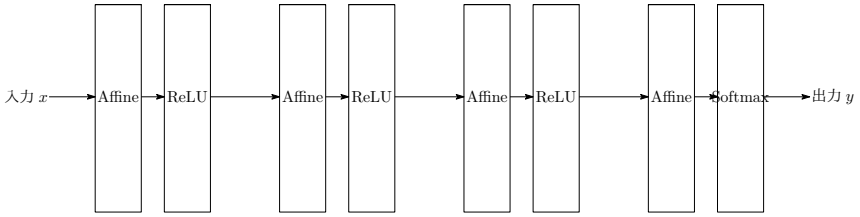
7　畳み込みニューラルネットワーク

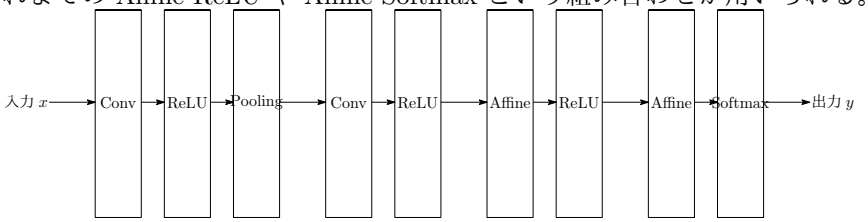
畳み込みニューラルネットワーク (convolutional neural network:CNN) について説明する。これは画像認識や音声認識など，いたるところで利用されている。

7.1　全体の構造

これまで見てきた NN は隣接する層のすべてのニューロン間で結合があった (全結合)。これを Affine レイヤという名前で実装した。この構造を図示すると以下のようになる。つまり Affine レイヤと活性化関数のレイヤを交互に通過するという構造である。



一方 CNN の構造は以下のようになる。新しく Convolution レイヤと Pooling レイヤが加わる。また特に注目する点としては，出力に近い層ではこれまでの Affine-ReLU や Affine-Softmax という組み合わせが用いられる。



7.2　畳み込み層

CNN においては各層を流れるデータは形状のあるデータになるという点で，全結合のネットワークとは異なる。

7.2.1　全結合層の問題点

全結合層の問題点はデータの形状が無視されてしまうことにある。たとえば入力データが画像の場合，縦方向，横方向，チャンネル方向の 3 次元の形状を持っている。しかし全結合層への入力としては 1 次元のデータに変形する必要があった。これは例えば画像データであれば，近くのピクセルの値は近いなどといった重要な情報を無視することになっており，これを活かすことができない。

CNN においては畳み込み層の入出力データを特徴マップという。さらに，畳み込み層の入力データを入力特徴マップ，出力データを出力特徴マップという。

7.2.2 畳み込み演算

畳み込みは入力データに対してフィルター (カーネル) を適用することによって行われる。入力をサイズ (H, W) の行列 A とし，フィルターをサイズ (FH, FW) の行列 B とする。このとき

$$X \circledast F = Y$$

によって表現される畳み込み演算は

$$y_{ij} = \sum_{k=1}^{FW} \sum_{\ell=1}^{FH} x_{(i-1+k)\ (j-1+\ell)} f_{k\ell}$$

のように定義されている。よってこのときに生成される行列 C のサイズは $(H - FH + 1, W - FW + 1)$ となっている。また畳み込み層はこの畳み込み演算に加えて全体にバイアスを加算することで完了する。

7.2.3 パディング

畳み込み層の処理を行う前に，入力データの周囲に固定のデータ (0 など) を埋めることがある (パディング)。これによって出力サイズを調整することができるようになる。畳み込み演算を行うたびにフィルターのサイズに依存した分だけ出力サイズは小さくなる。これを防ぎ空間的なサイズを一定にしたまま次の層へデータを渡すためにパディングを利用する。
パディングの大きさを P をしたとき，出力サイズは $(H - FH + P + 1, W - FW + P + 1)$ となる。

