

教科書輪講

ゼロから作る

Deep Learning

斎藤 康毅

---

秋山 研

M1 RAN Lu

2018.6.18

# 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 ハイパーパラメータの検証

6.6 まとめ

## 6.1 パラメータの更新

---

6.1 Update Parameters

# 6.1 パラメータの更新

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です



最適なパラメータ

最適化

( optimization)

# 6.1 パラメータの更新

最適化  
( optimization)

(難しそう)

↓このため

方法:

- 確率的勾配降下法  
(stochastic gradient descent, SGD)
- Momentum
- AdaGrad
- Adam

# Before SGD

## 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？

# Before SGD

## 冒険家の話

最適なパラメータの探索 = 冒険家と同じ暗闇の世界



解決方法

SGDの戦略 = 地面の傾斜

# SGD

---

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

$\mathbf{W}$ : 更新する重みパラメータ

$\frac{\partial L}{\partial \mathbf{W}}$ :  $\mathbf{W}$ に関する損失関数の勾配

$\eta$ : 学習係数 (learning rate)  
(0.01や0.001)

# SGDの実装

```
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]
```

`update(params, grads)`というメソッドを定義しますが、SGDでは、このメソッドが繰り返し呼ばれることになります。引数のparamsとgradsは、(これまでのニューラルネットワークの実装と同じく)ディクショナリ変数です。`params['W1']`、`grads['W1']`などのように、それぞれに重みパラメータと勾配が格納せっています。

