに注目すると、その出力値は、さらに 層目の出力値を使って下記のように表現できます。





これを (A) 式の に代入していくとかなり長い式になります。  
そして 層目の各出力値は、さらに層目の出力値を使った式で表現できます。

そうやって展開していくと、最終的に入力層のパラメータを使った式が導けることがわかるでしょうか。

しかしとてつもなく長い式になりそうですね。  
実際に式を導く必要はありませんが、そのように式展開できるということだけ理解しておいてください。

**和の記号を使う**

和の記号を使うと、(A)式はよりシンプルに書くことができます。



が添え字として使われている位置を見ると、その値の範囲は書かなくても明らかなので、省略して書いたりもします。

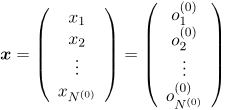


**ネットワークの入力と出力**

多層パーセプトロンは、ネットワーク全体で1つの学習機械です。  
このネットワーク（の入力層）にパラメータを入力し、（出力層から）出力されるのがより適切な値になる状態を目指すことになります。

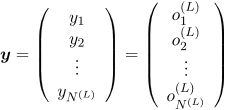
ネットワークへの入力パラメータをベクトルとします。

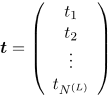
このベクトルの要素はそのまま入力層の各ユニットの出力値となりますから、下記のように表現できます。



ネットワークの出力値をベクトルとします。

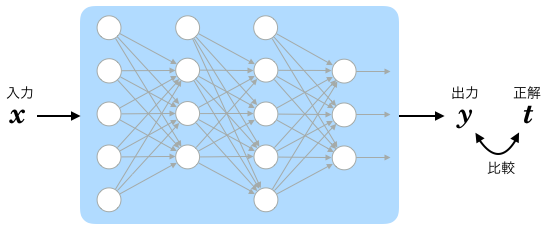
このベクトルの要素は出力層の各ユニットの出力値ですから、同じく下記のようになります。



出力層の出力値に対する正解ラベルは とします。

は、単純パーセプトロンの時のような 1 と -1 の二値ではなく、より一般的に「出力されてほしい理想の値」がそれぞれ入っていると考えます。

なお、中間層の出力値に対する正解ラベルは存在しないことに注意してください。



次回はいよいよ実際の学習法の理論に入ります。

1. の点で微分不可能というだけでなく、それ以外の点でも微分値が0になってしまいます。

1. グラフの特徴がよく見えるように、縦横の比率を意図的に操作しています。
2. コンピューター上では浮動小数などで近似されます。
3. シグモイド曲線を見ると、パラメータの絶対値がある程度大きくなると微分値が0に近くなってしまうのがわかります。このために重みの更新がなかなか進まないのが欠点なのですが、次回解説する誤差逆伝播法が理解できたら、なぜなかなか進まないのかを考えてみてください。ReLUが解決した問題はそこです。また、ReLUは微分値の計算コストも低いです。
4. 出力層と中間層で異なる活性化関数を用いるのが普通ですが、簡単のため、この記事ではそれをいったん無視します。

1. 当然と言えば当然ですが、記号の定義が資料によって違うのと同じく、添え字の位置関係も記述者によってまちまちですので注意してください。例えば の右下にはカンマ区切りで添え字が2つありますが、これの位置が逆転している資料も多いです。

# ５　誤差逆伝播法

**はじめに**

多層パーセプトロンの重みを更新する理論について解説します。  
更新すべき重みがたくさんあるので単純パーセプトロンより難しいですが、ここがわかると近年流行したディープラーニングを理解するための基本ができあがります。  
ただし、結構長いので、「理論はざっくりでいいから最終的に使える重み更新式が知りたい」という人は、別のサイトや本を読むのをお勧めします。

なお、記事中で使われる各種記号の定義は前回やりましたので、わからなくなったらそちらを参照してください。

**数学の前知識**

今回は数式がけっこう出てきます。  
そこで必要な数学知識について、いくつか説明を書いておきます。  
簡単のため前提条件を少し省略しているので、数学的に厳密ではない箇所はご容赦ください。

**偏微分**

偏微分は以前の記事でも出てきましたが、僕が誤解してまして、高校数学の範囲には含まれていないようでした。  
そんなに難しいことではなく、複数の変数による関数があるとして、その中の1つの変数のみに関する微分のことです。  
例えば として、 の に関する偏微分は です。  
以外の変数はすべて定数と見なすので、を含まない項は消えています。  
微分の記号はの代わりにを使います。