7.2.4 ストライド

これまではフィルターを適用する位置の間隔 (ストライド) は 1 であった。これをより一般化すると以下のような式で与えられる：

$$y_{ij} = \sum_{k=1-P}^{FW+P} \sum_{\ell=1-P}^{FH+P} x_{(Si-1+k)\ (Sj-1+\ell)} f_{k\ell}$$

ただし添え字が 1 から H などの定義された範囲を超えたときはパディングによって埋め込まれた値を返すものとする。このとき出力サイズは $(\frac{H + 2P - FH}{S} + 1, \frac{W + 2P - FW}{S} + 1)$ となる。

7.2.5　3 次元データの畳み込み演算

画像データの場合，縦と横に加えてチャンネル方向も考慮する必要がある。チャンネル方向についても同様に足し算を実行すればよい。

$$y_{ij} = \sum_{k=1-P}^{FW+P} \sum_{\ell=1-P}^{FH+P} \sum_{m=1}^{C} x_{(Si-1+k)\ (Sj-1+\ell)\ m}\ \ f_{k\ell m}$$

このとき注意すべき点は，入力データとフィルターのチャンネル数は同じ値にするということである (フィルターのサイズは自由に設定できるのになぜ?)。

7.2.6　ブロックで考える

チャンネル数が同じであるため，このままでは出力データはチャンネル方向のサイズは 1 である。畳み込み演算の出力をチャンネル方向にも複数持たせるには複数のフィルターを用いればよい。FN 個のフィルターを用いると考えると，

$$y_{ijk} = \sum_{\ell=1-P}^{FW+P} \sum_{m=1-P}^{FH+P} \sum_{n=1}^{C} x_{(Si-1+\ell)\ (Sj-1+m)\ n}\ \ f_{\ell mnk} \quad (k = 1, 2, \cdots, FN)$$

この式を見れば明らかなように，フィルターの重みデータは 4 次元のデータとして (output_channel,input_channnel,height,width) という順に表すことができる。
さらにバイアスの加算処理も追加すると，以下のようになる。FN 個の成分を持ったベクトルを考えて，

$$z_{ijk} = y_{ijk} + b_k$$

がバイアスの加算処理である。

7.2.7　バッチ処理

N 個のバッチサイズのデータに対して処理を行うものとする。このときデータの番号という新しい次元を追加した 4 次元のデータが流れることになる：

$$z^d_{ijk} = \sum_{\ell=1-P}^{FW+P} \sum_{m=1-P}^{FH+P} \sum_{n=1}^{C} x^d_{(Si-1+\ell)\ (Sj-1+m)\ n}\ \ f_{\ell mnk} + b_k$$

ここで $1 \leq d \leq N$ がデータの番号を走るラベルである。

7.3 プーリング層

プーリングによって，縦・横方向の空間を小さくする。$n \times n$ の Max プーリングをストライド n で行うとは，畳み込み演算と同じようにサイズ $n \times n$ のウィンドウを移動させつつ，指定された範囲から最大値を取り出して行列を作るということである。基本的にはストライドの値とウィンドウサイズを同じ値にとる。
Max プーリングのほかに，Average プーリングなどがある。画像認識の分野においては Max プーリングが主に用いられる。

7.3.1 プーリング層の特徴

プーリング層には①学習するパラメータがない②チャンネル数が変化しない③微小な変化に対してロバストという特徴がある。

7.4 Convolution/Pooling レイヤの実装

Convolution/Pooling レイヤの実装は複雑になりそうではあるが，im2col を利用することによって簡単に実装することができる。

7.4.1 4 次元配列

順にデータの番号，チャンネル，縦，横のデータが 4 次元配列に格納されている。

```
>>> x.shape
(10, 1, 28, 28)
>>> x[0].shape
(1, 28, 28)
```

