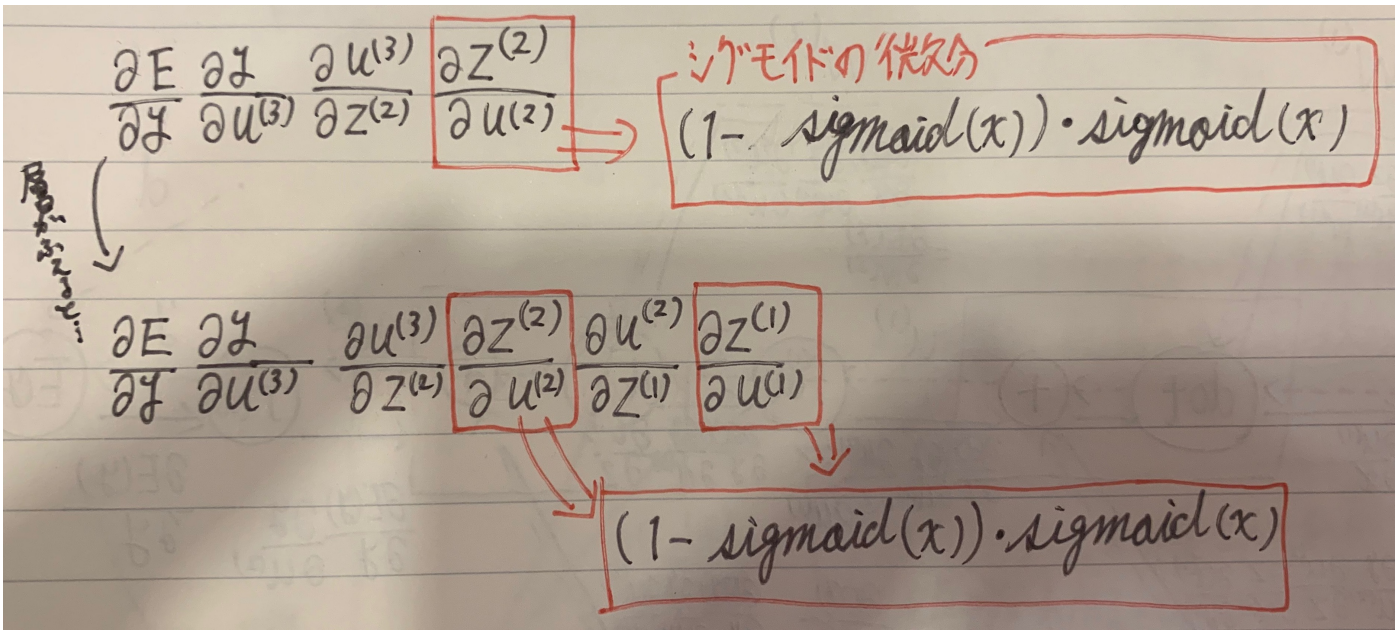
勾配消失問題

ニューラルネットワークの火付けとなった活性化関数でシグモイド関数の紹介をした。
シグモイド関数： $f(u) = \frac{1}{1+e^{-u}}$

シグモイド関数は、確かにステップ関数では表現できなかった0~1の値を表現することを可能としていた。
しかし、層が深くなればなるほど下位層(入力層側に近い層)にはバックプロパゲーションによるフィードバックが小さくなる**勾配消失問題**が起こっていた。

なぜ勾配消失問題が起きてしまうのか？

勾配消失問題の例として、4層のネットワークを考えてみる。(入力層１つ、中間層2つ、出力層1つ)
下位層に行くに従って、シグモイド関数の一次導関数を乗算する回数が増える例を下記に示した。



勾配消失問題が引き起こる理由は、
シグモイド関数の一次導関数の取りうる値の範囲が0~0.25であること、そして層が増えるにつれて乗算回数が増えるためである。
また、シグモイド関数の一次導関数の最大値は0.25である。(xが0の時最大値を取る。)

深層学習Day1でも書いたが、この勾配消失問題を対処するために**RELU関数**が誕生した。(スパース問題も軽減された。)