**合成関数の微分公式**

こちらは高校数学の範囲内だと思いますが、復習しておきます。

がの関数で、さらにがの関数とします。  
このとき、 をで微分したについて、下記のような公式があります。



**合成関数の偏微分公式**

これは大学数学の範囲だと思います。

がの関数（複数変数を持つ）で、 がそれぞれの関数である場合、 をで偏微分すると下記のようになります。



これをこの記事では合成関数の偏微分公式の「難しい方」と呼ぶことにします。  
（他所で言うと笑われるのでご注意ください。）

さらにこれの特殊ケースとして、「簡単な方」があります。  
のうち、実は 番目のみがの関数で、ほかはの影響を受けない場合を考えます。  
を表す式が を含まないのであれば、を で偏微分すると 0 になりますね。  
つまり、 の中の は、 の場合はすべて 0 になるので、の場合だけを考慮すればよくなります。



高校で習う方の公式とほぼ同じ形をしています。

繰り返しますが、偏微分において上記は特殊なケースです。  
が、今回は何度か出てきます。

使い方のポイントは下記です。

* 関数の変数〜のうち、が影響するのが1つだけだったら「簡単な方」を使う
* 関数の変数〜のうち、が影響するのが複数だったら「難しい方」を使う
* 難しい方には がつく

**勾配降下法の式**

ここから本題です。

単純パーセプトロンの説明で、勾配降下法について説明しました。  
多層パーセプトロンにおいても、その理論を用いて重みを更新します。  
前回新たに定義した記号を用いて勾配降下法の式を書き直すと、下記のようになります。



は任意の層番号、は層内でのユニット番号、はユニット内での重みの番号（= 1つ前の層のユニット番号）です。  
は誤差関数ですが、単純パーセプトロンと同じ誤差関数が使えないため、新しく定義します。

**多層パーセプトロンの誤差関数**

多層パーセプトロンはネットワーク全体で1つの学習機械ですから、誤差関数も全体で1つだけ定義します。  
誤差関数は次のように定義されます。[1](http://hokuts.com/2016/05/29/bp1/#fn-1104-error_func)



は出力層番目のユニットの出力値ですから、 はそのユニットの「理想の出力値と実際の出力値の差」です。

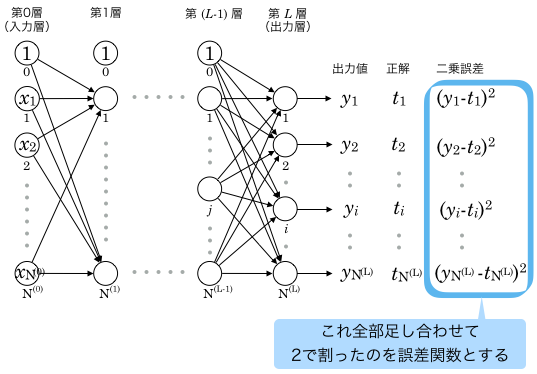
まずそれを二乗しています。二乗しているのは後の計算で都合がいいからです。負の値にならないメリットもありますね。

は、出力層の全ユニットについてその計算をして足し合わせるということです。[2](http://hokuts.com/2016/05/29/bp1/#fn-1104-sigma)

そしてさらに をかけています[3](http://hokuts.com/2016/05/29/bp1/#fn-1104-half)。これも後の計算で都合がいいからという理由です。

誤差を二乗した値の和（の定数倍）なので、このを**二乗誤差**と呼びます。

番目のユニットのみに注目した場合の も二乗誤差と言ったりしますが、あまり気にしないでください。



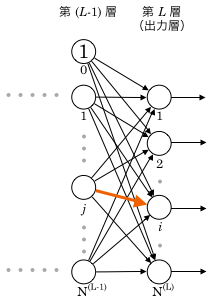
ですから、次のようにも書けます。



これは、 が の関数であることを意味しています。

**出力層の重み更新**

誤差関数が決まったので、勾配降下法を適用することを考えます。  
まず、出力層番目のユニットの番目の重み、 に注目します。



**更新式の展開**

この重みの勾配降下法による更新式は



です。  
この中の を解いていきます。

は の関数、 は の関数、 は の関数と見なせます。  
また、は出力層番目以外のユニット出力値も変数として持ちますが、上図を見ると分かるとおり、 の影響は 番目のユニット以外には及びません。  
よって、合成関数の偏微分公式の「簡単な方」を使い、下記が成り立ちます。



