

教科書輪講

ゼロから作る

Deep Learning

斎藤 康毅

秋山 研

M1 RAN Lu

2018.6.25

6章

学習に関するテクニック

目録

- 6.1 パラメータの更新
- 6.2 重みの初期値
- 6.3 Batch Normalization
- 6.4 正規化
- 6.5 ハイパーコード
- 6.6 まとめ

復習 6.1 パラメータの更新

目的: 最適なパラメータを見つける
(損失関数の値をできるだけ小さくする
パラメータ)

方法

SGD

Momentum

AdaGrad

Adam

ある時効率低い

効率高い

復習 6.2 重みの初期値

初期値 $\neq 0$



どのように設定するか

隠れ層のアクティベーション分布の観察

実験：

- 重みの初期値として標準差1のガウス分布
- ...標準差0.01...
- ...標準差「Xavierの初期値」...
- ...標準差「Heの初期値」... (ReLUの場合)

6.3 Batch Normalization

Batch Normalization

ここまで学んだことは、重みの初期値を適切に設定すれば、各層のアクティベーションの分布は適度な広がりを持ち、学習がスムーズに行えるということでした



“強制的”にアクティベーションの分布を調整します

Batch Normalization

Batch Normalization

2015年に提案された手法

利点：

- 学習を速く進行させることができる
- 初期値にそれほど依存しない
- 過学習を抑制する

Batch Normalization

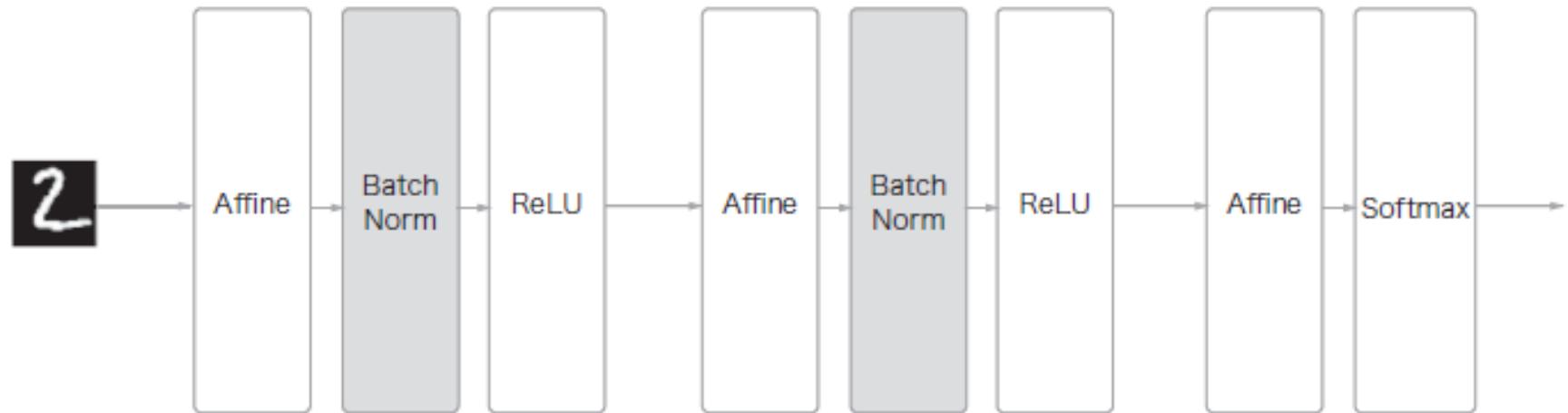


図 6-16 Batch Normalization を使用したニューラルネットワークの例 (Batch Norm レイヤは背景をグレーで描画)

Batch Norm のアイデア:

Batch Norm 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

Batch Normalization

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います

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

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

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

ここでは、ミニバッチとして $B = \{x_1, x_2, \dots, x_m\}$ という m 個の入力データの集合に対して、平均 μ_B 、分散 σ_B^2 を求めます