以後話すことではあるが、実は勾配消失問題対策でRELU関数以外にも解決策が出されていた。

勾配消失問題対策

・活性化関数(RELU関数)
・**初期値の設定**
・**バッチ正則化**

重みの初期化設定

勾配消失問題は、初期値の設定によりある程度は軽減される。
今回紹介する初期化設定の手法は下記である。
・**Xavier**(ザビエルと読むらしい)
・**He**

重みの初期化設定: Xavier

初期値の設定方法
　重みの要素を、前の層のノード数の平方根で除算した値

Xavierの初期値を設定する際の活性化関数
・ReLU関数
・シグモイド(ロジスティック)関数
・双曲線正接関数

In [3]:

```
#Xavierによる初期化
# network['W1'] = np.random.randn(input_layer_size, hidden_layer_size) / np.sqrt(input_layer_size)
# network['W2'] = np.random.randn(hidden_layer_size, output_layer_size) / np.sqrt(hidden_layer_size
```

重みの初期化設定: He

Relu関数のための初期化設定手法。(Relu関数をさらに生かす目的で誕生した。)
Relu関数によってある程度は勾配消失は抑えられるが、*He*を用いてさらに勾配消失問題を抑える

初期値の設定方法
　重みの要素を、前の層のノード数の平方根で除算した値に対し$\sqrt{2}$をかけ合わせた値
　Xavierの手法に対して$\sqrt{2}$をかけ合わせた手法になる。

In [4]:

```
# Heによる初期値
# network['W1'] = np.random.randn(input_layer_size, hidden_layer_size) / np.sqrt(input_layer_size) * n
# network['W2'] = np.random.randn(hidden_layer_size, output_layer_size) / np.sqrt(hidden_layer_size
```

バッチ正則化

バッチ正規化の効果

・計算量の高速化
・勾配消失が起こりにくくなる

ミニバッチ単位で，入力値のデータの偏りを抑制する
結果として、データのばらつきが無くなる(正規化)
⇨データがコンパクトになる。

バッチ正則化のタイミング

活性化関数に値を渡す前後に，バッチ正規化の処理を含めた層を加える

学習率最適化手法

勾配降下法にて、学習率の話を少し触れた
復習として学習率について簡易的にまとめておく

学習率の値が大きい場合
・最適値にいつまでもたどり着かず発散する
学習率の値が小さい場合
・発散することはないが、小さすぎると収束するまでに時間がかかる
・大域局所最適値に収束しづらくなる

では、学習率はどうやって決めるのだろうか？

初期の学習率設定方法の指針として下記であれば良いように考えられる
・初期の学習率を大きく設定し、徐々に学習率を小さくしていく
・パラメータ毎に学習率を可変させる

上記のようなことを実現させるために**学習率最適化手法**がある
次から、学習率最適化手法を用いた学習率の最適化について説明する

学習率最適化手法は下記のような手法がある

・モメンタム
・AdaGrad
・RMSProp
・Adam

モメンタム

誤差をパラメータで微分したものと学習率の積を減算した後、
現在の重みに前回の重みを減算した値と慣性の積を加算する

モメンタムによるパラメータ更新式は下記の②である。
$V_t = \mu V_{t-1} - \varepsilon \nabla E \quad ...①$
$w^{(t+1)} = w^{(t)} + V_t \quad ...②$
μ：**慣性**

最初、上記の文章を見てよくわからなかったが文章通りに式変形してみると理解できた。
②の式を次に更新する$w^{(t+1)}$ではなくて、現在の重み($w^{(t)}$)について式に変形する。
すると、現在の重み($w^{(t)}$)は、下記の式になる。
$w^{(t)} = w^{(t-1)} + V_{t-1}$

さらに式変形すると下記になる。
$V_{t-1} = w^{(t)} - w^{(t-1)} ...③$

この③式を①式のV_{t-1}に代入をすると下記の式になる。
$V_t = \mu(w^{(t)} - w^{(t-1)}) - \varepsilon \nabla E \quad ...④$

④式を②式のV_t代入すると下記の式になる。
$w^{(t+1)} = w^{(t)} + \mu(w^{(t)} - w^{(t-1)}) - \varepsilon \nabla E \quad ...⑤$

⑤式がモメンタムの説明内容を表しており、文章と数式の色を対応づけさせた。
式変形により得られた⑤式から、**前回の重みの影響を受けている(赤色)**という意味がわかった。