7.4.2 im2col による展開

im2col はフィルターにとって都合のいいように入力データを展開する。まずフィルターを適用する範囲を横方向に 1 列に展開する。これをフィルターを適用するすべての場所に対して行い，並べることで 2 次元の行列を生成する。次にフィルターを 1 列に展開し，並べることで 2 次元の行列を生成する。これらの積をとることによって, 出力データを 2 次元の行列として得ることができる。行ごとに入力データの縦横の位置が与えられており (位置の自由度)，列ごとにどのフィルターを通ったかが与えられている (チャンネルの自由度)。これを適切に成形することによって出力データが得られる。

7.4.3 Convolution レイヤの実装

im2col(input_data,filter_h,filter_w,stride=1,pad=0) によって 2 次元の行列が生成される。まず各行について考える。ある位置にフィルターが固定されたときに，filter_h,filter_w とチャンネル数の分だけの要素が横一列に並べられることになる。
次に各列について考える。上のようにして並べられたベクトルが，選択される位置とデータ数の分だけ積み重なることになる。よって以下のような結果が得られる：

```
>>> import sys,os
>>> sys.path.append(os.pardir)
>>> from common.util import im2col
>>> x1 = np.random.rand(1,3,7,7)
>>> col1 = im2col(x1,5,5,stride=1,pad=0)
>>> col1.shape
(9, 75)
>>> x2 = np.random.rand(10,3,7,7)
>>> col2 = im2col(x2,5,5,stride=1,pad=0)
>>> col2.shape
(90, 75)
```

行列の形状が決まる過程は以下のような式で与えられる：

$$
\begin{aligned}
75 &= 5 \times 5 \times 3 \\
9 &= 1 \times (\frac{7-5}{1} + 1) \times (\frac{7-5}{1} + 1) \\
90 &= 10 \times (\frac{7-5}{1} + 1) \times (\frac{7-5}{1} + 1)
\end{aligned}
$$

最後に行列を適切に加工することによって出力データが得られることに注意。まずデータ数で適当に行列の行を切り分けて，次にフィルターの高さと横幅で切り分ける。あとはチャンネルで切り分ければよい。また実装上は軸のラベルを変更する必要があることにも注意。
逆伝播の時は im2col の逆の処理 col2im が必要になることにも注意。それを除けば，Affine レイヤと同様に実装することができるようになる。

7.4.4 Pooling レイヤの実装

Pooling レイヤも Convolution レイヤと同様に im2col を使って展開する。ただしプーリングの場合はチャンネル方向には独立であることに注意。つまりこれまではチャンネル方向も横に続けて展開していたが，チャンネルが異なるものは下に積み重ねる形で展開することになる (データについても先ほどと同様下に積み重ねる形で展開する)。
その後 1 番目の軸 (0 から始めることに注意) について最大値を求めるとこれ

が max プーリングになる。1 番目の軸は列のことであり，列ラベルについて走ったときの最大値を求めることになる。
最後に成形しなおせば処理が完了する。逆伝播も同様にして処理できる。

7.5 CNN の実装

ここで実装するネットワークの構成は Convolution-ReLU-Pooling-Affine-ReLU-Affine-Softmax という流れであり，これを SimpleConvNet という名前のクラスで実装する。

7.6 パラメータの更新

これまでに扱ってきた確率的勾配降下法 (SGD) は単純な方法であり，問題によっては SGD よりも効率の良い手法が存在する。以下では SGD の欠点を指摘し，別の最適化手法を説明する。

7.6.2 SGD

SGD は数式では以下のように表現することができる；

$$\boldsymbol{W} \leftarrow \boldsymbol{W} - \eta \frac{\partial L}{\partial \boldsymbol{W}}$$

これを SGD というクラスで実装すると以下のようになる：

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

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

パラメータの更新に使うクラスは grads を引数とすることに注意。key はディクショナリ変数 params と grads の引数を与える。実際にニューラルネットワークのパラメータの更新を行う時には，まず optimizer = SGD() のようにして最適化を行う手法を選択する。次に optimizer.update(params,grads) のようにしてパラメータの更新を行う。これにより他の最適化手法を選ぶときも optimizer = Momentum() と変更するだけでよい。