ε は小さな値(たとえば、 $10e-7$ など)です

Batch Normalization

さらに、Batch Normレイヤは、この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います

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

ここで、 γ と β はパラメータです

最初は $\gamma=1$ 、 $\beta=0$ からスタートして、学習によって適した値に調整されています

順伝播

Batch Normalization

なお、5章で説明した計算グラフを用いれば、
Batch Normは：

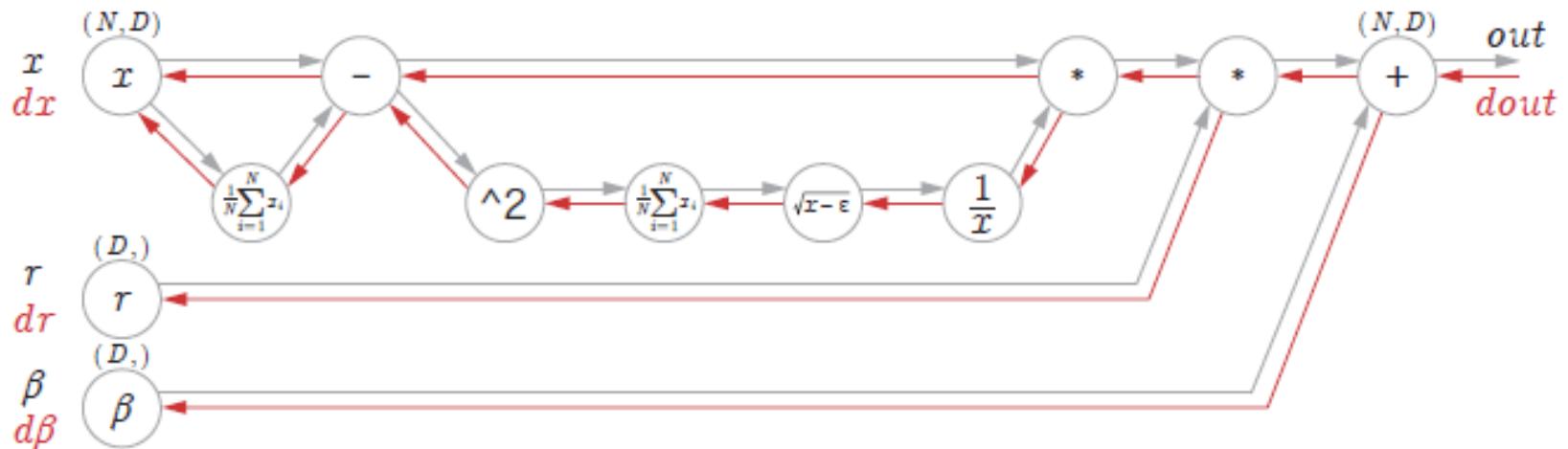


図 6-17 Batch Normalization の計算グラフ（文献 [13] より引用）

Batch Normalization

評価

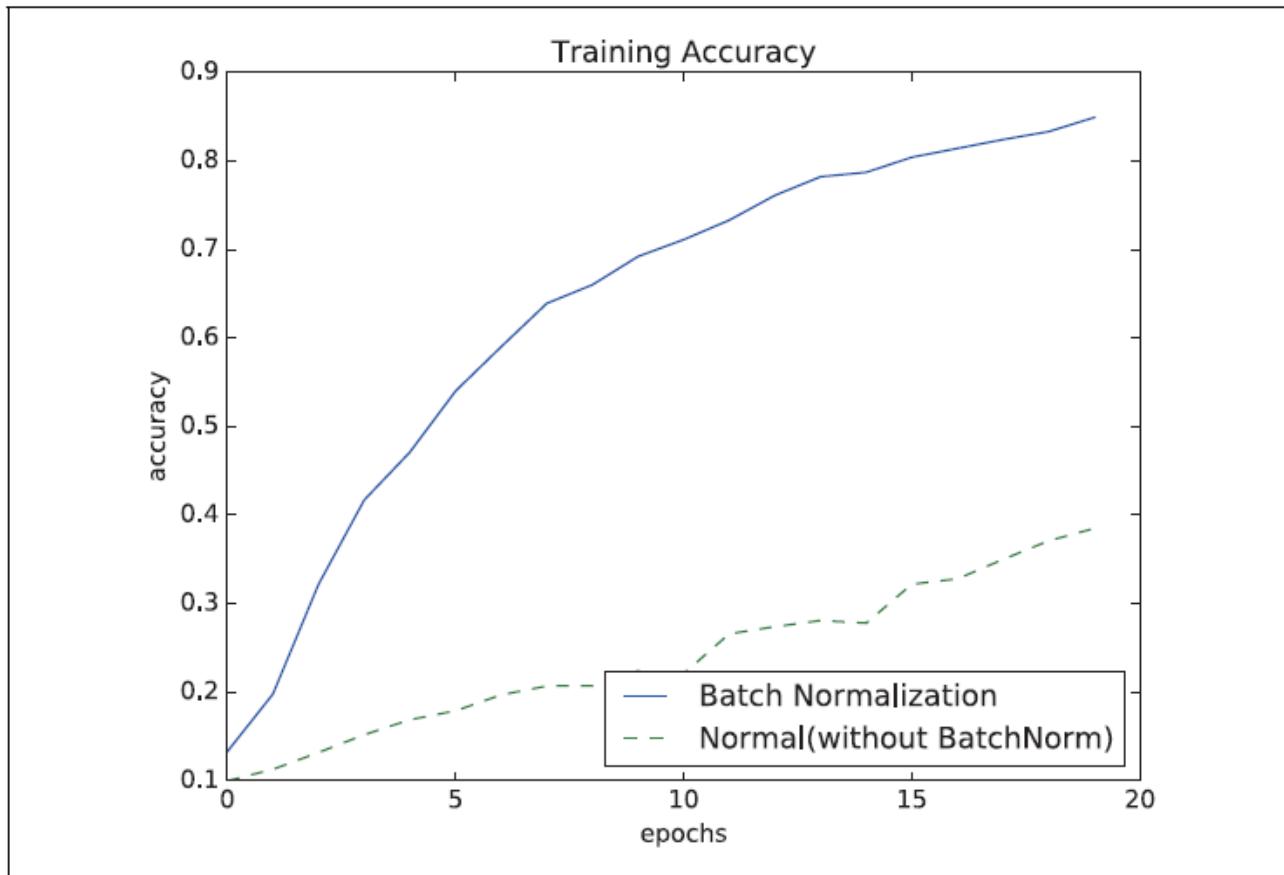


図 6-18 Batch Norm による効果：Batch Norm によって、学習の進み具合が速くなる

Batch Normalization

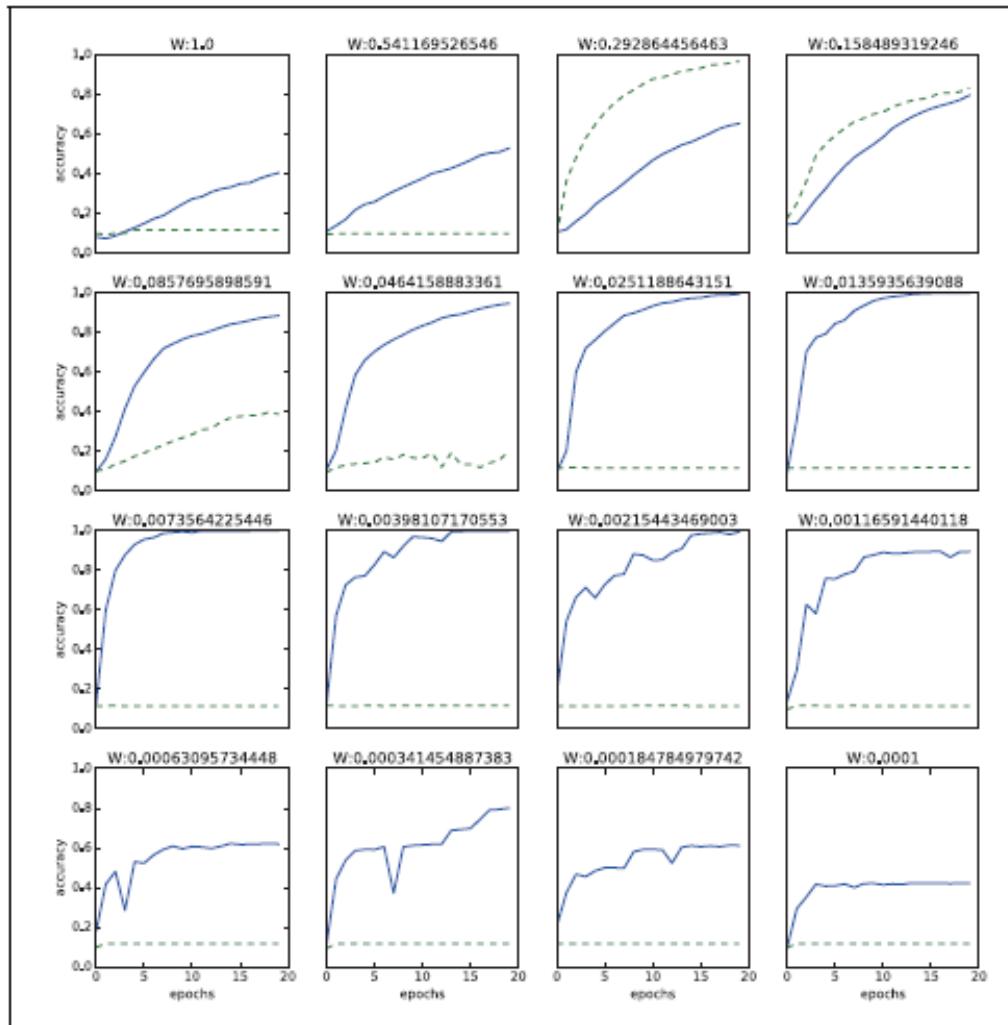


図 6-19 グラフの実線が Batch Norm を使用した場合の結果、点線が Batch Norm を使用しなかった場合の結果：図のタイトルに重みの初期値の標準偏差を表記する

6.4 正規化

正規化

過学習：

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。機械学習で目指すことは、汎化性能です。訓練データには含まれないまだ見ぬデータであっても、正しく識別できるモデルが望されます。複雑で表現力の高いモデルを作ることは可能ですが、その分、過学習を抑制するテクニックが重要になってくるのです。

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とこと

過学習

この2つの用件をわざと満たして、過学習を発生させたいと思います...

そのために：

MNISTデータセットの訓練データ
60, 000個 → 300個

ネットワークの複雑性を高めるため
7層のネットワーク

各層のニューロンの個数
100個

ReLU

過学習

まず、データ読み込みのコードです：

```
(x_train, t_train), (x_test, t_test) = load_mnist(normalize=True)
# 過学習を再現するために、学習データを削減
x_train = x_train[:300]
t_train = t_train[:300]
```

続いて、訓練を行うコードです：

```
network = MultiLayerNet(input_size=784, hidden_size_list=[100, 100, 100,
100, 100], output_size=10)
optimizer = SGD(lr=0.01) # 学習係数 0.01 の SGD でパラメータ更新

max_epochs = 201
train_size = x_train.shape[0]
batch_size = 100

train_loss_list = []
train_acc_list = []
test_acc_list = []
```

