6章
学習に関するテクニック

山田研
松田理美, 松川将也, 下田英寿, 山城京太郎, 玉橋優太郎

ニューラルネットワークの学習の目的

　損失関数の値をできるだけ小さくするパラメータを見つけること
=最適なパラメータを見つける

→このような問題を解くことを**最適化**と言う。

最適化は難しい

　パラメータ空間は非常に複雑で、数式を解いて一瞬で最小値を求めるといった方法は取れない。
→
最適な解は簡単には見つけられない。

ディープなネットワークになるほど、パラメータの数が膨大になり、より深刻な問題になっていく。

確率的勾配法

パラメータの勾配(微分)を使って、勾配方向にパラメータを更新するというステップを何度も繰り返して、徐々に最適なパラメータへと近づけていった。

→確率的勾配法(stochastic gradient descent) = SGD

SGDよりも賢い手法が存在する。

冒険家の話

ここに風変わりな冒険家がいます。彼は、広大な乾燥地帯を旅しながら、日々深い谷底を求めて旅を続けています。彼の目標は、最も深く低い谷底——彼はその場所を「深き場所」と呼ぶ——へたどり着くこと。それが彼の旅する目的です。しかも、彼は、厳しい“制約”を 2 つ自分に課しています。ひとつは地図を見ないこと、もうひとつは目隠しをすることです。そのため、彼には、広大な土地のどこに一番低い谷底が存在するのか分かりません。しかも、外は何も見えないのです。そのような厳しい条件の中、この冒険家は、どのように「深き場所」を目指せばよいでしょうか？ どのように歩を進めれば、効率良く「深き場所」を見つけることができるでしょうか？

冒険家の話

　最適なパラメータを探索するとき、冒険者と同じ、広大で複雑な地形を、地図もなく、目隠しをして「深き場所」を探さなくてはならない。

この状況で重要になるのが、地面の「傾斜」。
周りの景色は見えないが、今いる場所の傾斜は分かる。
→一番傾斜がきつい方向に進めば、いつかたどり着ける

これがSGD

SGD

SGDは数式で次のように表す。

$$W \leftarrow W - \eta \frac{\delta L}{\delta W}$$

w:更新する重みパラメータ
$\frac{\delta L}{\delta W}$:wに関する損失関数の勾配
η :学習係数
←:右辺の値で左辺の値を更新する

勾配方向へある一定の距離だけ進むのがSGD

PythonでのSGDクラス実装

SGDを、Pythonのクラスとして実装。

初期化の際のlrは学習係数を表す。
インスタンス変数として保持される。

updateメソッドは、SGDで繰り返し呼ばれることになる。

paramsとgradsは、それぞれ重みパラメータと勾配が格納されているディクショナリ変数。

```
class SGD:
    def __init__(self, lr=0.01):
        self.lr = lr

    def update(self, params, grabs):
        for key in params.keys():
            params[key] -= self.lr * grabs[key]
```

SGDクラスを使ってパラメータの更新

SGDのクラスを使ってmニューラルネットワークのパラメータの更新を行うことができる。

```
network = TwoLayerNet(...)
optimizer = SGD()

for i in range(10000):
    …
    x_batch, t_batch = get_mini_batch(...) #ミニバッチ
    grads = network.gradient(x_batch, t_batch)
    params = network.params
    optimizer.update(params, grads)
```

SGDの欠点

SGDは単純で実装も簡単だが、問題によっては非効率になる場合がある。
例として、次の関数の最小値を求める問題を考える。

$$f(x, y) = \frac{1}{20}x^2 + y^2$$

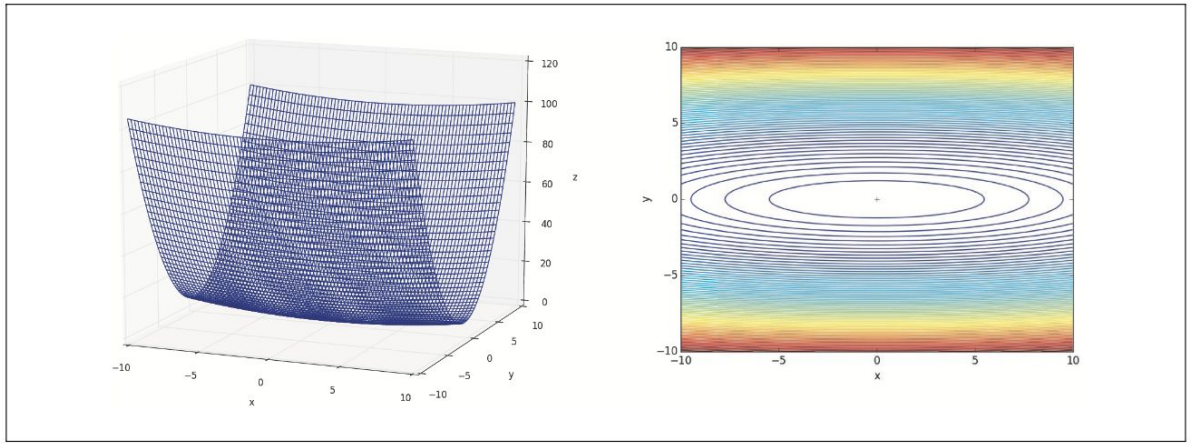


図6-1　$f(x, y) = \frac{1}{20}x^2 + y^2$ のグラフ（左図）とその等高線（右図）

勾配の方向

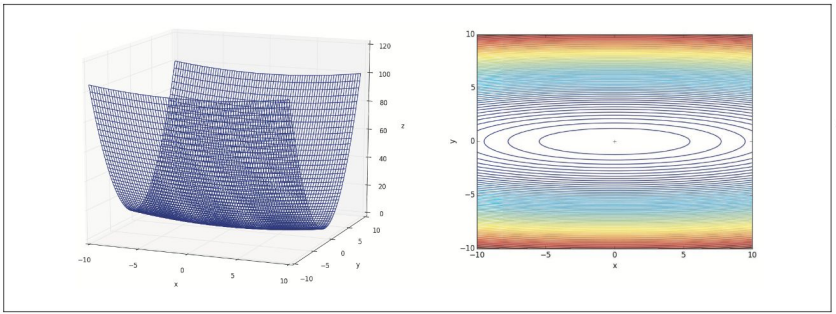


図6-1　$f(x, y) = \frac{1}{20}x^2 + y^2$ のグラフ（左図）とその等高線（右図）

y軸方向の勾配が大きく、
x軸方向の勾配が小さい。

最小値の場所は(0,0)だが、ほとんどの勾配が最小値の場所を指さない。

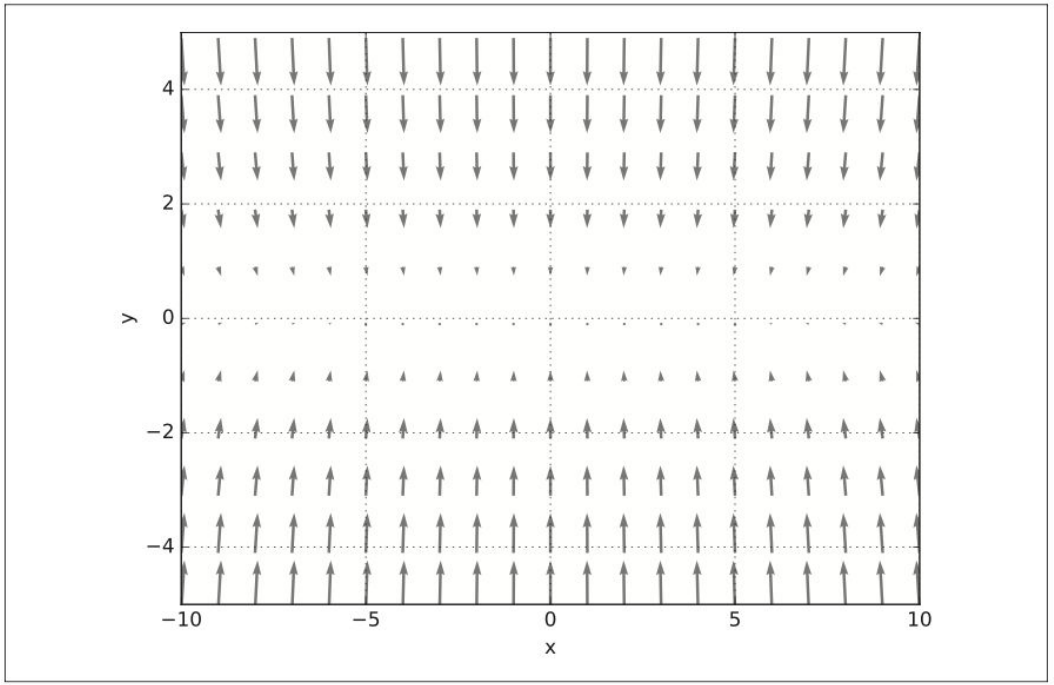


図6-2　$f(x, y) = \frac{1}{20}x^2 + y^2$ の勾配

SGDで解く

初期値は(x,y)=(-7.0,2.0)