7.6.3 SGD の欠点

次の関数の最小値を計算する問題を考える：

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

SGD によってこの最適化を行うとまっすぐ最小値に向かわず効率的でないということがわかる。これは勾配の方向が最小値の方向を向いていないためであり，より適切な方法を選ぶ必要がある。

```
optimizer = SGD()
x = np.array([-7.0,2.0])
T = 100
trajectory = []
for t in range(T):
        g = numerical_gradient(f,x)
        trajecory.append(x)
        optimizer.update_array(x,g)
```

ここで SGD.update の引数はディクショナリ型であったため，配列を引数とする SGD.update_array を適切に定義したことに注意。
図 1 は上の計算結果をプロットしたものである。確かに振動しつつ最小値に

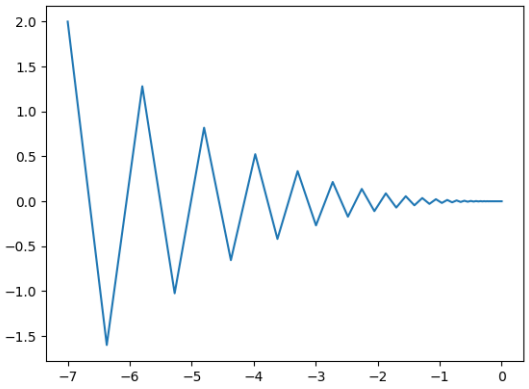


図 1: SGD による最適化

向かっており，非効率的な動きになっている。

7.6.4 Momentum

Momentum は次のような式で表すことができる：

$$\begin{aligned} \boldsymbol{v} &\leftarrow \alpha\boldsymbol{v} - \eta\frac{\partial L}{\partial \boldsymbol{W}} \\ \boldsymbol{W} &\leftarrow \boldsymbol{W} + \boldsymbol{v} \end{aligned}$$

SGD は $\alpha = 0.0$ に対応する。α はもとの速度を保とうとする効果 (慣性) を与えている。また $\alpha < 1$ ならばこの効果は指数的に減衰する。

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

        def update_dict(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]

        def update_array(self,params,grads):
                if self.v is None:
                        self.v = np.zeros_like(params)
                for i in range(params.size):
                        self.v[i] = self.momentum*self.v[i]-
                            self.lr*grads[i]
                        params[i] += self.v[i]
```

実装すると以上のようになる。self.v が初めに定義されていないことに注意。図 2 は上の計算結果をプロットしたものである。慣性を導入したことで滑らかに移動していることが確認できる。しかし今回は慣性が強いため最小値付近までたどり着いても収束するのに長い時間がかかっている。

7.6.5 AdaGrad

学習率 η の大きさを適切に決めることが正しい学習には不可欠である。初めは大きくパラメータを動かし，次第にパラメータの動きを小さくさせるた

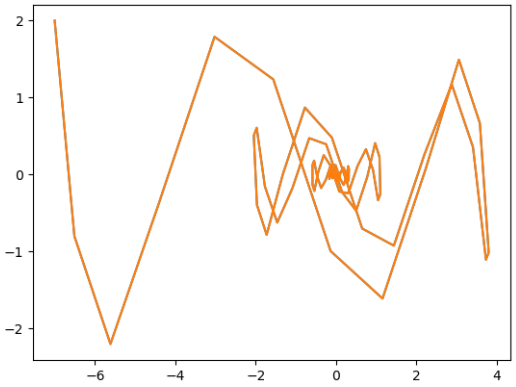


図 2: Momentum による最適化

め，学習率を減衰させるという手法がよく用いられる。AdaGrad は一つ一つのパラメータに対して個別に学習率を減衰させる。数式では以下のようになる：

$$\begin{aligned} \boldsymbol{h} &\leftarrow \alpha \boldsymbol{h} + \eta \frac{\partial L}{\partial \boldsymbol{W}} \cdot \frac{\partial L}{\partial \boldsymbol{W}} \\ \boldsymbol{W} &\leftarrow \boldsymbol{W} - \eta \frac{1}{\sqrt{\boldsymbol{h}}} \frac{\partial L}{\partial \boldsymbol{W}} \end{aligned}$$

まず $\cdot$ はベクトルを要素ごとに掛けてベクトルを生成する演算子である。よって $\boldsymbol{h}$ はこれまでに経験した勾配の値をパラメータごとに二乗和として保存する。これによって，大きく更新されたパラメータの学習率は小さくなり，個別の学習率の減衰を実現することができる。

AdaGrad を用いるとパラメータの更新量が 0 になってしまうことに注意。これを防ぐため，過去の勾配に指数的減衰を与える RMSProp という手法がある。これを指数移動平均という。

AdaGrad は以下のように実装することができる：

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

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