過学習

続いて、訓練を行うコードです：

```
iter_per_epoch = max(train_size / batch_size, 1)
epoch_cnt = 0

for i in range(1000000000):
    batch_mask = np.random.choice(train_size, batch_size)
    x_batch = x_train[batch_mask]
    t_batch = t_train[batch_mask]

    grads = network.gradient(x_batch, t_batch)
    optimizer.update(network.params, grads)

    if i % iter_per_epoch == 0:
        train_acc = network.accuracy(x_train, t_train)
        test_acc = network.accuracy(x_test, t_test)
        train_acc_list.append(train_acc)
        test_acc_list.append(test_acc)

    epoch_cnt += 1
    if epoch_cnt >= max_epochs:
        break
```

過学習

`train_acc_list`, `test_acc_list` には、エポック単位——すべての訓練データを見終わった単位——の認識精度が格納されます。それでは、それらのリスト (`train_acc_list`, `test_acc_list`) をグラフとして描画してみます。結果は次の図6-20 のようになります。

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% です。しかし、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ります。このような認識精度の大きな隔たりは、訓練データだけに適応しそぎてしまった結果です。訓練の際に使用しなかった汎用的なデータ（テストデータ）への対応がうまくできていないことが、このグラフから分かります。

過学習

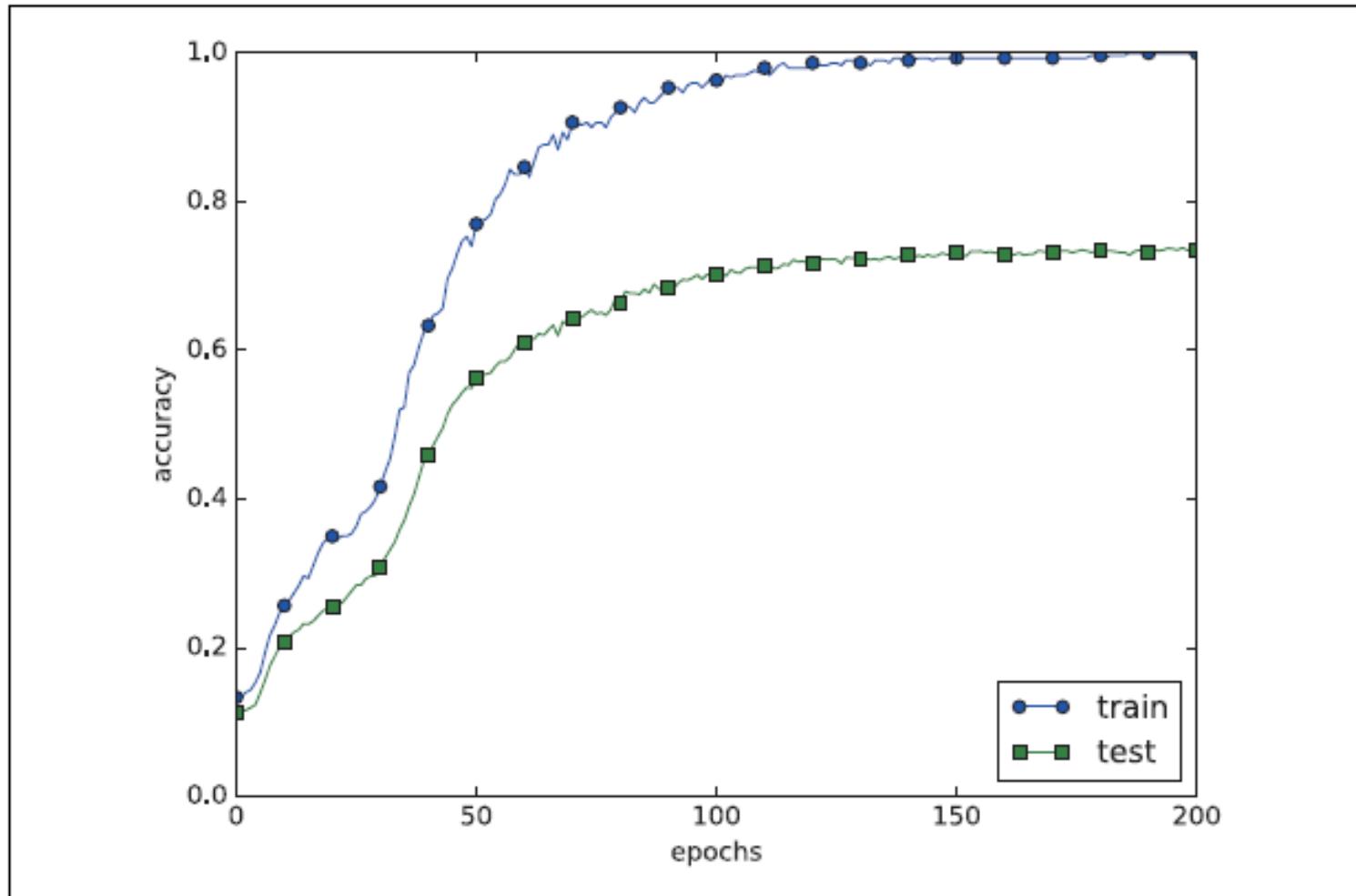


図 6-20 訓練データ (train) とテストデータ (test) の認識精度の推移

過学習抑制のために

方法一: Weight Decay

過学習抑制のために昔からよく用いられる手法に、Weight decay（荷重減衰）という手法があります。これは、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、過学習を抑制しようというものです。そもそも過学習は、重みパラメータが大きな値を取ることによって発生する事が多いのです。