# SGDの実装

このSGDというクラスを使えば、ニューラルネットワークのパラメータの更新は、次のように行うことができます（次に示すコードは、実際には動作しない擬似コードです）

```
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の実装

---

ここで登場する変数名のoptimizerとは、「最適化を行う者」という意味の単語です。

ここではSGDが、その役割を担います。パラメータの更新は、optimizerが責任を持って遂行してくれます。

私たちがここでやるべきなのは、optimizerにパラメータと勾配の情報を渡すことだけです。

# SGDの欠点

SGDは単純で実装も簡単ですが、問題によっては非効率な場合があります。ここでは、SGDの欠点を指摘するにあたって、次の関数の最小値を求める問題を考えたいと思います。

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

# SGDの欠点

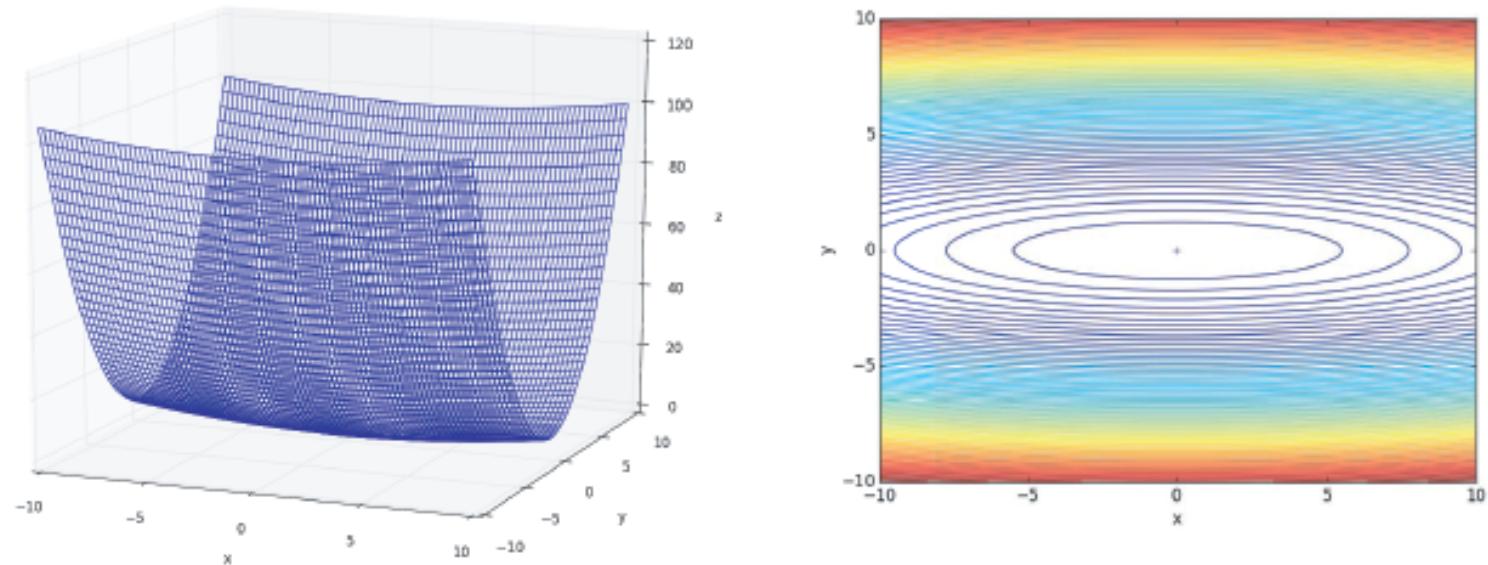