```
11
12                 for key in params.keys():
13                         self.h[key] += grads[key]*grads[key]
14                         params[key] -= self.lr*grads[key]/(np
                               .sqrt(self.h[key])+1e-7)
15
16         def update_array(self,params,grads):
17                 if self.h is None:
18                         self.h = np.zeros_like(params)
19                 self.h += grads*grads
20                 params -= self.lr*grads/(np.sqrt(self.h)+1e
                       -7)
```

これを用いて最適化した結果が図 3 である。初めは y 軸方向に大きな勾配を

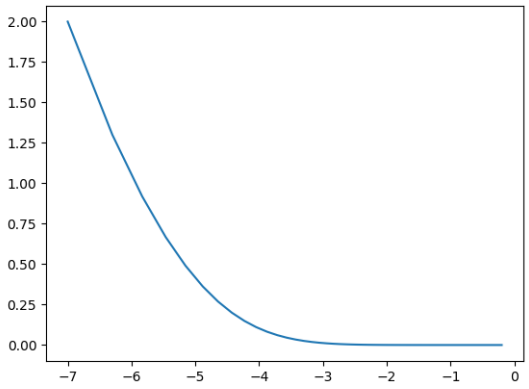


図 3: Momentum による最適化

持っているが，それが抑制されて振動が軽減されている。

7.6.6 Adam

Adam は Momentum と AdaGrad を融合させたような手法である。先ほどと同様に最適化を行うと，Momentum のように振動しつつ中心に向かうが，その振動は Momentum よりも軽減されている。これは学習率が適応的に更新されているためである。
なお，ハイパーパラメータのバイアス補正が行われているということも Adam の特徴である。

7.6.7　どの更新手法を用いるか?

$f(x, y) = \frac{1}{20}x^2 + y^2$ を最小化するという目的に限れば AdaGrad が最も効率的であるといえる。しかし考えたいタスクによって用いるべき最適化手法は当然異なり，それぞれの特徴を踏まえたうえで適切に選ぶ必要がある。

7.6.8　MNIST データセットによる更新手法の比較

手書き数字認識を対象に，これまでの 4 つの手法を比較する。ネットワークの構成は 4 章と全く同じである。このタスクに関しては SGD よりも他の

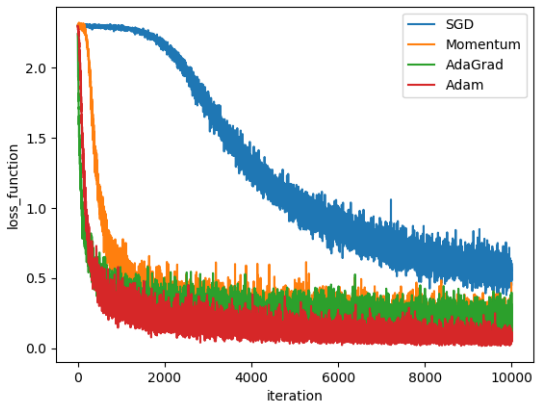


図 4: MNIST データセットの学習の比較。

手法が早く学習できていることがわかる。SGD よりも他の手法の方が早く学習でき，時には最終的な認識性能も高くなるという傾向がある。AdaGrad が最も効率的に学習しているようだが，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ということに注意。

7.7　重みの初期値

重みの初期値がニューラルネットワークの学習の成否を分けることが実際によくある。またニューラルネットワークの学習を速やかに進めるためにも初期値の設定は重要である。

7.7.1　重みの初期値を 0 にすることの問題点

過学習が生じているときには重みの値が大きくなっていることが多い。よって重みパラメータの値が小さくなるように学習を行えば過学習を防ぐことが

できると期待できる。この手法を Weight decay という。
重みの値が小さくなるように，初期値をすべて 0 にすることも考えられるが，これでは重みの更新が生じず，したがって学習が進まなくなってしまう。学習を行う上では，重みの対称的な構造を崩すことが重要になる。

7.7.2 隠れ層のアクティベーション分布