さて、復習になりますが、ニューラルネットワークの学習は、損失関数の値を小さくすることを目的として行われます。このとき、たとえば、重みの2乗ノルム（L2ノルム）を損失関数に加算します。そうすれば、重みが大きくなることを抑えることができそうです。記号で表すと、重みを W とすれば、L2ノルムの Weight decay は、 $\frac{1}{2}\lambda W^2$ になり、この $\frac{1}{2}\lambda W^2$ を損失関数に加算します。ここで、 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です。 λ を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ります。また、 $\frac{1}{2}\lambda W^2$ の先頭の $\frac{1}{2}$ 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です。

Weight decay は、すべての重みに対して、損失関数に $\frac{1}{2}\lambda W^2$ を加算します。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 を加算します。

過学習抑制のために

方法一: Weight Decay

$W=(w_1, w_2, \dots, w_n)$ の重みがあるとすれば

$$\text{L2 norm} = \sqrt{{w_1}^2 + {w_2}^2 + \cdots + {w_n}^2}$$

それでは、実験を行いましょう！

先ほど行った実験に対して、 $\lambda=0.1$ として
Weight Decayを適用します

過学習抑制のために

Weight Decayの実験：

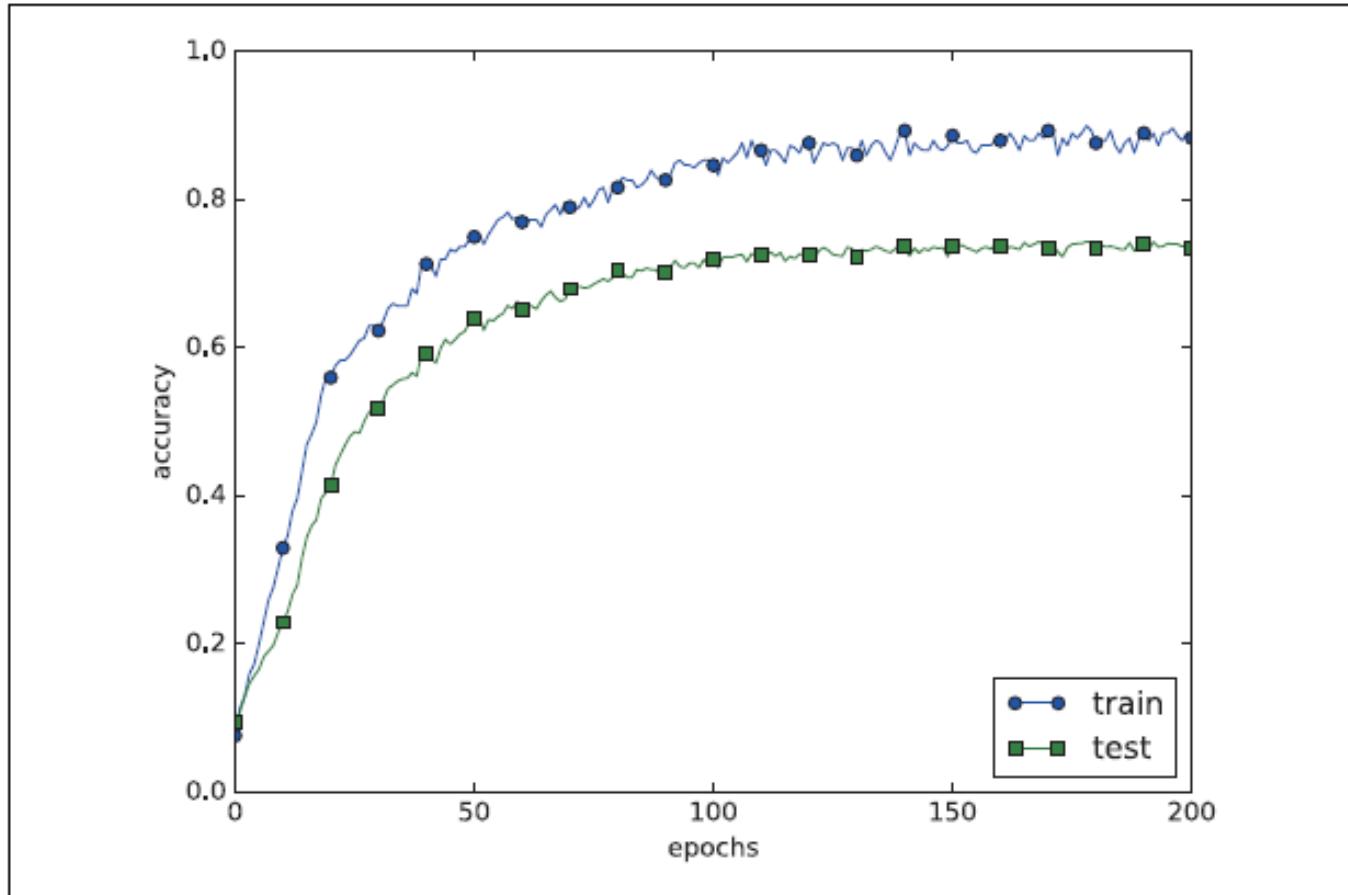


図 6-21 Weight decay を用いた訓練データ (train) とテストデータ (test) の認識精度の推移

過学習抑制のために

方法二: Dropout

過学習を抑制する手法として、損失関数に対して重みの L2 ノルムを加算する Weight decay という手法を説明しました。Weight decay は簡単に実装でき、ある程度過学習を抑制することができます。しかし、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、Weight decay だけでは対応が困難になってきます。そこで、Dropout [14] という手法がよく用いられます。