図 6-1  $f(x, y) = \frac{1}{20}x^2 + y^2$  のグラフ（左図）とその等高線（右図）

# SGDの欠点

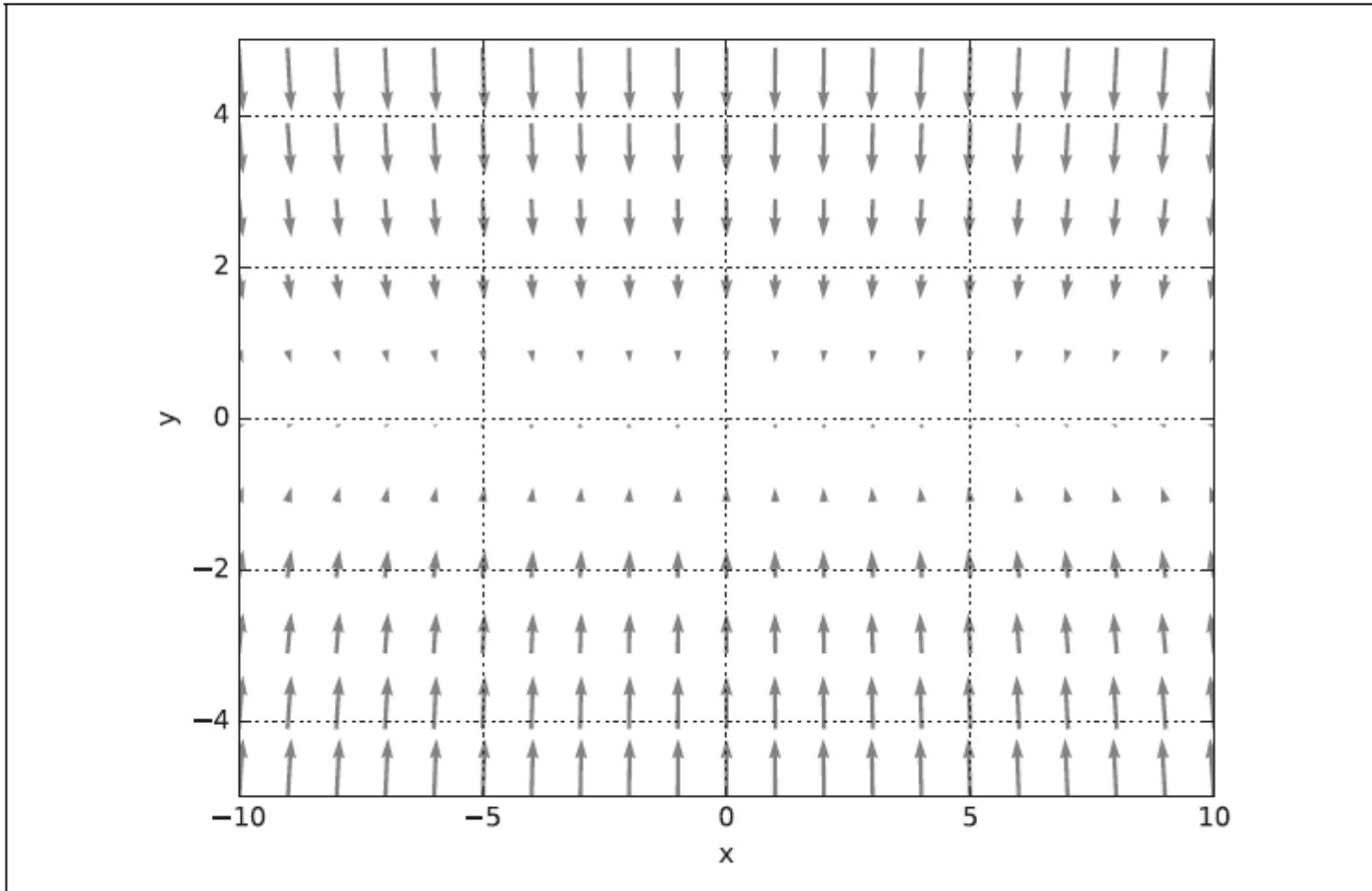


図 6-2  $f(x, y) = \frac{1}{20}x^2 + y^2$  の勾配

# SGDの欠点

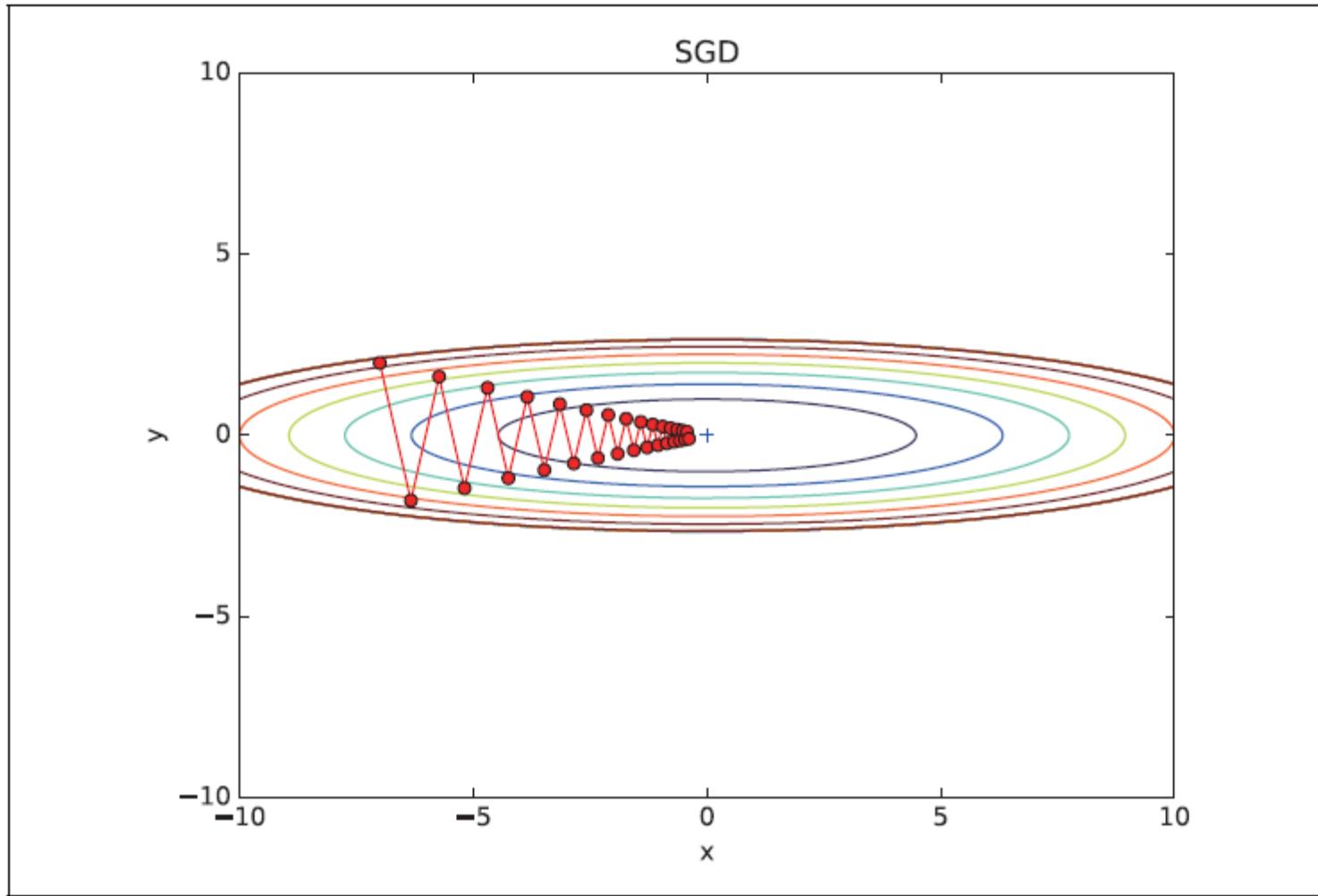


図 6-3 SGD による最適化の更新経路：最小値の  $(0, 0)$  へジグザグに動くため非効率

# SGDの欠点

SGDは、図6-3に示すようなジグザグな動きをします。これはかなり非効率な経路です。つまり、SGDの欠点は、関数の形状が等方的でないと——伸びた形の関数だと——、非効率な経路で探索することになる点にあるのです。そこで、SGDで行ったような、単に勾配方向へ進むよりも、もっとスマートな方法が求められるのです。なお、SGDの非効率な探索経路の根本的な原因は、勾配の方向が本来の最小値ではない方向を指していることに起因します。

この欠点を改善するため、SGDに代わる手法として、**Momentum**、**Adagard**、**Adam**という三つ手法を紹介します