活性化関数の出力の分布をアクティベーション分布という。以下のヒストグラムは，5 つの隠れ層を持つニューラルネットワークに対してランダムに生成した入力データを流し込んだ時のアクティベーション分布を表したものである。それぞれ重みパラメータを標準偏差 0.01 から 1.0 としたときのアクティベーション分布を示している。 標準偏差 1.0 のときのアクティベーション分布を見ると，多くが 0 と 1 に偏っている。このとき損失関数の勾配はシグモイド関数の振る舞いから考えて小さくなると考えることができる (勾配消失)。

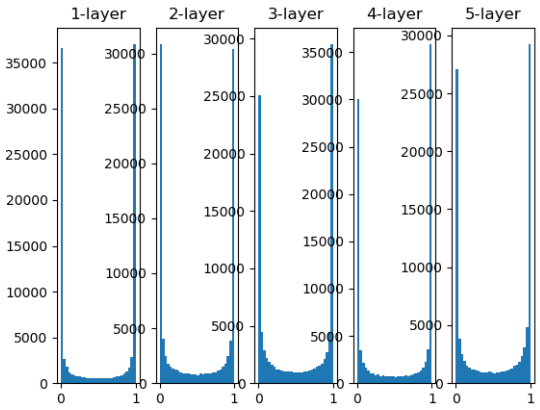


図 5: 標準偏差 1.0 のガウス分布を用いたときのアクティベーション分布

標準偏差 0.01 のときのアクティベーション分布はほとんどが 0.5 に近い値をとっている。これは勾配消失の問題は生じないが，ほとんどのニューロンが同じ値を示すのならばそれらは 1 つのニューロンで表すことができる。つまり表現力の問題が生じる。
標準偏差 0.1 のときのアクティベーション分布は適度な広がりを保っている。これによりシグモイド関数の表現力が妨げられることもなく，うまく学習が進む。これは前層のニューロン数を n としたとき，初期値の分散を $1/\sqrt{n}$ 程度にとる Xavier の初期値を用いている。
なお，上位層の分布の形状がいびつになったのは tanh 関数ではなく sigmoid

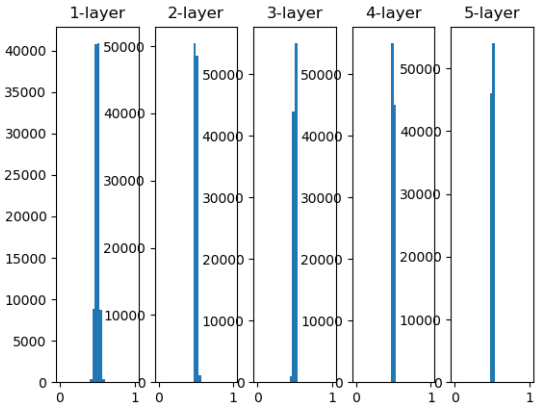


図 6: 標準偏差 0.01 のガウス分布を用いたときのアクティベーション分布

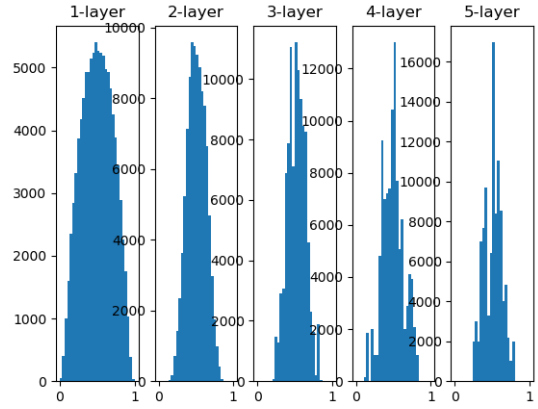


図 7: 標準偏差 0.1 のガウス分布を用いたときのアクティベーション分布

関数を用いたためである。活性化関数としては原点対象であることが望ましいとされている。

7.7.3 ReLU の場合の重み初期値

Xavier の初期値は活性化関数が線形であることを前提に導かれている。sigmoid 関数や tanh 関数は左右対称で中央付近が線形関数とみなせるため Xavier の初期値が適している。
一方 ReLU を用いるときには ReLU に特化した He の初期値を用いることが推奨されている。これは前層のノード数が n 個であるとき，$\sqrt{\frac{2}{n}}$ を標準偏差とするガウス分布を用いる初期値である。

7.7.4 MNIST データセットによる重み初期値の比較

MNIST データセットに対し重みの初期値の与え方を変えたときに，学習の進み方がどのように変わるかを調べたところ，ReLU を用いたときには He の初期値が最も効率的に学習が進む。

7.8 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分布を調整する手法が Batch Normalization である。

7.8.1 Batch Normalization のアルゴリズム

Batch Norm には次の利点がある：

1. 学習を早く進行させることができる
2. 初期値にそれほど依存しない
3. 過学習を抑制する

これは各層において Batch Norm レイヤを挿入することで，データ分布の正規化を行う。具体的な計算は以下のようになる：

$$
\begin{aligned}
\mu_{\mathrm{B}} &\leftarrow \frac{1}{m}\sum_{i=1}^{m} x_i \\
\sigma_{\mathrm{B}}^2 &\leftarrow \frac{1}{m}\sum_{i=1}^{m}(x_i-\mu_{\mathrm{B}})^2 \\
\hat{x}_i &\leftarrow \frac{x_i-\mu_{\mathrm{B}}}{\sqrt{\sigma_{\mathrm{B}}^2+\epsilon}} \\
y_i &\leftarrow \gamma\hat{x}_i+\beta
\end{aligned}
$$

ここで ϵ は発散を防ぐための小さなパラメータであり，また γ と β は学習によって適した値に調整されるパラメータである。

7.8.2 Batch Normalization の逆伝播の導出