3項に分解できました。1項ずつ見てみましょう。  
後ろの項の方が今までの知識で理解できるので、後ろから順にいきます。

**第3項:** 

は、出力層の一つ前、つまり第層の各ユニット出力に重みをかけて合計したものなので、下記のようになります。





**第2項:** 

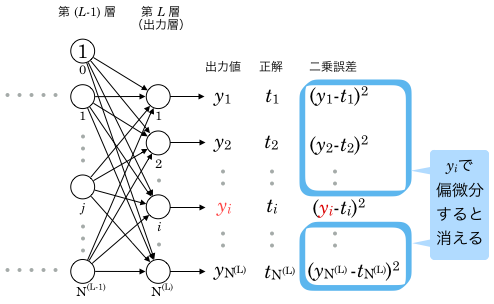
は出力層番目のユニットの出力ですから、 です。



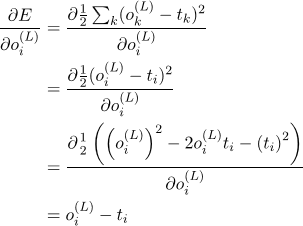
つまりこれは活性化関数の微分です。[4](http://hokuts.com/2016/05/29/bp1/#fn-1104-delta_activation)  
実際の式は活性化関数の定義によって決まります。

**第1項:** 

出力層 番目のユニット出力値 （）で誤差関数を偏微分するということです。  
下図をご覧ください。



誤差関数は、出力層すべてのユニットの二乗誤差を合計して したものでした。  
これを（）で偏微分すると 以外の変数はすべて定数と見なして消えるので、 番目以外のユニットの二乗誤差はすべて無視することができます。  
よって、

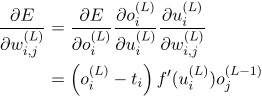


となります。[5](http://hokuts.com/2016/05/29/bp1/#fn-1104-error_half)

2行目で番目以外の項を消しています。

**出力層ユニットの重み更新式**

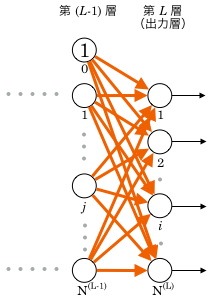
以上より、



なので、これを勾配降下法の式に当てはめると、 の更新式は下記になります。



複雑に見えますが、プログラムとして実装すると大したことはありません。  
, の値を変えていくと、出力層ユニットにひもづく重みはすべて更新できます。



**出力層より1つ前の層の重み更新**

上記の式で多層パーセプトロン内のすべての重みが更新できるといいのですが、出力層以外の重みには別の更新式が必要になります。

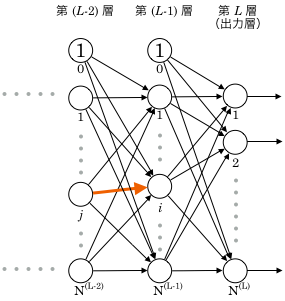
出力層より一つ前の層、つまり第層のユニットに紐付く重みについて考えます。

ここでも前出の誤差関数 を使って勾配降下法を適用します。  
第層向けに誤差関数を新しく定義したりはしません。

第層 番目のユニットの番目の重み、 に注目します。

なお、前の節でも , という記号を使いましたが、今回の , はそれとは別だと思ってください。

新たに記号を定義するのが大変なので使い回しています。



**更新式の展開**

この重みの勾配降下法による更新式は



です。

前回少し触れましたが、出力層の出力値は、第層の出力値を使った式に展開することができます。  
つまり誤差関数中に現れる もすべて を含む式に展開できるということです。

その状態ので考えてみます。

展開後の式はとても長くなるのでどんな式かはハッキリわからなくていいですが、 が消えて の関数になっていることだけ意識しておいてください。

は の関数であり、また、 は の関数で、 は の関数です。  
そしてこの は第層 番目以外の出力値も変数として持ちますが、 上の図を見るとわかるように、 が影響を与えるのは、第層だと 番目のユニットだけです。（第層まで伝播すると複数のユニットに影響しますが、 は今から消えているのでした。下記では、微分のパラメータとしても 第層に関する変数を使っていません。）