# Momentum

$$\mathbf{v} \leftarrow \alpha \mathbf{v} - \eta \frac{\partial L}{\partial \mathbf{W}}$$

$$\mathbf{W} \leftarrow \mathbf{W} + \mathbf{v}$$

$\mathbf{W}$ : 更新する重みパラメータ

$\frac{\partial L}{\partial \mathbf{W}}$ :  $\mathbf{W}$ に関する損失関数の勾配

$\eta$ : 学習係数 (learning rate)

$\mathbf{v}$ : 物理で言うところの「速度」に対応します

# Momentum

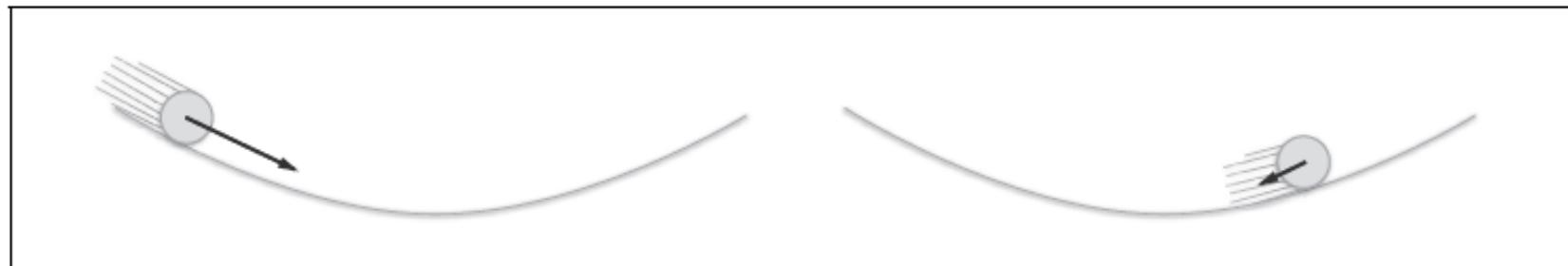


図 6-4 Momentum のイメージ：ボールが地面の傾斜を転がるように動く

$\alpha v :$

この項は、物体が何も力を受けないときに徐々に減速するための役割を担います ( $\alpha$  は 0.9 などの値を設定します)。

物理では、地面の摩擦や空気抵抗に対応します。

# Momentumの実装