ジグザグな動きをする(非効率)。

SGDの欠点
・関数の形状が等方的でないと、非効率な経路で探索する
　←勾配の方向が本来の最小値ではない方向を指しているため

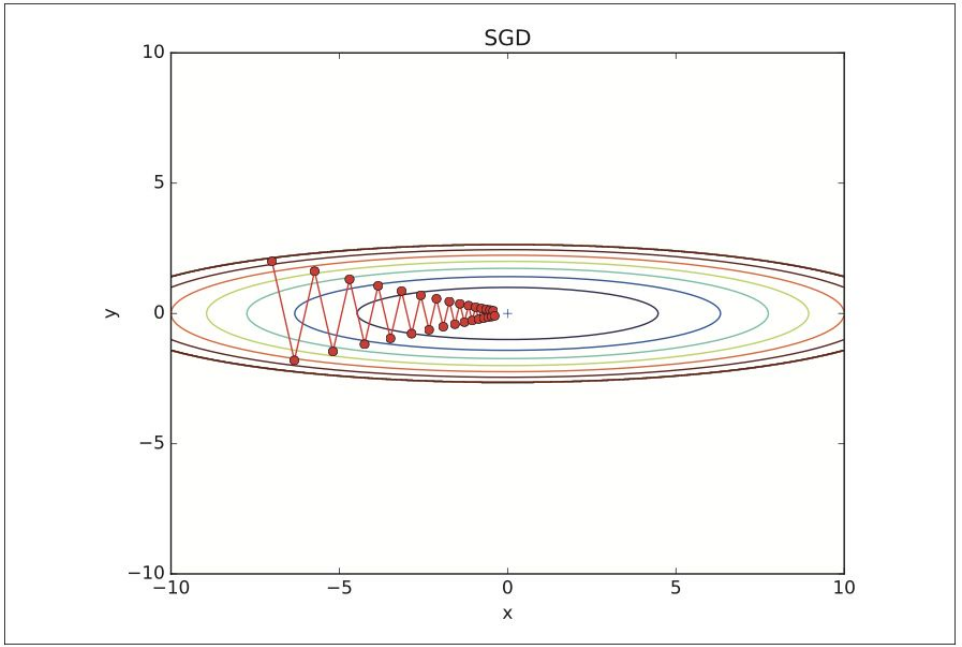


図6-3　SGD による最適化の更新経路：最小値の (0, 0) へジグザグに動くため非効率

Momentum

Momentumとは、「運動量」という意味。

$$v \leftarrow \alpha v - \eta \frac{\delta L}{\delta W}$$
$$W \leftarrow W + v$$

W:更新する重みパラメータ　　:\$\frac{\delta L}{\delta W}$関する損失関数の勾配

η:学習係数　　v:物理で言うところの「速度」

αv:物体が何も力を受けていないときに徐々に減速する役割

$0 \leq \alpha < 1$

物体が勾配方向に力を受け、その力によって物体の速度が加算されるという物理法則を表している。

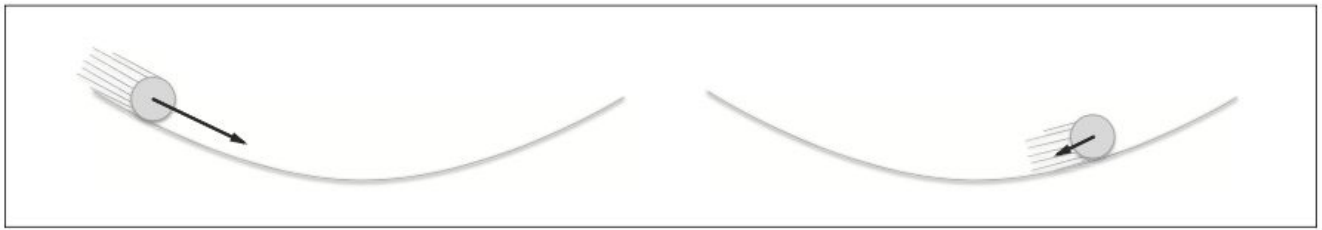


図6-4　Momentum のイメージ：ボールが地面の傾斜を転がるように動く

Momentumの実装

Momentumを、Pythonのクラスとして実装。

インスタンス変数のvは物体の速度を保持。初期化時は何も保持しない。

update()が初めに呼ばれるときに、パラメータと同じ構造のデータをディクショナリ変数として保持する。

```
class Momentum:
    def __init__(self, lr=0.01, momentum=0.9):
        self.lr = lr
        self.momentum = momentum
        self.v = None

    def update(self, params, grads):
        if self.v is None:
            self.v = {}
            for key, val in params.items():
                self.v[key] = np.zeros_like(val)

        for key in params.keys():
            self.v[key] = self.momentum*self.v[key] - self.lr*grads[key]
            params[key] += self.v[key]
```

Momentumで解く

↓SGD

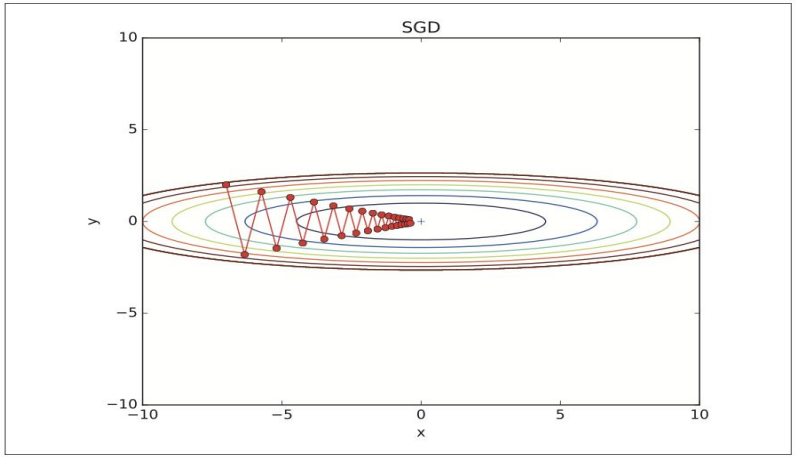


図 6-3　SGD による最適化の更新経路：最小値の (0, 0) へジグザグに動くため非効率

↓Momentum

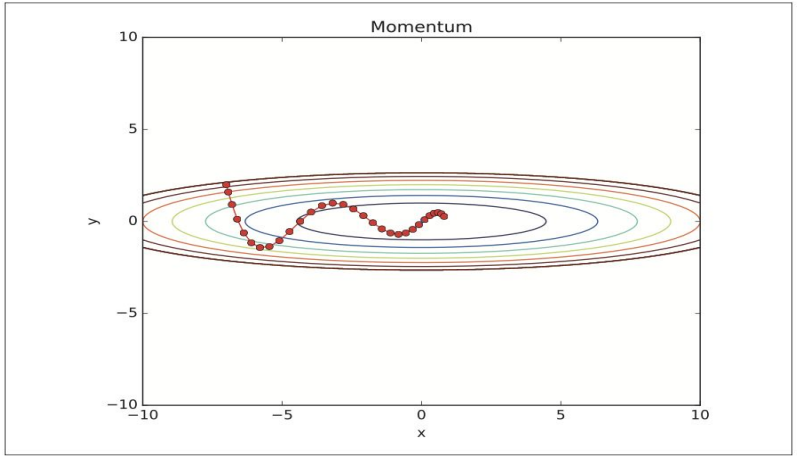


図 6-5　Momentum による最適化の更新経路

x軸方向に常に同じ方向の力を受けるため、同じ方向へ一定して加速している。
y軸方向に生と負の方向の力を交互に受け、それらが互いに打ち消し合い、y軸方向の速度は安定しない。それによって、x軸方向へ速く近づくことができる。

AdaGrad

学習係数の値が小さすぎると学習に時間がかかりすぎ、大きすぎると発散して正しい学習が行えない。
有効なテクニックとして、学習係数の減衰という方法がある。
→最初は大きく学習し、次第に小さく学習する手法
これを発展させたのが、AdaGrad。

$$h \leftarrow h + \frac{\delta L}{\delta W} \odot \frac{\delta L}{\delta W}$$
$$W \leftarrow W - \eta \frac{1}{\sqrt{h}} \frac{\delta L}{\delta W}$$

W:更新する重みパラメータ　　:$\frac{\delta L}{\delta W}$関する損失関数の勾配
η:学習係数　h:これまで経験した勾配の値を2乗和として保持
$\odot$:行列の要素ごとの掛け算

パラメータ更新の際に　$\frac{1}{\sqrt{h}}$乗算することで、よく動いたパラメータの学習係数を次第に小さくし、パラメータごとに学習係数の減衰を行う。

AdaGradの実装

```
class AdaGrad:
    def __init__(self, lr=0.01):
        self.lr = lr
        self.h = None

    def update(self, params, grads):
        if self.h is None:
            self.h = {}
            for key, val in params.items():
                self.h[key] = np.zeros_like(val)

        for key in params.keys():
            self.h[key] += grads[key] * grads[key]
            params[key] -= self.lr * grads[key] / (np.sqrt(self.h[key]) + 1e-7)
```

最後に1e-7という小さい値を加算しているのは、self.h[key]の中に0があった場合に0で除算してしまうことを防ぐ役割がある。

AdaGradで解く

　y軸方向への勾配が大きいので、最初は大きく動くが、その大きな動きに比例して、更新のステップが小さくなるように調整が行われている。
そのため、y軸方向への更新度合いは弱められていき、ジグザグの動きが軽減される。
SGD、Momentumより効率的に動いている。