https://kratzert.github.io/2016/02/12/understanding-the-gradient-flow-through-the-batch-normalization-layer.html に基づいて Batch Norm の逆伝播を導出する。

7.8.3 Batch Normalization の評価

Batch Norm を利用した方が学習の進みが早くなる。また重みの初期値を適当に設定してもうまく学習が進むことが多い。

7.9 正則化

過学習を抑制するための手段として正則化というものがある。

7.9.1 過学習

過学習が生じる原因として，主に次の 2 つを上げることができる：

1. パラメータを大量に持ち，表現力が高いモデルであること
2. 訓練データが少ないこと

がある。実際に MNIST のデータに対して訓練データを 300 個にし，7 層のニューラルネットワークを用いたところ，訓練データに対する認識制度は 100% であったが，テストデータに対しては 100% からは大きな隔たりがある。

7.9.2 Weight decay

Weight decay は大きな重みパラメータを持つことにペナルティーを科すことで過学習を抑制するという手法である。つまり損失関数に対し，L2 ノルムに比例する $\frac{1}{2}\lambda \boldsymbol{W}^2$ という項を加えることでこれを実現できる。よって重みの勾配を計算するときには誤差逆伝播法による結果に加えて $\lambda \boldsymbol{W}$ を考慮する必要がある。ここで L1 ノルム，L2 ノルム，L∞ ノルムのいずれを用いてもよいことに注意。それぞれの定義は以下のようになる：

$$
\begin{aligned}
|\boldsymbol{W}|_1 &= |w_1| + |w_2| + \cdots + |w_n| \\
|\boldsymbol{W}|_2 &= \sqrt{w_1^2 + w_2^2 + \cdots + w_n^2} \\
|\boldsymbol{W}|_\infty &= \max(w_1, w_2, \cdots, w_n)
\end{aligned}
$$

これを利用して先ほどと同様の学習を行うと，テストデータに対する認識制度が向上する。また訓練データに対する認識制度が 100% に達しないということにも注意。

7.9.3 Dropout

Weight decay は単純な方法で，ある程度過学習を抑制することができる。しかしニューラルネットワークのモデルが複雑になると Dropout という手法が良く用いられるようになる。
Dropout はニューロンをランダムに消去しながら学習するという手法である。実際にこれを適用して学習をすることで，過学習が抑制される。

7.10 ハイパーパラメータの検証

各層のニューロン数やバッチサイズ，学習率や Weight decai(の λ) などがハイパーパラメータと呼ばれる。これは学習によって変化することのない値であり，うまく決定する必要がある。ここでは効率的にハイパーパラメータの値を探索する方法について説明する。

7.10.1 検証データ

ハイパーパラメータの性能を評価するときにはテストデータを利用してはならない。なぜならばハイパーパラメータの値がテストデータに対して過学習を起こすことになるためである。よってハイパーパラメータ専用の確認データ (検証データ) が必要になる。訓練データとテストデータしかない場合は，訓練データの中から 20% 程度を先に分離して検証データとすればよい。

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

>>> validation_rate = 0.20
>>> calidation_num = int(x_train.shape[0]*validation_rate)
>>> 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:]
```