```
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の更新経路

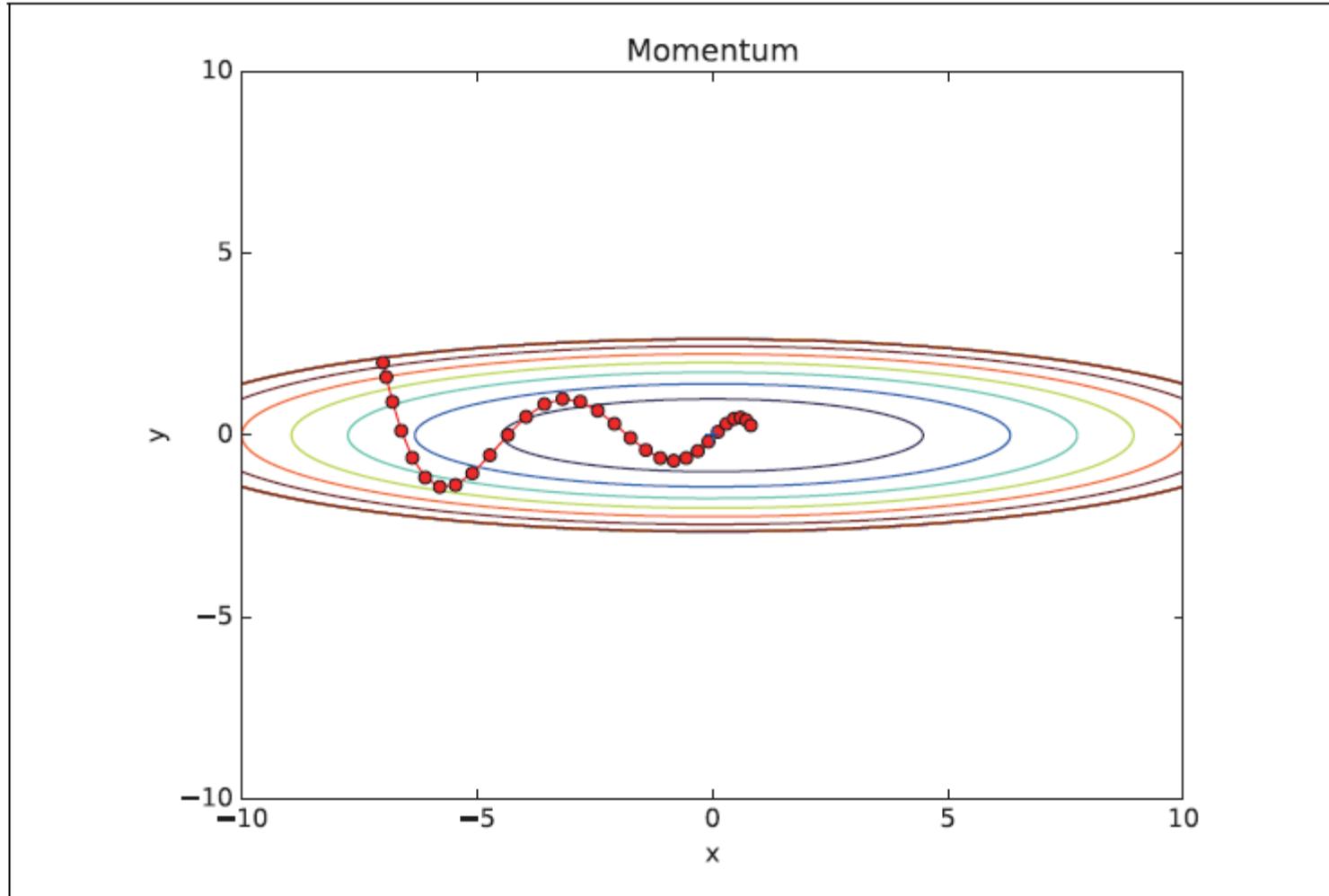


図 6-5 Momentum による最適化の更新経路

# AdaGrad

この学習係数に関する有効なテクニックとして、**学習係数の減衰** (learning rate decay) という方法があります。これは、学習が進むにつれて学習係数を小さくするという方法です。最初は“大きく”学習し、次第に“小さく”学習するという手法で、ニューラルネットワークの学習では実際によく使われます。

$$\mathbf{h} \leftarrow \mathbf{h} + \frac{\partial L}{\partial \mathbf{W}} \odot \frac{\partial L}{\partial \mathbf{W}}$$

$$\mathbf{W} \leftarrow \mathbf{W} - \eta \frac{1}{\sqrt{\mathbf{h}}} \frac{\partial L}{\partial \mathbf{W}}$$