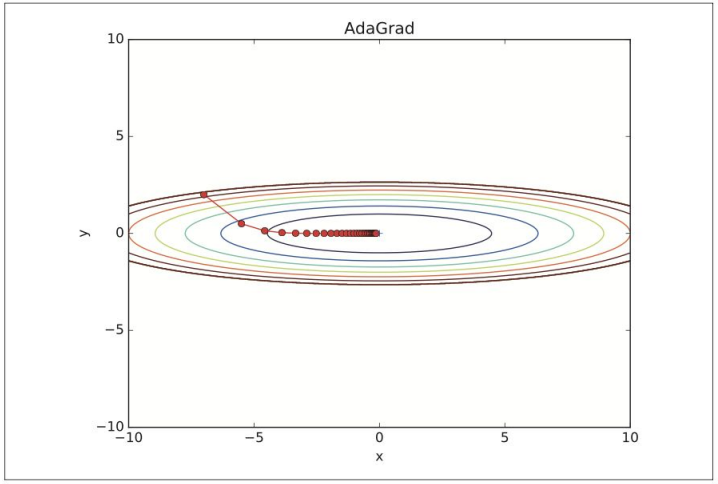
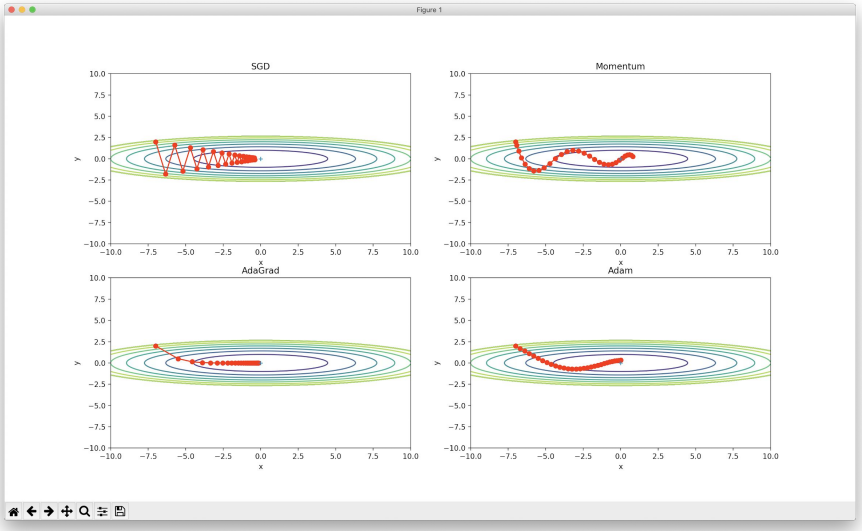


図6-6　AdaGrad による最適化の更新経路

4つの手法の違い

AdaGradが最も速く学習が行われている。

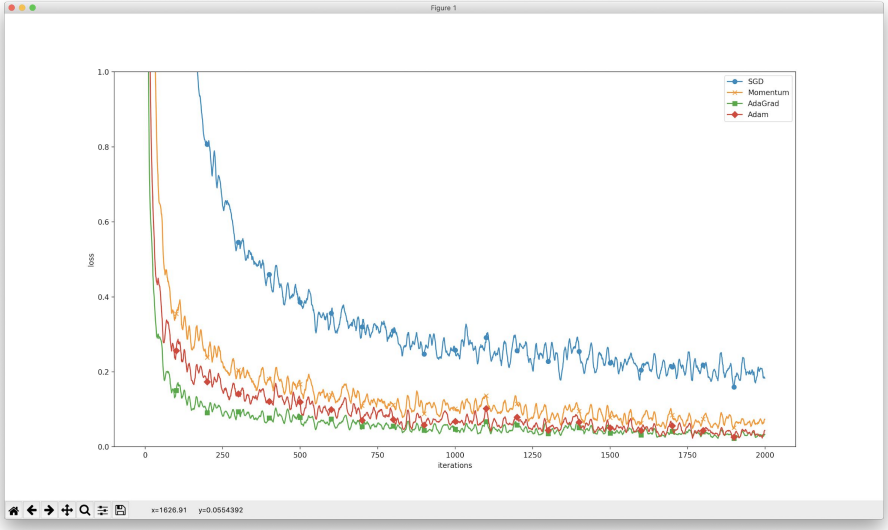
学習係数のハイパーパラメータによって結果は変わる。



4つの手法の違い(mnist)

5層のニューラルネットワークで、各層100個のニューロンを持つネットワーク。
活性化関数には、ReLUを使用。

AdaGradが最も速く学習が行われている。
学習係数のハイパーパラメータや、ニューラルネットワークの構造によって結果は変化するが、一般にSGDよりも他の3つの手法のほうが速く学習できる。



Adam

Momentumは、ボールがお椀を転がるような動きをしていた。
AdaGradは、パラメータの要素ごとに、更新ステップを調整していた。

この2つの利点を合わせた手法が、Adam。
ハイパーパラメータのバイアス補正が行われている。

6.2 重みの初期値

重みの初期値をどのように設定するかによって学習の成否が分かれる

ではどのように重みの初期値を設定すれば良いか？

6.2.1 重みの初期値を0にする？(過学習を抑える手法)

Weight decay(荷重減衰)という重みパラメータの値が小さくなるように学習する手法で, 過学習が起きにくくなる

重みを小さい値にしたければ初期値もできるだけ小さい値からスタートするのが正攻法である

6.2.1 重みの初期値を0にする？(0にするとどうなるか)

重みの値を小さくしたいのであれば0からスタートすれば良いのでは？

→重みの初期値を0にすると正しい学習が行えない

全ての重みの初期値を0(または均一な値)にしてしまうと重みが同様に更新されてしまうため，重みが重複した値を持つようになってしまう

→複数の重みを持つ意味がなくなってしまう

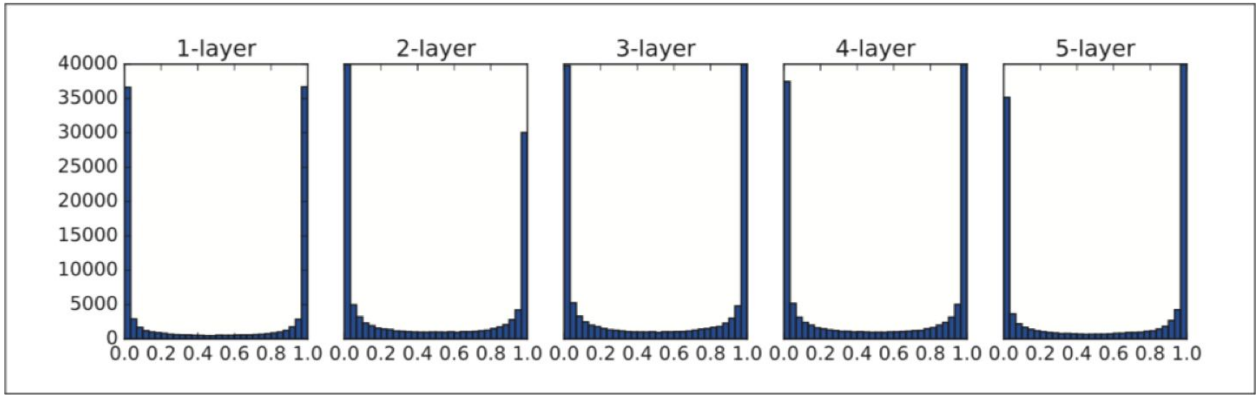
6.2.2 隠れ層のアクティベーション分布

隠れ層のアクティベーション(活性化関数の後の出力データ)の分布を観察することで多くの知見を得られる

5層のニューラルネットワーク(活性化関数にシグモイド関数を使用)に，ランダムに生成したデータを流し，各層のアクティベーションのデータ分布をヒストグラムで描画する実験を行う

6.2.2 隠れ層のアクティベーション分布(重み初期値に標準偏差1のガウス分布)

重み初期値に標準偏差1のガウス分布を用いて実験すると下図のような分布になる



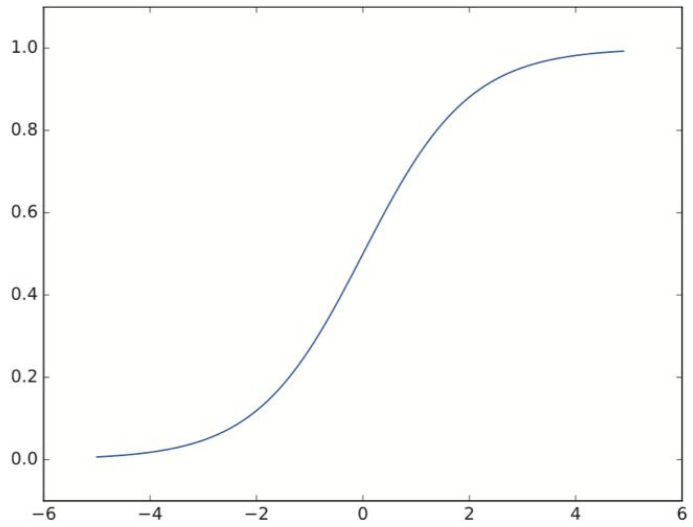
図を見ると, 0と1に偏っているのがわかる

6.2.2 隠れ層のアクティベーション分布(勾配消失)

ここで用いられているシグモイド関数は出力が0, または1に近づくにつれて微分の値は0に近づく

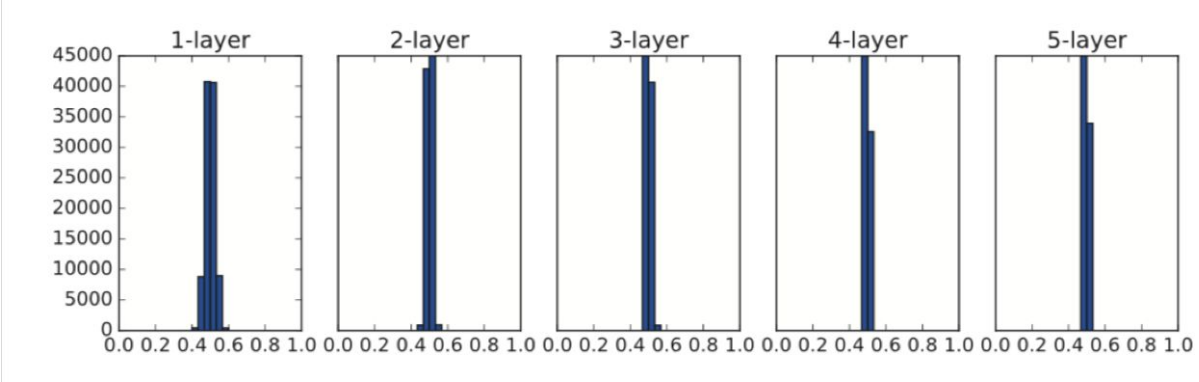
0と1に偏ったデータ分布では, 逆伝播の勾配の値が小さくなり消えてしまう

→この問題を勾配消失と呼ぶ



6.2.2 隠れ層のアクティベーション分布(重み初期値に標準偏差0.01のガウス分布)