ここで shuffle_dataset() という関数を利用している。この定義は以下のようになっている：

```
def shuffle_dataset(x, t):
    permutation = np.random.permutation(x.shape[0])
    x = x[permutation,:] if x.ndim == 2
        else x[permutation,:,:,:]
    t = t[permutation]

    return x, t
```

x の次元数によって場合分けをしていること，スライシングの表記がいまいちよくわかっていない。

7.10.2 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最適値が存在する範囲を徐々に絞り込んでいくということにある。そのときにはグッリドサーチなどの規則的な探索よりも，ランダムに (対数スケールで) サンプリングして得られた認識精度を観察して最適値を絞り込むほうが効率が良い。これは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ハイパーパラメータごとに異なるためである。以上の手順をまとめると以下のようになる：

1. ハイパーパラメータの範囲を設定する
2. 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
3. 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 (ただしエポックは小さく設定)
4. 1. と 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

より洗練された手法としてはベイズ最適化がある。これを用いるとより厳密に効率よく最適化を行うことができる。

7.10.3 ハイパーパラメータ最適化の実装

weight_decay $=$ 10**np.random.uniform(-8,-4) のような表記を用いると，対数スケールでランダムにサンプリングをすることができる。このように得られた 20 程度のサンプルのうち，認識精度の高いものについてハイパーパラメータの範囲を調べる。ここからより範囲を狭めることができる。

7.11　まとめ

パラメータの更新方法や，重みの初期値の与え方，Batch Normalization や Dropout など，現代のニューラルネットにとっては書くことのできない技術になっている。またこれらは最先端の DL においても頻繁に利用されている。