$\mathbf{h}$ :これまで経験した勾配の値を2乗和

# AdaGradの実装

AdaGradは、過去の勾配を2乗和としてすべて記録します。そのため、学習を進めれば進めるほど、更新度合いは小さくなります。

```
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)
```

0で除算してしま  
うことを防ぐため



# Adam

Momentum は、ボールがお椀を転がるように物理法則に準じる動きをしました。AdaGrad は、パラメータの要素ごとに、適応的に更新ステップを調整しました。それでは、その 2 つの手法—— Momentum と AdaGrad——を融合するとどうなるでしょうか？それが Adam [8] という手法のベースとなるアイデアです<sup>†1</sup>。

Adam は 2015 年に提案された新しい手法です。その理論はやや複雑ですが、直感的には、Momentum と AdaGrad を融合したような手法です。先の 2 つの手法の利点を組み合わせることで、効率的にパラメータ空間を探索することが期待できます。また、ハイパーパラメータの「バイアス補正（偏りの補正）」が行われていることも Adam の特徴です。ここでは、これ以上踏み込んで説明することは避けます。詳細は原著論文 [8] を参照してください。また、Python の実装については、common/optimizer.py に Adam というクラスで実装してあるので、興味のある方は参照してください。

# Adamの更新経路

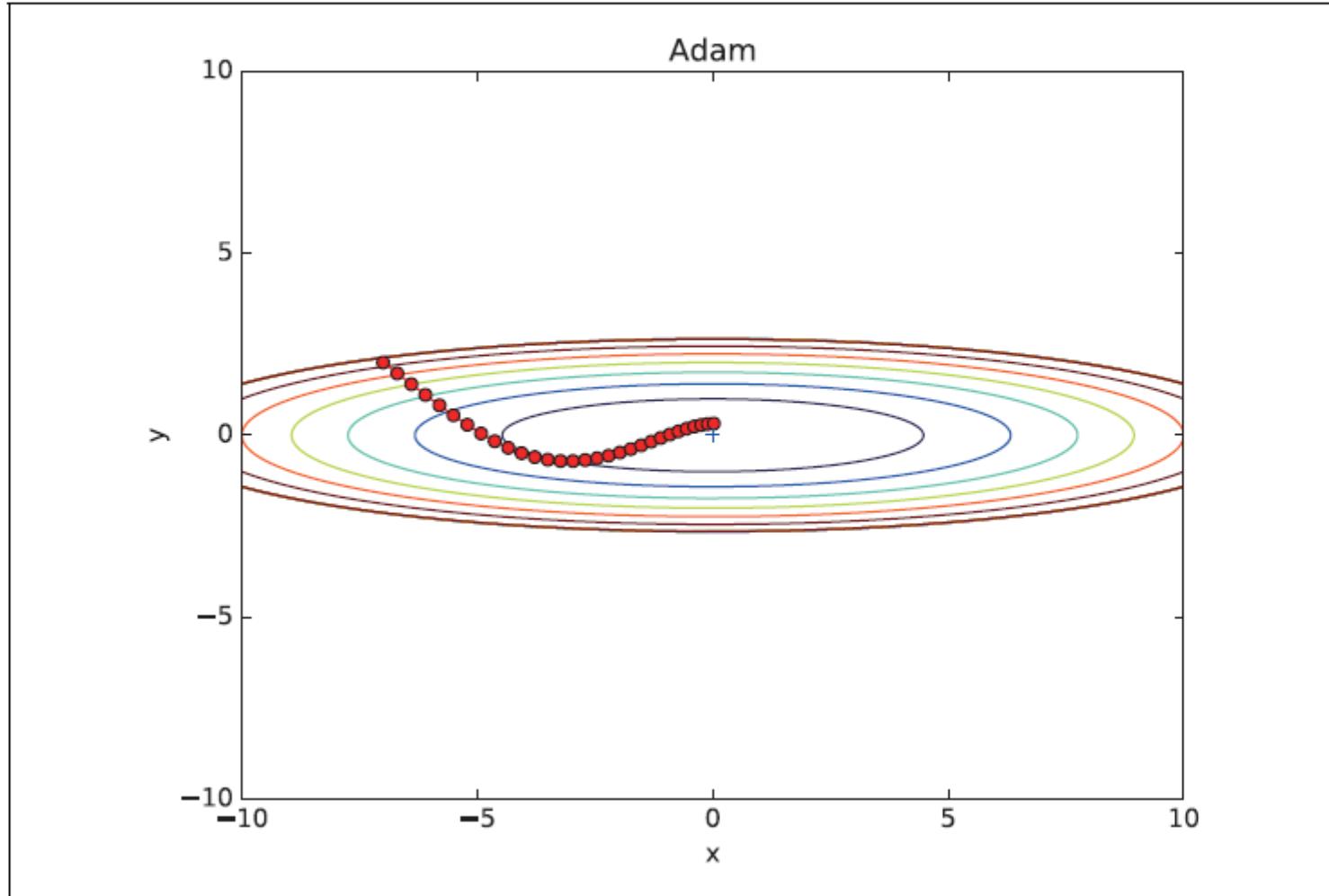


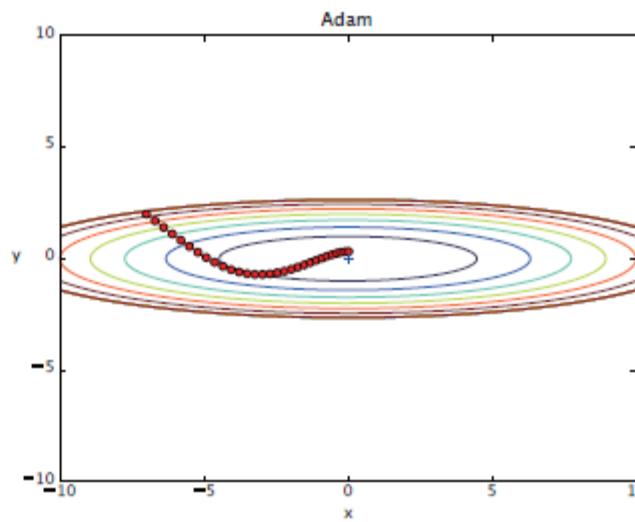
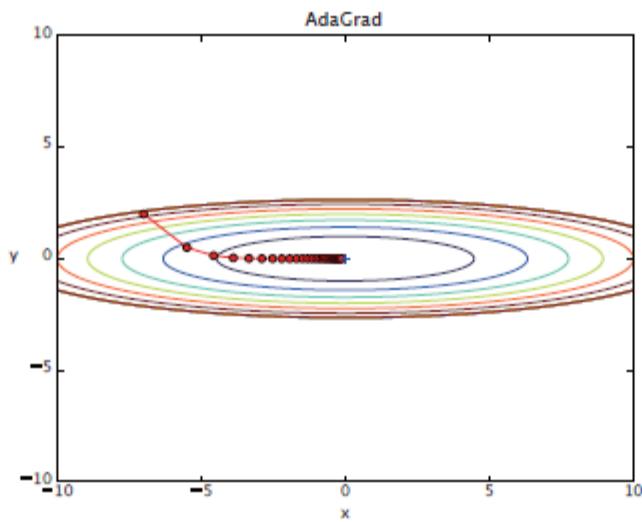
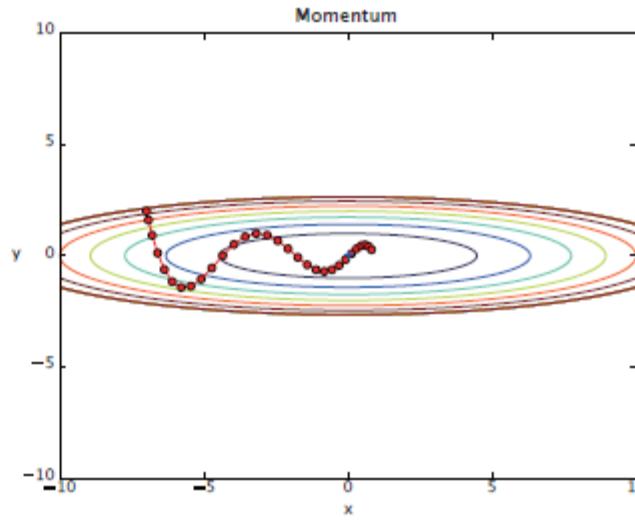
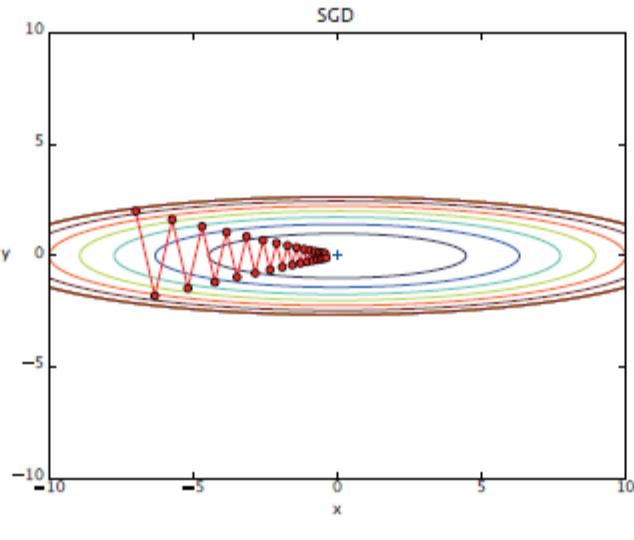
図 6-7 Adam による最適化の更新経路