重み初期値に標準偏差0.01のガウス分布を用いて実験すると下図のような分布になる



勾配消失の問題はないが, アクティベーションの偏りが大きい

6.2.2 隠れ層のアクティベーション分布(偏りが大きい場合の問題)

偏りが大きい = 複数のニューロンが同じ値を出力する

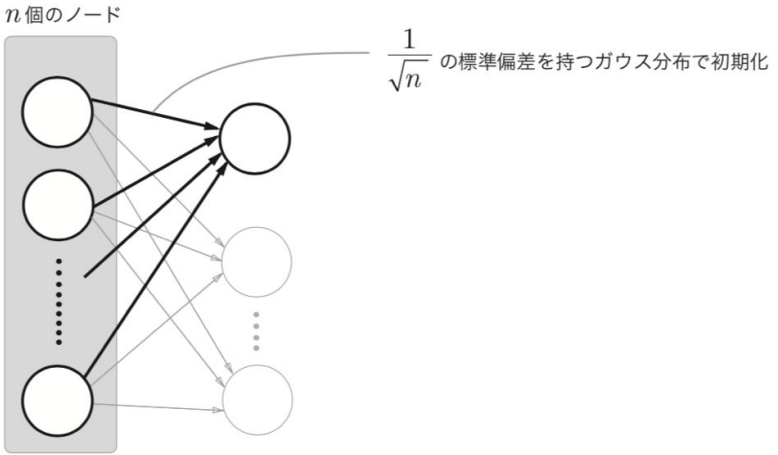
→複数ニューロンが存在する意味がなくなる

1個のニューロンでも同じことを表現できるため, 表現力に乏しい

6.2.2 隠れ層のアクティベーション分布(Xavierの初期値)

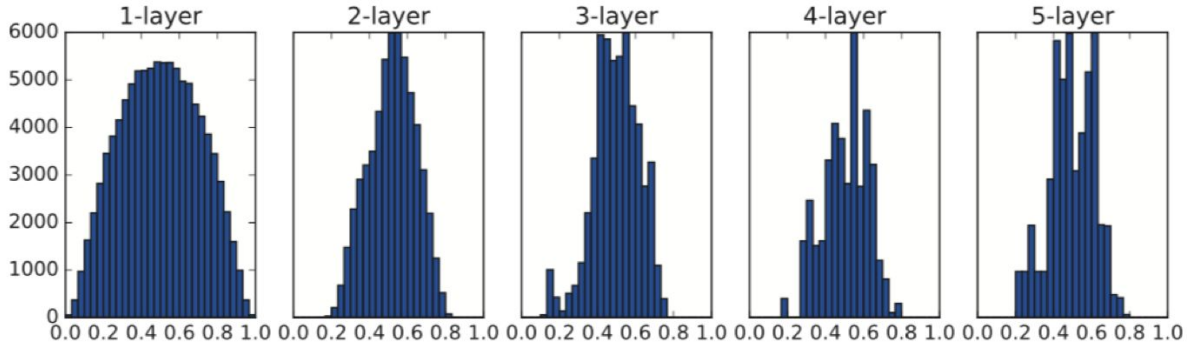
現在,「Xavierの初期値」は一般的なディープラーニングのフレームワークで用いられている

前走のノードの個数をnとした場合，1/√nの標準偏差を持つ分布を使う



6,2.2 隠れ層のアクティベーション分布(Xavierの初期値を適用)

「Xavierの初期値」を用いた結果は下図のようになる



これまでと比べて適度な広がりを持っている

6.2.2 隠れ層のアクティベーション分布(tanh関数)

シグモイド関数を用い「Xavierの初期値」を重みの初期値として分布を描画した場合は上位層の分布の形状がややいびつになる

シグモイド関数の代わりにtanh関数を用いると綺麗な釣鐘型の分布になる

どちらもS字カーブであるが, どの点で対称であるかが違う

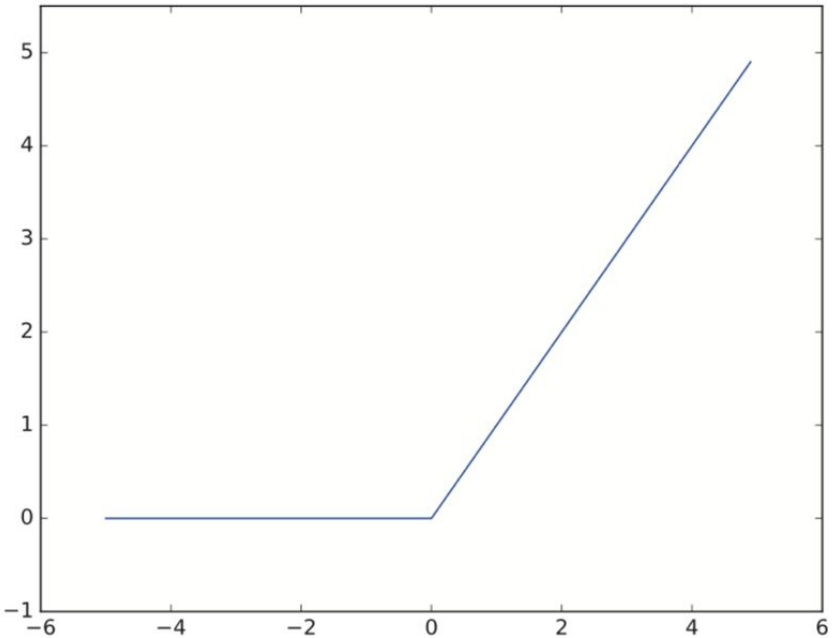
→活性化関数に用いる関数は原点対称であることが望ましい

6.2.3 ReLUの場合の重みの初期値

「Xavierの初期値」は，活性化関数が線形であることを前提に導いた結果である

→シグモイド関数やtah関数は中央付近が線形関数とみなせる

ReLUを用いる場合はReLUに特化した初期値を用いることが推奨される



6.2.3 ReLUの場合の重みの初期値(Heの初期値)

ReLU関数に適している初期値が「Heの初期値」である

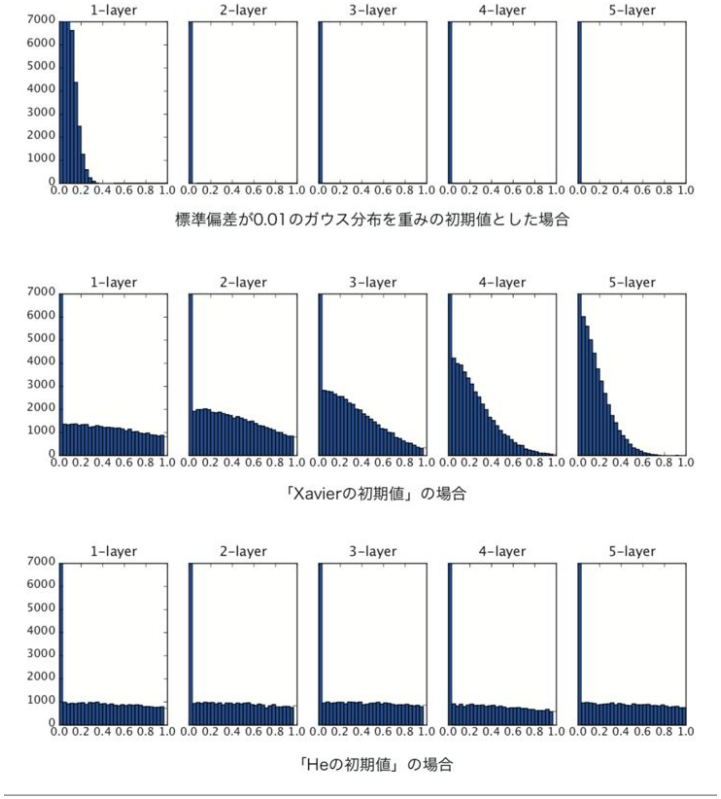
「Heの初期値」は, 前走ノードの数がn個の場合, √2/√nを標準偏差とするガウス分布を用いる