Dropout は、ニューロンをランダムに消去しながら学習する手法です。訓練時に隠れ層のニューロンをランダムに選び出し、その選び出したニューロンを消去します。消去されたニューロンは、図 6-22 に示すように、信号の伝達が行われなくなります。なお、訓練時には、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します。そして、テスト時には、すべてのニューロンの信号を伝達しますが、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します。

過学習抑制のために

方法二: Dropout

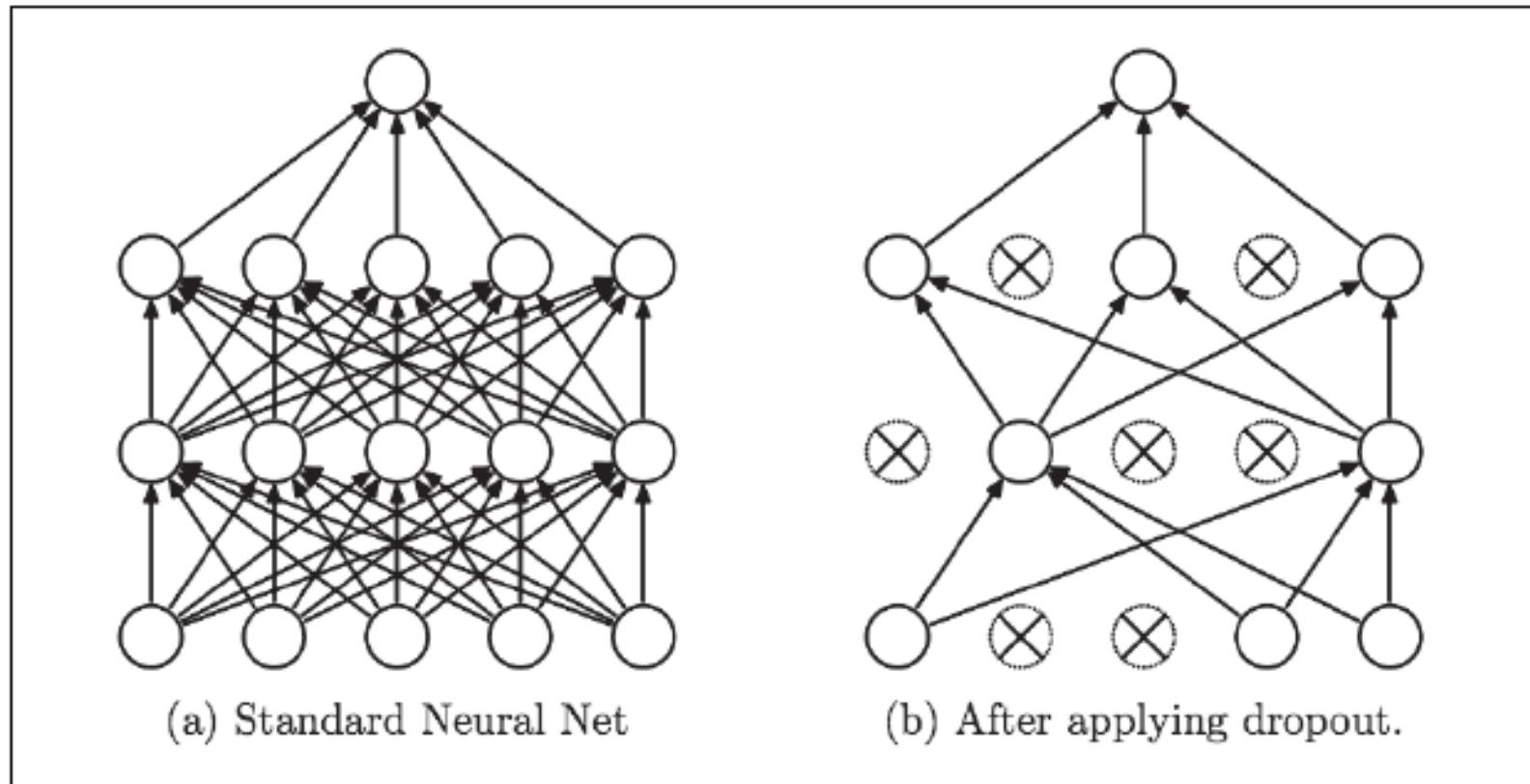


図 6-22 Dropout の概念図（文献 [14] より引用）：左が通常のニューラルネットワーク、右が Dropout を適用したネットワーク。Dropout はランダムにニューロンを選び、そのニューロンを消去することで、その先の信号の伝達をストップする