よって、合成関数の偏微分公式の「簡単な方」を使って下記のように展開できます。



これも後ろから一項ずつ見ていきます。

**第3項:** 

これは層番号が違うだけで、出力層の時と同じパターンです。

**第2項:** 

これも同じですね。層番号だけが違います。





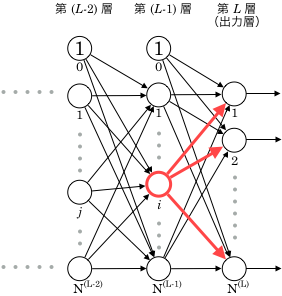
**第1項:** 

これが出力層より少し厄介です。  
第層番目のユニット出力値で誤差関数を偏微分しています。  
を の関数と見なしているのでしたが、この関数はよくわからないくらい長いのでした。  
このままだと計算がしにくいので、改めてを元の定義（ の関数）に戻し、合成関数の偏微分公式でさらに分解します。

使うのは難しい方です。



「難しい方」を使ったのは、 が第層のユニットすべてに影響を与えるからです。



**もう少し解いてみる**

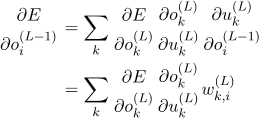
上記の式、中の第三項、 を解きます。

は第層の各ユニット出力に第層番目のユニットの重みを掛けて合計したものなので、





です。  
これを前述の式に代入すると下記のようになります。



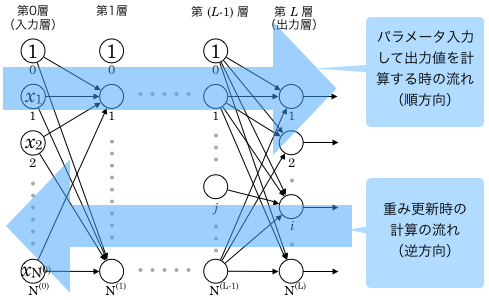
の中に、偏微分の記号を使った項があと2つ残っていますね。  
この2項の中の各変数を見てみると、すべて層番号が となっています。  
実はこの2項、第層の重み更新を計算する際にすでに算出していることにお気づきでしょうか？  
それがポイントになってきます。

**誤差逆伝播法の発想**

流れでいくと第層の重み更新式を記載したいところですが、ここであえて多層パーセプトロンの学習法の概要を書きます。

第層の重み更新を先に計算すると、その計算結果の一部を第層の重み更新の際に利用できそうでした。  
プログラム上では一時変数に保持しておけば使えますね。  
それを使って第層の重み更新の計算をし、さらにその計算結果の一部を第層の重み更新に使い…ということができると、第1層の重みまですべてを更新することができます。

これが**誤差逆伝播法**という重み更新法の考え方です。



多層パーセプトロンの入力層に を入力した時は、重みを掛けながら第1層、第2層…と伝播していき、出力層から が出力されます。

重み更新の際には、まず出力層の重み更新を計算し、その計算結果の一部を伝播させながら 第層、第層…という “逆方向” の順番に計算していくことになります。

誤差 “逆” 伝播法 というのはそういう意味です。

重み更新は、プログラムだと出力層から順にループで処理していく形になります。  
前の節で第層の更新について考えたとき、層番号以外は第層の時とほとんど同じ形の式が現れたことを思い出してください。  
これもループに適していそうですよね。

**更新式の決定版は次回**

ここまでは第層と第層に注目しましたが、全ての層に適用できる更新式を導くため、任意の第層で一般化して考えます。  
長くなったのでまた次回。

1. ベクトルとノルム（詳細は割愛）を使うと次のようにも書けます。  
   

また、簡単のため、この定義はオンライン学習でベクトル を1件だけ入力した場合の誤差として記述しています。  
一括学習やバッチ学習で、複数の教師データを入力して全ての誤差の和（または平均）を計算するという場合は、この誤差関数にさらにがもう1つつきます。

1. の真下に が表示されているのと の右下に が表示されているのは同じ意味です。  
   この記事で使用している数式表示のシステムの仕様です。
2. 定数倍しても相対的な大小は変わりません。
3. 多層パーセプトロンの活性化関数としてステップ関数が使えない理由はこれです。  
   ステップ関数では微分値が0になるので、重みの更新量も0になり、つまり更新されないのです。
4. を で偏微分したときに現れる係数 が、誤差関数の定義で付与した と打ち消し合っているので、結果的に係数が消えてシンプルになっています。  
   誤差関数でをつけたのはこのためです。

# ６　デルタの定義

ここで、今回とても重要になる定義をします。  
任意の第層 番目のユニットの重み付き和を のように書くと定義しましたが、この で誤差関数 を偏微分したものを とおきます。



上記はすべての出力ユニット[1](http://hokuts.com/2016/10/09/bp2/#fn-1485-all_unit)でそれぞれ定義されることにご注意ください。