ReLU関数は負の領域が0となるため,「Xavierの初期値より広がりを持たせるよう倍の係数を持たせている」

6.2.3 ReLUの場合の重みの初期値(各層のアクティベーション)

右図にReLUを用いた場合のアクティベーションの分布を示す(標準偏差0.01のガウス分布,「Xavierの初期値」, Heの初期値」の場合

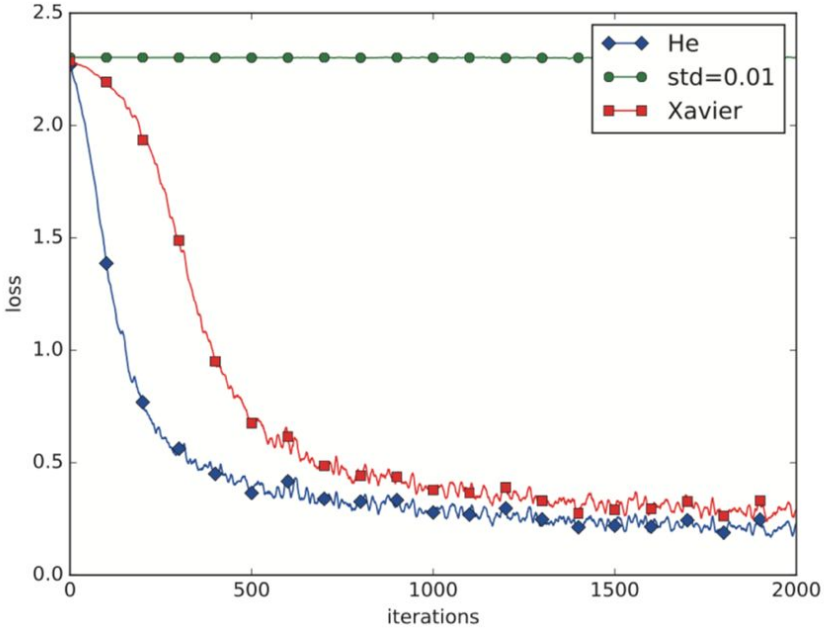
「Heの初期値は各層で分布の広がりが均一に広がっている」



6.2.4 MNISTデータセットによる重み初期値の比較

5層のニューラルネットワークで活性化関数にReLUを用い,「標準偏差0.01のガウス分布」,「Xavierの初期値」,「Heの初期値」の3つのケースで学習にどれだけの影響を与えるか比較する