過学習抑制のために

Dropoutの実装

```
class Dropout:  
    def __init__(self, dropout_ratio=0.5):  
        self.dropout_ratio = dropout_ratio  
        self.mask = None  
  
    def forward(self, x, train_flg=True):  
        if train_flg:  
            self.mask = np.random.rand(*x.shape) > self.dropout_ratio  
            return x * self.mask  
        else:  
            return x * (1.0 - self.dropout_ratio)  
  
    def backward(self, dout):  
        return dout * self.mask
```

過学習抑制のために

Dropoutの実験：

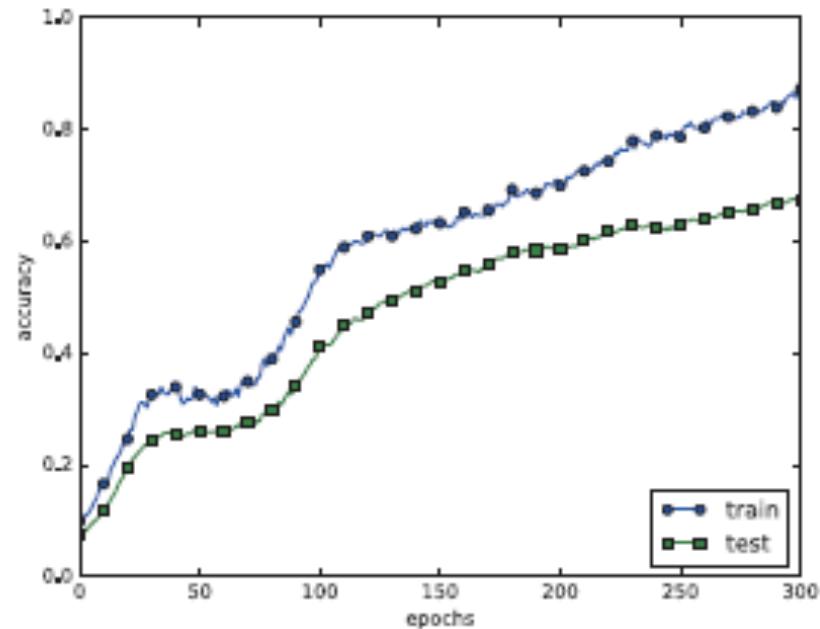
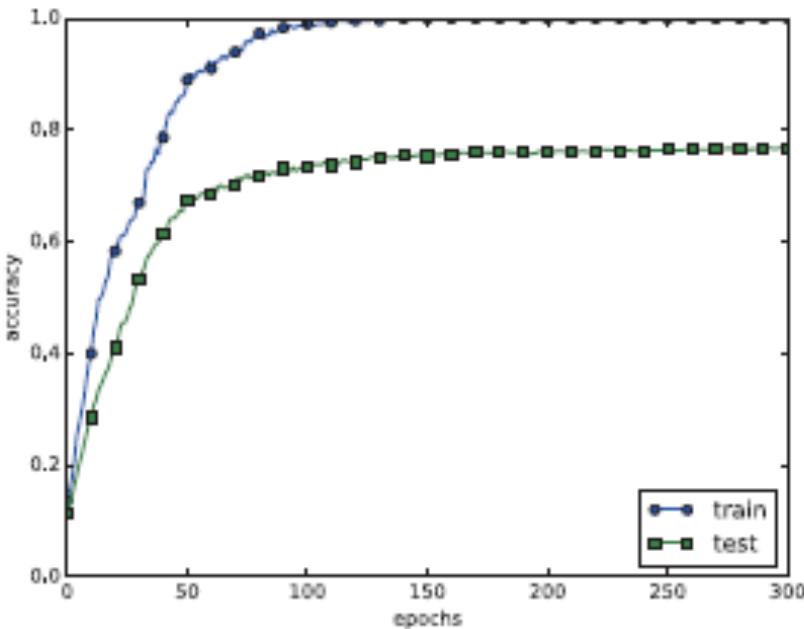


図6-23 左は Dropout なし、右は Dropout あり (dropout_rate=0.15)

6.5 ハイパーパラメータの検証

ハイパーコンフィグの定義

ハイパーコンフィグとは

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、ハイパーコンフィグ (hyper-parameter) が多く登場します。ここで言うハイパーコンフィグとは、たとえば、各層のニューロンの数やバッチサイズ、パラメータの更新の際の学習係数や Weight decay などです。そのようなハイパーコンフィグは、適切な値に設定しなければ、性能の悪いモデルになってしまいます。ハイパーコンフィグの値はとても重要ですが、ハイパーコンフィグの決定には一般に多くの試行錯誤が伴います。ここでは、できるだけ効率的にハイパーコンフィグの値を探索する方法について説明します。

検証データ

これまで使用したデータセット：

訓練データ

学習を行う

テストデータ

汎化性能を評価する

検証データ

ハイパーパラメータを調整する

検証データ

テストデータを使ってハイパーパラメータを評価してはいけません！！

なぜ？

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すことになるからです。そうなると、他のデータには適応できない汎化性能の低いモデルになってしまふかもしれません

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ります。また、データセットによっては、訓練データとテストデータの2つだけに分離されて提供されているものや、そのような分離は行われていないものもあります。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ります。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% 程度を検証データとして先に分離することです。コードで書くと次のようになります。