実装に必要な式は、①と②であることは忘れずに！
数式とcodeは下記である。

################## **モメンタム** ####################
$V_t = \mu V_{t-1} - \varepsilon \nabla E$
code:
$self.v[key] = self.momentum * self.v[key] - self.learning\_rate * grad[key]$

$w^{(t+1)} = w^{(t)} + V_t$
code:
$params[key] \mathrel{+}= self.v[key]$
#####################################

モメンタムのメリット
・局所的最適解にはならず、大域的最適解となる。
・谷間についてから最も低い位置(最適値)にいくまでの時間が早い。

AdaGrad

誤差をパラメータで微分したものと再定義した学習率の積を減算する

$h_0 = \theta$
$h_t = h_{t-1} + (\nabla E)^2$
$w^{(t+1)} = w^{(t)} - \varepsilon \frac{1}{\sqrt{h_t} + \theta} \nabla E$

###################### **AdaGrad** ####################
$h_0 = \theta$
code:
$self.h[key] = np.zeros\_like(val)$

$h_t = h_{t-1} + (\nabla E)^2$
code:
$self.h[key] \mathrel{+}= grad[key] * grad[key]$
$w^{(t+1)} = w^{(t)} - \varepsilon \frac{1}{\sqrt{h_t} + \theta} \nabla E$
code:
$params[key] \mathrel{-}= self.learning\_rate * grad[key]/(np.sqrt(self.h[key]) + 1e-7)$
#####################################

AdaGradのメリット
・勾配の緩やかな斜面に対して、最適値に近づける
課題
・学習率が徐々に小さくなるので、鞍点問題を引き起こす事があった

RMSProp

誤差をパラメータで微分したものと再定義した学習率の積を減算する

$h_t = \alpha h_{t-1} + (1 - \alpha)(\nabla E)^2$
$w^{(t+1)} = w^{(t)} - \varepsilon \frac{1}{\sqrt{h_t} + \theta} \nabla E$

###################### **RMSProp** ####################
$h_t = \alpha h_{t-1} + (1-\alpha)(\nabla E)^2$
code:
$self.h[key] * = self.decay\_rate$
$self.h[key] += (1 - self.decay\_rate) * grad[key] * grad[key]$

$w^{(t+1)} = w^{(t)} - \varepsilon \frac{1}{\sqrt{h_t}+\theta} \nabla E$
code:
$params[key] -= self.learning\_rate * grad[key] / (np.sqrt(self.h[key]) + 1e-7)$
#####################################

RMSPropのメリット
・(AdaGradと比較して)局所的最適解にはならず、大域的最適解となる
・ハイパーパラメータの調整が必要な場合が少ない

Adam

Adamとは，モメンタムとRMSPropの良い部分を取り入れたもの

モメンタムの、過去の勾配の指数関数的減衰平均
RMSPropの、過去の勾配の２乗の指数関数的減衰平均
それぞれを含んだ最適化アルゴリズムである

過学習

過学習とは

テスト誤差と訓練誤差とで学習曲線が乖離すること。
特定の訓練サンプルに対して、特化して学習すること。

過学習の原因

・パラメータが多い
　変なところを学習してしまう可能性がある
・パラメータの値が適切でない
・ノードが多い…　など

⇨上記の原因らを総称して、ネットワークの自由度が高いと言う

過学習の対策

　ネットワークの自由度が高いため、過学習が引き起こされていた。
　そこでネットワークの自由度を制限し、過学習を抑制する手法ができた。
　⇨ **正則化**

正則化

・L1正則化、L2正則化
・ドロップアウト

荷重減衰

・過学習の原因
　重みが大きい値をとることで、過学習が発生することがある
　学習させていくと、重みにばらつきが発生する。
　重みが大きい値は、学習において重要な値であり、重みが大きいと過学習が起こる

・過学習の解決策
　誤差に対して、正則化項を加算することで、重みを抑制する
　過学習がおこりそうな重みの大きさ以下で重みをコントロールし、かつ重みの大きさにばらつきを出す必要がある