結果は右図のようになる

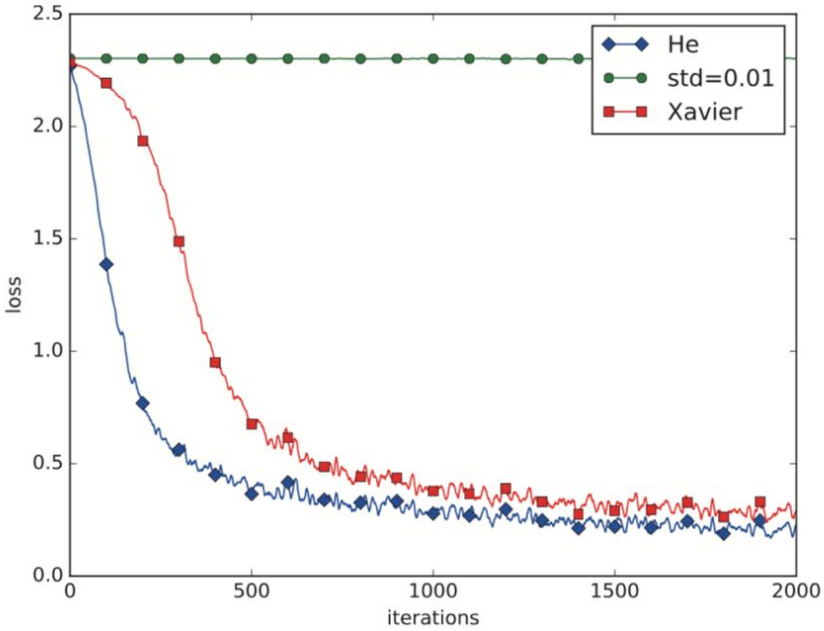


6.2.4 MNISTデータセットによる重み初期値の比較(図からわかること)

標準偏差0.01のガウス分布を初期値として用いた場合, 学習が進んでいない

「Heの初期値」と「Xavierの初期値」では順調に学習が行われ,「Heの初期値」の方が学習の進みが早い

ニューラルネットワークにおいて重みの初期値は重要だということがわかる



6.3 Batch Normalization

・各層で適度な広がりを持つように、”強制的”にアクティベーションの分布を調　整するアイデアを主軸とした手法

6.3.1　Batch Normalization のアルゴリズム

利点

・学習を早く進行させることができる

（学習係数を大きくすることができる）

・初期値にそれほど依存しない

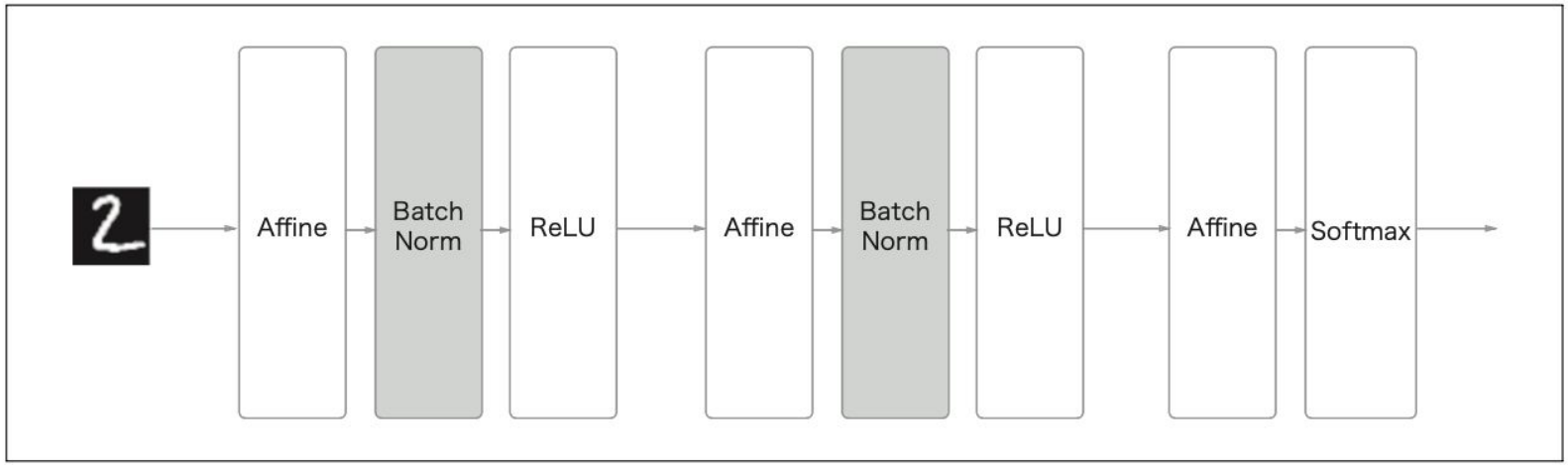
（初期値に対してそこまで神経質にならなくてよい）

・過学習を抑制する

（Dropout などの必要性を減らす）

6.3.1 Batch Normalization のアルゴリズム

Batch Normalization レイヤとして、データ分布の正規化を行うレイヤをニューラルネットワークに挿入します。



6.3.1 Batch Normalization のアルゴリズム

学習を行う際のミニバッチを単位として、ミニバッチごとに正規化を行う。

具体的には、

データの分布が平均が0で、分散が1になるように正規化を行う。

$$\mu_B \leftarrow \frac{1}{m}\sum_{i=1}^{m} x_i$$

$$\sigma_B^2 \leftarrow \frac{1}{m}\sum_{i=1}^{m}(x_i - \mu_B)^2$$

$$\hat{x}_i \leftarrow \frac{x_i - \mu_B}{\sqrt{\sigma_B^2 + \varepsilon}}$$

6.3.1 Batch Normalization のアルゴリズム

ミニバッチとして B = {x_1,x_2,…,x_m}　という m個の入力データ集合

入力データを平均が0で分散が1になる ―**適切な分布になる**― ように正規化する

ε は、小さな値（10e-7など）　→0で除算されるのを防止する

活性化関数の前後どちらかに挿入することで、データの分布の偏りを減らすことが可能になる

$$\mu_B \leftarrow \frac{1}{m}\sum_{i=1}^{m} x_i$$

$$\sigma_B^2 \leftarrow \frac{1}{m}\sum_{i=1}^{m}(x_i - \mu_B)^2$$

$$\hat{x}_i \leftarrow \frac{x_i - \mu_B}{\sqrt{\sigma_B^2 + \varepsilon}}$$

$$\{x_1, x_2, \cdots, x_m\}$$

平均0、分散1に変換

$$\{\hat{x}_1, \hat{x}_2, \cdots, \hat{x}_m\}$$

6.3.1 Batch Normalization のアルゴリズム

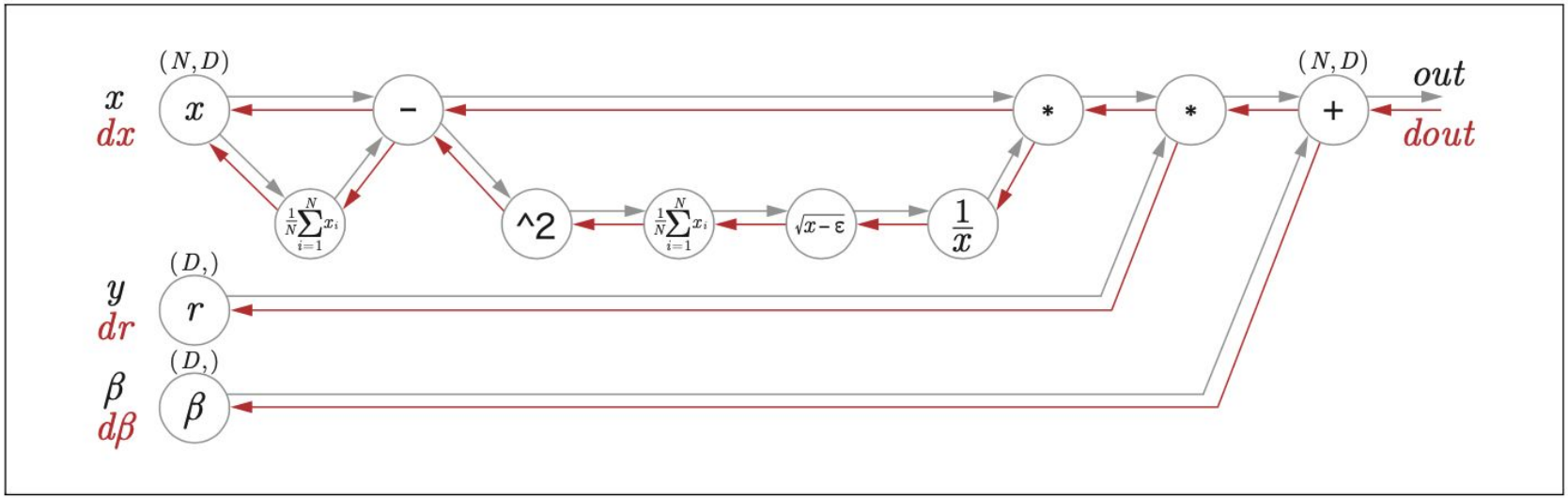
正規化されたデータに対して、固有のスケールとシフトで変換を行う。

γ と β はパラメータ

$\gamma = 1, \beta = 0$ からスタートして、学習によって適した値に調整される。

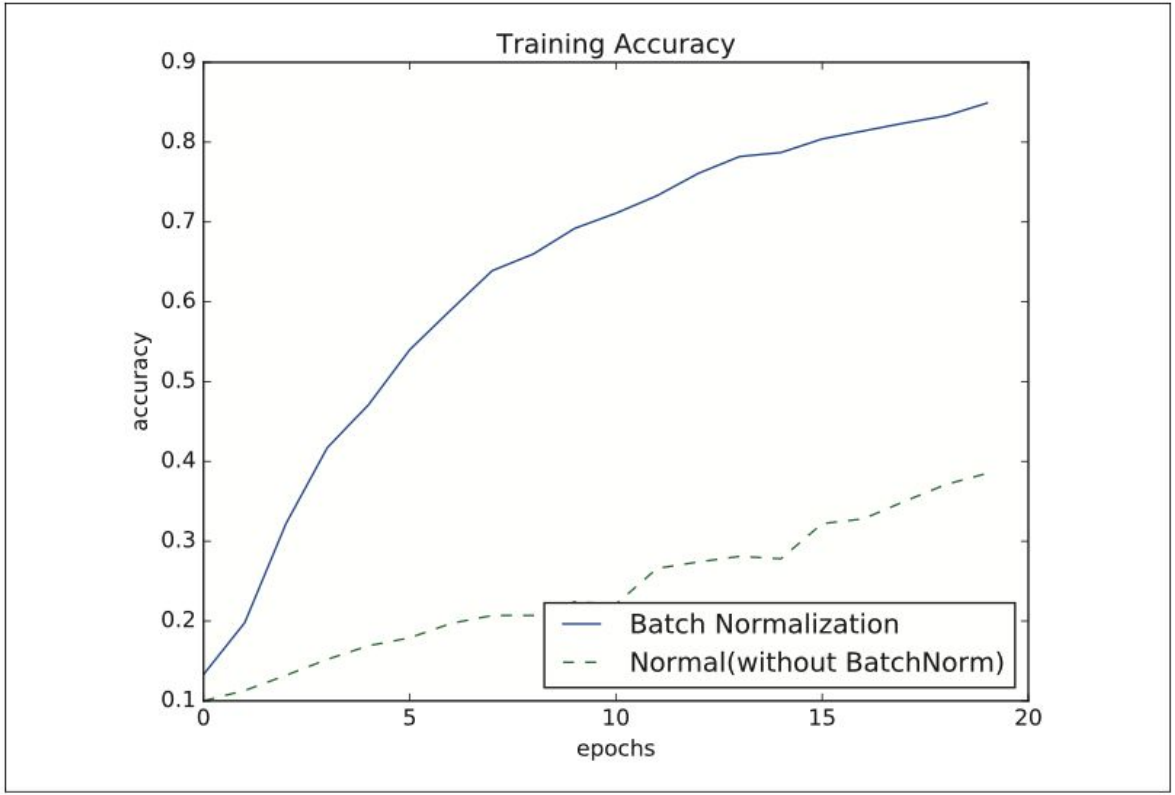
$$y_i \leftarrow \gamma \hat{x}_i + \beta$$

6.3.1 Batch Normalization のアルゴリズム

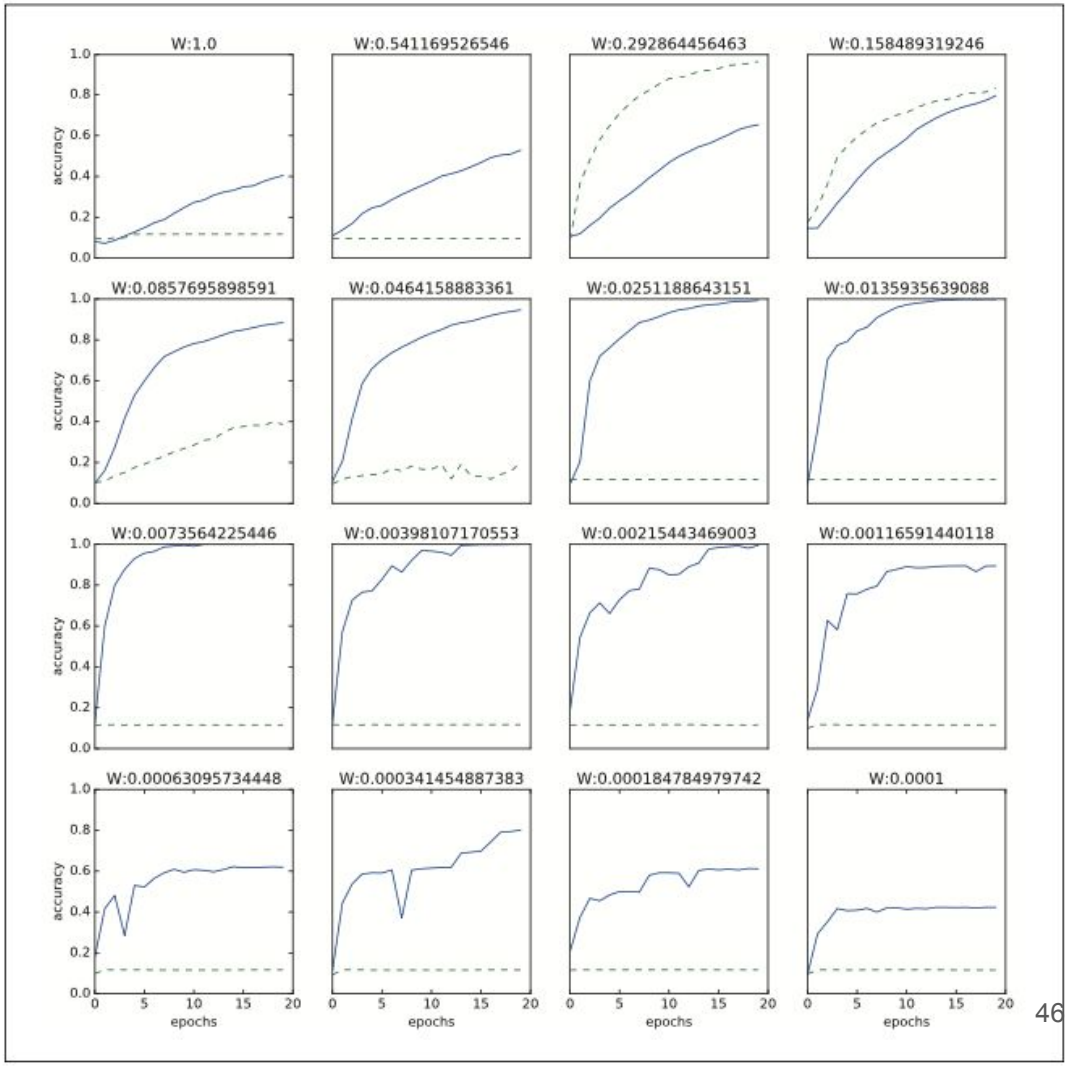


詳しい解説は、Frederik Kratzert のブログ「Understanding the backward pass through Batch Normalization Layer」まで

6.3.2 Batch Normalization の評価



6.3.2 Batch Normalization の評価



— Batch Normalization

– – Normal(without BatchNorm)

6.4正則化

機械学習の問題では過学習が問題になることが多くあります

過学習とは訓練データだけに適応しすぎてしまい、訓練データに含まれない他のデータにはうまく対応できない状態を言います

6.4.1過学習

過学習が起きる原因

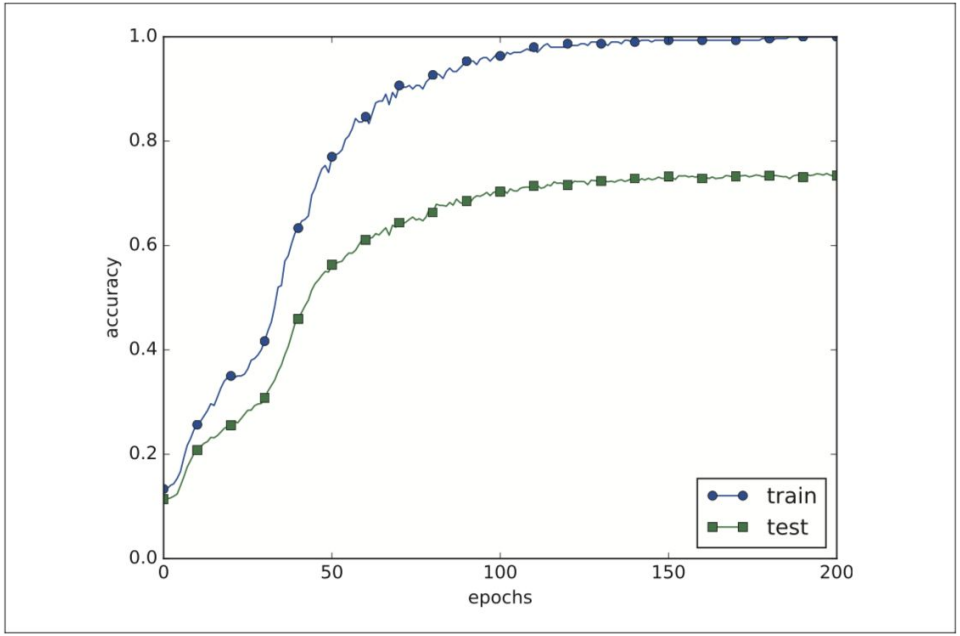
・パラメータを大量に持ち、表現力の高いモデルであること

・訓練データが少ないこと

この2つの要件をわざと満たして、過学習をさせる。

MNISTデータセットの訓練データを本来の 60000個から300個に限定し、7層のネットワークで各層のニューロンの個数は 100個、活性化関数はReLUを使用する