例： 、

なお、この値の意味は…と深く考える必要はありません。  
後の計算式がわかりやすくなるように定義したものです。

はギリシャ文字の （デルタ）の小文字です。  
以下、これらの値をそのまま「デルタ」と呼ぶことにします。[2](http://hokuts.com/2016/10/09/bp2/#fn-1485-delta)

**デルタを使って出力層の更新式を表す**

出力層にひもづく重みの更新式は前回導きましたが、これをデルタを使った式で改めて導き直してみます。  
まず出力層（第層）番目のユニット出力値の式は下記ですね。

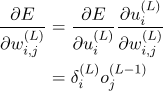




勾配降下法による重み更新式はこうでした。（第層 番目ユニットの 番目の重み）



上記の について

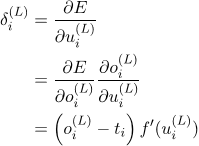


となります。  
よって出力層の重みの更新式は



と表せます。  
随分シンプルな式になりました。

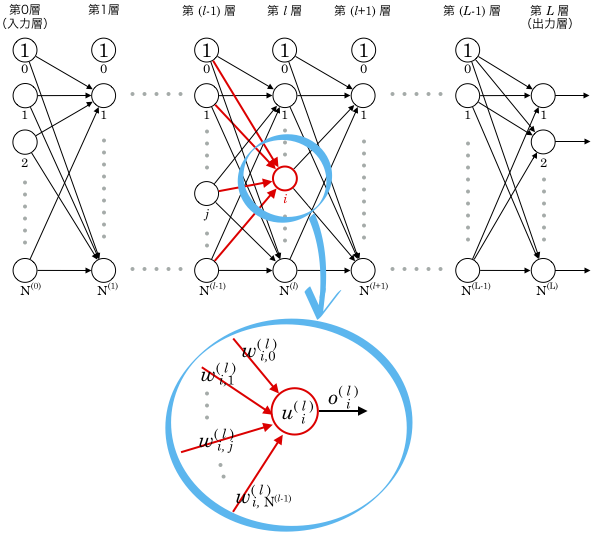
ところで、出力層のデルタはどんな式で表せるでしょうか。展開してみます。



項ごとの変形は前回やったことと同じですので、意味がわからない場合は前回の記事を参照してみてください。

**デルタを使って出力層以外の更新式を表す**

出力層以外の任意の第層について考えてみます。



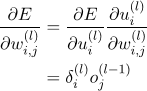
第層の番目のユニットの式は下記のようになります。



勾配降下法による重み更新式はこうですね。（第層 番目ユニットの 番目の重み）



上記の は



と変形できるので、更新式は下記のようになります。

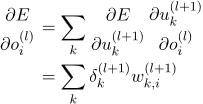


出力層の場合とほとんど同じ式になっていることがわかるでしょうか？

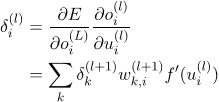
ただし、デルタを展開すると、出力層とはだいぶ違う式になります。  
展開してみましょう。下記の部分までは同じです。



上式の最終行 第1項は、下記のように展開できます。  
（前回名付けた、偏微分の公式の「難しい方」を使っています。）



式中に第層のデルタが出現しました。  
前出のデルタ展開式に上記を当てはめます。



第層のデルタを表現するのに、第層のデルタを使うことができました。  
これは、下記のことを表しています。

* 上の層の更新式を計算すると、その計算結果の一部を下の層の更新式に使い回すことができる
* ただし、この式は出力層では使えない

1つ目については、前回も触れた内容ですね。  
2つ目については、層番号は出力層のが最大で、第層というのはないからです。  
つまりデルタの式は、出力層（）とそれ以外（）で違うということです。  
出力層のデルタは、この記事の前半で式展開しましたね。

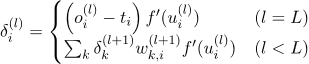
まとめます。

**多層パーセプトロンの重み更新アルゴリズム**

多層パーセプトロンの第層番目ユニットの番目の重みの更新式は下記で表せます。



ただし、



更新の手順は下記です。

* 多層パーセプトロン内のすべての重みをランダムに初期化
* 学習が完了するまで下記を繰り返す
  + 多層パーセプトロンにパラメータを入力し、出力誤差を計算
  + 上記更新式に従い、出力層から順にすべての重みの更新量を計算
  + 計算した更新量をすべての重みに適用

既に述べましたが、この手法は **誤差逆伝播法** または **バックプロパゲーション** と呼ばれます。

これで多層パーセプトロンの実装ができます。