・L1、L2正則化

$E_n + \frac{1}{p}\lambda \|x\|_p$

$\|x\|_p = (|x_1|^p + \cdots + |x_n|^p)^{\frac{1}{p}}$

　p = 1の場合、 L1正則化と呼ぶ
　p = 2の場合、 L2正則化と呼ぶ

ドロップアウト

正則化で最も使われている手法
　NNの学習時に、ある更新で層の中のノードのうちのいくつかを無効にして（そもそも存在しないかのように扱って）学習を行い、
　次の更新では別のノードを無効にして学習を行うことを繰り返す。
　これにより学習時にネットワークの自由度を強制的に小さくして汎化性能を上げ、過学習を避けることができる。
　⇨**データ量を変化させずに，異なるモデルを学習させていると解釈できる**

問題：リッジ回帰の特徴として正しいものはどれか？
→ ハイパーパラメータに大きな値を設定すると，すべての重みが限りなく0に近づく.

畳み込みニューラルネットワーク

CNNとは

画像の分野で非常によく使われている
入力の加工によっては，音声などの時系列データに対しても適用できる
複数の**畳み込み層**と**プーリング層**で構成される

畳み込み層とは

入力値にフィルター（全結合でいう重み）を掛けて，出力値を得る。それにバイアスを足して活性化関数を通す。
フィルターは大域的な特徴を抽出するためのものと理解した。

RGB次元の場合は？
各チャネルに対して上記の処理と同様ことを行って最後に結果を画素ごとに全チャネルにわたって加算する

CNNのメリット
畳み込み層では，画像の場合，縦，横，チャンネルの３次元のデータをそのまま学習し，次に伝えることができる
また、チャネル間の相関関係も自動で学習(フィルタの値)してくれる優れもの。
３次元の空間情報も学習できるような層が，畳み込み層である

講義以外に参考にした文献も載せる
⇨http://www.orsj.or.jp/archive2/or60-4/or60_4_198.pdf (http://www.orsj.or.jp/archive2/or60-4/or60_4_198.pdf)

プーリング層とは

CNNでは大域的な領域における特徴を捉えると述べた。
プーリングはCNNで捉えた特徴量の中からさらに特徴を抽出する。
プーリングには様々種類があるが今回は二つ紹介する。

マックスプーリング
よく使われる。
畳み込み層の出力に対してあるサイズの領域を取り出し，その最大値を出力値とする
CNNで捉えた特徴量の中から最も際立った特徴を抽出すると解釈しても良いと思う

アベレージプーリング
畳み込み層の出力に対してあるサイズの領域を取り出し，その平均値を出力値とする

余談だが、最近では全結合を撤廃してグローバルアベレージプーリングが主流になりつつあるらしい。
⇨パラメーター数が計算量が減り、わりと精度が出せる。

パディング

入力画像の外側に架空の入力値を追加することで，入力画像のサイズを調整する手法。
畳み込み処理でどんどん画像が小さくなることを防ぐことができる。
さらに、パディングを入れることで畳み込み層を増やすことができる。
ゼロパディングするのが一般的。

なぜゼロなのか？
ランダムにしてしまうと値による影響を受けてしまう。
そのため、０にする。そうすることで、入力のデータの情報のまま学習ができる。

ストライド

フィルタのずらし方のこと
8x8，パディング1の入力画像に対して，3x3のフィルタをストライド2で作用させると3x3の出力画像が得られる
6x6，パディング1の入力画像に対して2x2のフィルタをストライド1で作用させると7x7の出力画像が得られる(確認テスト参考)

スライド幅が大きいと計算量が減るが、細かい特徴を見逃す可能性がある。
逆にスライド幅が小さいと計算量が増えるが、細かい特徴を見つける可能性がある。

チャンネル

入力画像を複数の空間に分ける（RGBなど）。分解した総数をチャンネル数という。
それぞれに対してフィルターを用意する。

全結合層で処理すると
縦，横，チャンネルの３次元データが１次元データとして処理される。
その結果RGBの各チャンネルの相関関係が学習に反映されない。

AlexNet

2012年の画像認識コンペティションで２位に大差をつけて優勝，ディープラーニングが大きく注目を浴びる
５層の畳み込み層およびプーリング層などそれに続く３層の全結合相から構成
過学習を防ぐ施策として，サイズ4096の全結合相の出力にドロップアウトを使用