コードは教科書に書いているため省略する



訓練データを用いて計測した認識精度は、100エポックを過ぎたあたりからほとんど 100%

テストデータに対しては 100%の認識精度からは大きな隔たりがある

6.4.2Weight decay

Weight decay(荷重減衰)とは

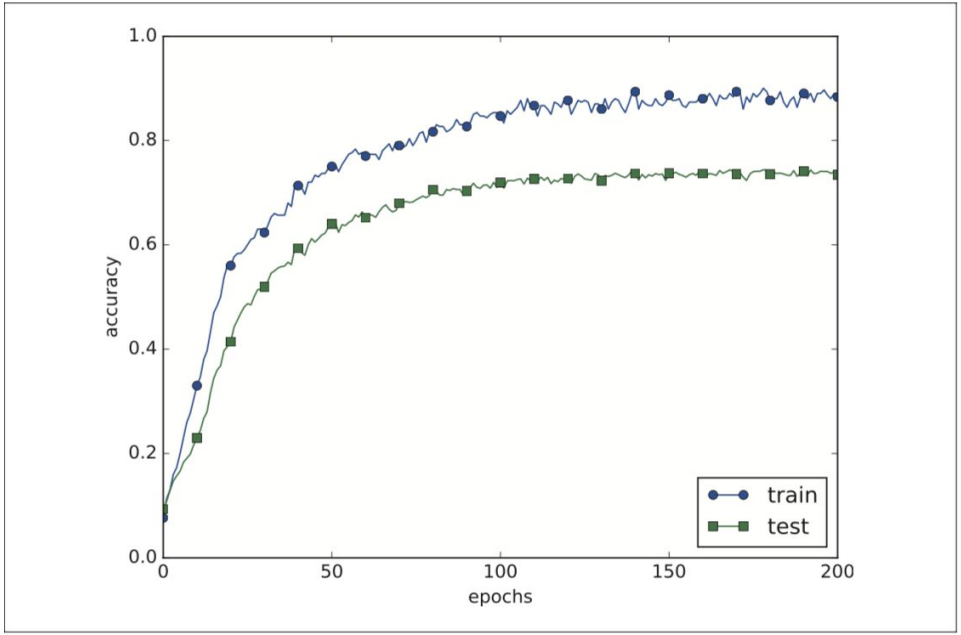
学習の過程において、大きな重みを持つことに対してペナルティを課すことで、過学習を抑制

重みをWとした時、重みの二乗ノルム(L2ノルム)のWeight decayは $\frac{1}{2}\lambda W^2$ になりこれを損失関数に加算

λは正則化の強さをコントロールするハイパーパラメータ
大きく設定すればするほど、大きな重みを取ることに対して強いペナルティを課すことになる

$\frac{1}{2}$ は $\frac{1}{2}\lambda W^2$ の微分の結果をλWにするための調整用の定数

Weight decayはすべての重みに対して、損失関数に $\frac{1}{2}\lambda W^2$ を加算する
そのため重みの勾配を求める計算では、これまでの誤差逆伝播法による結果に、
正則化項の微分λWを加算する



訓練データの認識精度とテストデータの認識精度には隔たりがあるが Weight decay適用前と比べると隔たりが小さくなっている →過学習が抑制されている

訓練データの認識精度が 100%に到達していない点も注目すべき点

6.4.3Dropout

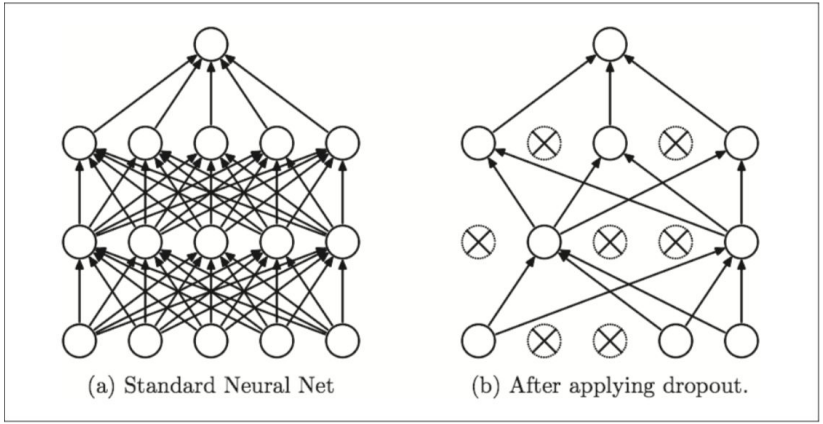
Dropoutとは

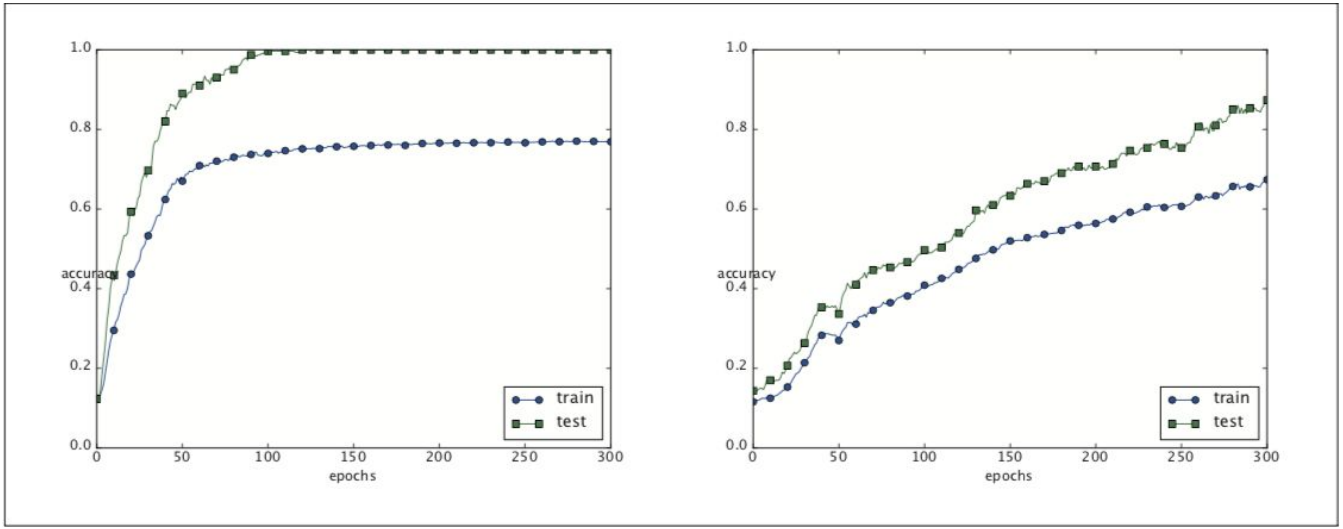
ニューロンをランダムに消去しながら学習する手法

訓練時に隠れ層のニューロンをランダムに選び出し、そのニューロンを消去する
データが流れるたびに消去するニューロンをランダムに選択する

テスト時にはすべてのニューロンの信号を伝達するが、各ニューロンの出力に対して、
訓練時に消去した割合を乗算して出力する

Weight decayではニューラルネットワーク
のモデルが複雑になってくると対応が困難
になってくるためDropoutを用いる





Dropoutなし

Dropoutあり

訓練データとテストデータの認識精度の隔たりが小さくなっている

訓練データが100%に到達することもない

6.5 ハイパーパラメータの検証

・ニューラルネットワークでは、重みやバイアスといったパラメータとは別に、ハイパーパラメータが多く登場する。

・ここで言うハイパーパラメータとは、各層のニューロンの数やパッチサイズ、パラメータの更新の際の学習係数やWeight decayなどのこと。

・ハイパーパラメータを適切な値に設定しなければ、性能の悪いモデルになる。

・ハイパーパラメータの値の決定には一般的に多くの試行錯誤が必要になる。

この節では、できるだけ効率的にハイパーパラメータの値を探索する方法を説明する。

6.5.1 検証データ

・データセットは、訓練データとテストデータの2つに分離して利用する。訓練データで学習を行い、テストデータで汎化性能を評価する。
→それによって訓練データだけに適応していないか(過学習を起こしていないか)、そして、汎化性能はどれくらいかということを評価することができる。

・ハイパーパラメータの値を設定して検証するときの注意点は、テストデータを使ってハイパーパラメータの性能を評価してはいけないということ。
→理由は、テストデータに使ってハイパーパラメータを調整すると、ハイパーパラメータの値はテストデータに対して過学習を起こしてしまうから。