検証データ

```
(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:]
```

続いて、検証データを使ってハイパー・パラメータの最適化手法を見ていきましょう

ハイパーパラメータの最適化

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。

ハイパーアラメータの最適化

実装：

ハイパーアラメータのランダムサンプリング

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

さて、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、結果は次の図 6-24 のようになります。

ハイパー・パラメータの最適化

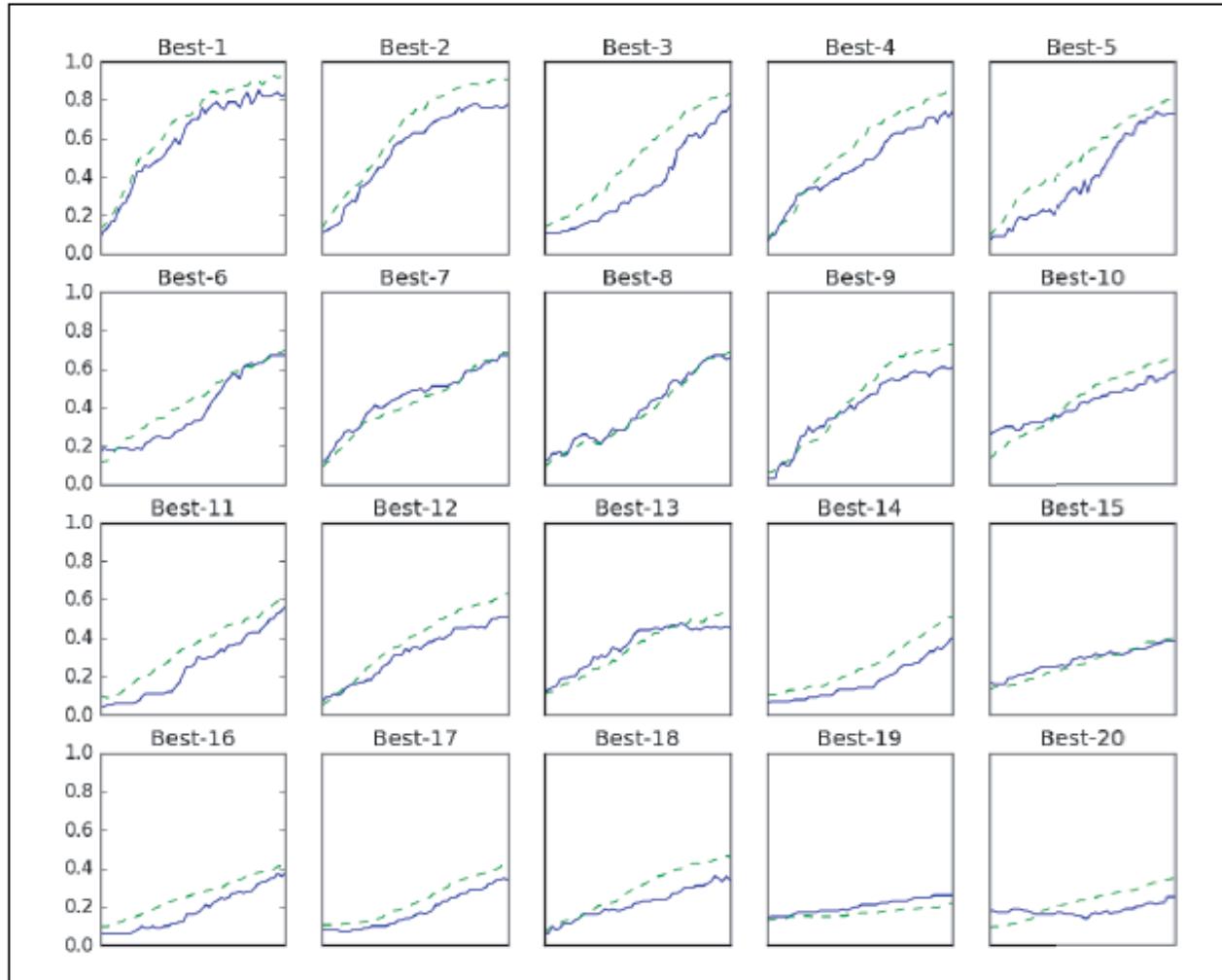


図 6-24 実線は検証データの認識精度、点線は訓練データの認識精度

ハイパーコンフィグレーションの最適化

これを見ると、「Best-5」ぐらいまでは順調に学習が進んでいることが分かります。そこで、「Best-5」までのハイパーコンフィグレーションの値（学習係数と Weight decay 係数）を見てみることにします。結果は、次のようにになります。

```
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
```

この結果を見ると、うまく学習が進んでいるのは、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分かります。このように、うまくいきそうなハイパーコンフィグレーションの範囲を観察し、値の範囲を小さくしていきます。そして、その縮小した範囲で同じ作業を繰り返していくのです。そのようにして、適切なハイパーコンフィグレーションの存在範囲を狭め、ある段階で、最終的なハイパーコンフィグレーションの値をひとつピックアップします。

6.6 まとめ

まとめ

パラメータの更新方法

SGD

Momentum

AdaGrad

Adam

重みの初期値

Xavierの初期値

Heの初期値

まとめ

Batch Normalization

学習を速く進めるができる

初期値に対してロバストになる

過学習を抑制する技術

Weight decay

Dropout

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

Thanks