・そのため、ハイパーパラメータを調整する際には、ハイパーパラメータ専用の確認データが必要となる。このデータのことを一般に検証データと呼ぶ。

6.5.1 検証データ

・データセットによっては、あらかじめ訓練データ・検証データ・テストデータの3つに分離されているものがあるが、訓練とテストの2つだけのものや分離が行われてないものもある。その場合、データの分離はユーザーの手によって行う必要がある。MNISTデータセットの場合、検証データを得るための最も簡単な方法は訓練データの中から20%程度を検証データに先に分離することである。

コード例は右のようになる。

ここでは、訓練データの分離の前に、入力データと教師ラベルをシャッフルしている。これはデータセットによってデータの偏りがあるかもしれないためである。
shuffle_datasetという関数は、np.random.shuffleを利用したもので、common/util.pyに実装がある。

```
(x_train, t_train), (x_test, t_test) = load_mnist()

# 訓練データをシャッフル
x_train, t_train = shuffle_dataset(x_train, t_train)

# 検証データの分割
validation_rate = 0.20
validation_num = int(x_train.shape[0] * validation_rate)

x_val = x_train[:validation_num]
t_val = t_train[:validation_num]
x_train = x_train[validation_num:]
t_train = t_train[validation_num:]
```

6.5.2 ハイパーパラメータの最適化

・ハイパーパラメータの最適化を行う上で重要なポイントは、ハイパーパラメータの「良い値」が存在する範囲を徐々に絞り込んでいく、ということである。
最初は大まかに範囲を設定し、その範囲の中からランダムにハイパーパラメータを選び出し(サンプリング)、そのサンプリングした値で認識精度の評価を行う。これを複数回繰り返し行い、認識精度の結果を観察し、その結果からハイパーパラメータの「良い値」の範囲を狭めていく。この作業を行うことによって、適切なハイパーパラメータの範囲を徐々に限定していくことができる。

・ニューラルネットワークのハイパーパラメータの最適化では、グリッドサーチなどの規則的な探索よりもランダムにサンプリングして探索するほうが良い結果になることが報告されている。
→複数あるハイパーパラメータのうち、最終的な認識精度に与える影響度合いがことなるから。

6.5.2 ハイパーパラメータの最適化

・ハイパーパラメータの範囲は、「10の階乗」のスケールでと指定するのが有効である。
(これを「対数スケール(log scale)」で指定するとも表現する。)

・ハイパーパラメータの最適化で注意すべき点は、ディープラーニングの学習には多くの時間(数日や数週間など)が必要になるということ。そのため、ハイパーパラメータの探索では、筋の悪そうなものは早い段階で見切りをつける必要がある。
→学習のためのエポックを小さくして、1回の評価に要する時間を短縮するのが有効

6.5.2 ハイパーパラメータの最適化

ハイパーパラメータの最適化のまとめ

ステップ0

ハイパーパラメータの範囲を設定する

ステップ1

設定されたハイパーパラメータの範囲から、ランダムにサンプリングする。

ステップ2

ステップ1でサンプリングされたハイパーパラメータの値を使用して学習を行い、
検証データで認識精度を評価する(ただし、エポックは小さく設定)。

ステップ3

ステップ1とステップ2をある回数(100回など)繰り返し、それらの認識精度の結果から、
ハイパーパラメータの範囲を狭める。

6.5.3 ハイパーパラメータ最適化の実装

学習係数(learning rate)とWeight decay 係数の2つの探索する問題を対象にハイパーパラメータの最適化を行う。

先ほど述べたように、ハイパーパラメータの検証は対数スケールの範囲からランダムにサンプリングして検証を行う。

```
weight_decay = 10 ** np.random.uniform(-8, -4)
lr = 10 ** np.random.uniform(-6, -2)
```

上記のように10**np.random.uniform()を使ってランダムにサンプリングし、学習を行う。

そして、複数回様々なハイパーパラメータの値で繰り返し学習を行い、筋の良さそうなハイパーパラメータはどこに存在するかを観察する。

実装の詳細は省略して、結果のみを載せます。
ソースコードはch06/hyperparameter_optimization.pyにあります。

6.5.3 ハイパーパラメータ最適化の実装

Weight decay係数を10**-8から10**-4、学習係数を10**-6から10**-2の範囲で実験を行うと、結果は次のスライドの図のようになります。(図は検証データの学習の推移を認識精度が高い順に並べている。)
この図を見ると、「Best-5」ぐらいまでは学習が進んでいることがわかる。
そこで「Best-5」までのハイパーパラメータの値を見ると、下の結果となった。

この結果を見ると学習がうまく進んでいるのは、学習係数が0.001から0.01、Weight decay係数が10**-8から10**-6ぐらいということがわかる。

```
Best-1 (val acc:0.83) | lr:0.0092, weight decay:3.86e-07
Best-2 (val acc:0.78) | lr:0.00956, weight decay:6.04e-07
Best-3 (val acc:0.77) | lr:0.00571, weight decay:1.27e-06
Best-4 (val acc:0.74) | lr:0.00626, weight decay:1.43e-05
Best-5 (val acc:0.73) | lr:0.0052, weight decay:8.97e-06
```

このようにうまくいきそうな範囲を観察し、値の範囲を小さくしていく。その縮小した範囲で同じ作業を繰り返すと、適切なハイパーパラメータの存在範囲を狭められる。
そして、ある段階で最終的なハイパーパラメータの値を一つピックアップすることができる。

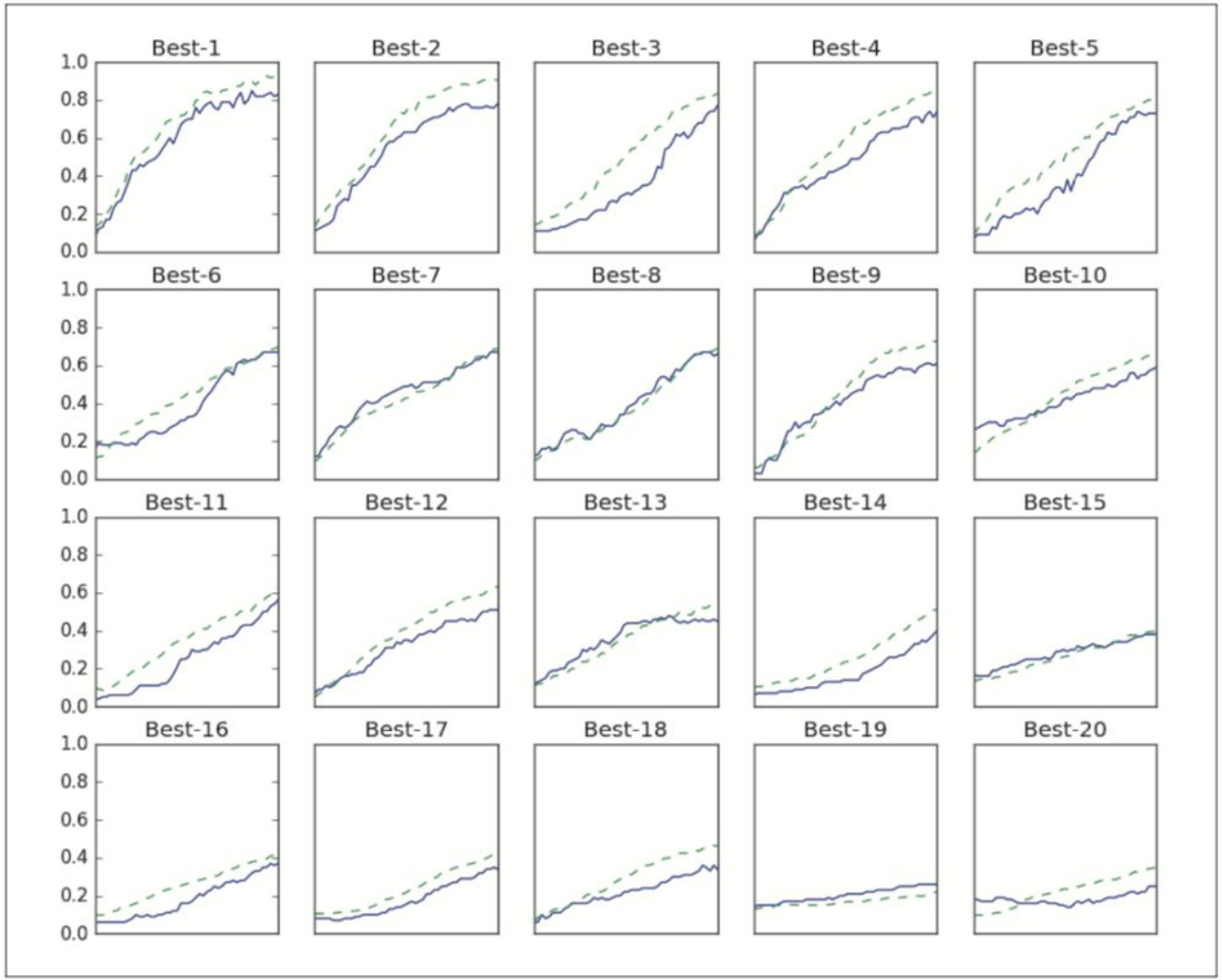


図6-24　実線は検証データの認識精度、点線は訓練データの認識精度

まとめ

- パラメータの更新方法には、SGDの他に有名なものとして、MomentumやAdaGred、Adamなどの手法がある。
- 重みの初期値の与え方は、正しい学習を行う上で非常に重要
- 重みの初期値として、「Xavierの初期値」や「Heの初期値」などが有効
- Batch Normalizationを用いることで、学習を速く進めることができ、また、初期値に対してロバストになる。
- 過学習を抑制するための正則化の技術として、Weight decayやDropoutがある。
- ハイパーパラメータの探索は、良い値が存在する範囲を徐々に絞りながら進めるのが効率の良い方法である。