# How to choose

---

SGD、Momentum、AdaGrad、Adam と 4 つの手法を説明してきましたが、どれを用いたらよいのでしょうか？ 残念ながら、すべての問題で優れた手法というのは（今のところ）ありません。それぞれに特徴があり、得意な問題、不得意な問題があります。

# How to choose



## 6.2 重みの初期値

---

6.2 Initial Weight Value

# 重みの初期値

0??? No

なぜ重みの初期値を 0 にしてはいけない——正確には、重みを均一な値に設定してはいけない——のでしょうか？ それは誤差逆伝播法において、すべての重みの値が均一に（同じように）更新されてしまうからです。たとえば、2 層のニューラルネットワークにおいて、1 層目と 2 層目の重みが 0 だと仮定します。そうすると、順伝播時には、入力層の重みが 0 であるため、2 層目のニューロンにはすべて同じ値が伝達されます。2 層目のニューロンですべて同じ値が入力されるということは、逆伝播のときに 2 層目の重みはすべて同じように更新されることになります（「乗算ノードの逆伝播」を思い出しましょう）。そのため、均一の値で更新され、重みは対称的な値（重複した値）を持つようになってしまふのです。これでは、たくさんの中の重みを持つ意味がなくなってしまいます。この「重みが均一になってしまふこと」を防ぐ——正確には、重みの対称的な構造を崩す——ために、ランダムな初期値が必要なのです。

# 隠れ層のアクティベーション分布

---

実験：

重みの初期値によって隠れ層のアクティベーションがどうのようになりますか

条件：

5層のニューラルネットワーク(sigmoid関数)に、ランダムに生成した入力データを流します

# 実験のソースコード

```
import numpy as np
import matplotlib.pyplot as plt

def sigmoid(x):
    return 1 / (1 + np.exp(-x))

x = np.random.randn(1000, 100) # 1000 個のデータ
node_num = 100              # 各隠れ層のノード（ニューロン）の数
hidden_layer_size = 5        # 隠れ層が 5 層
activations = {}            # ここにアクティベーションの結果を格納する

for i in range(hidden_layer_size):
    if i != 0:
        x = activations[i-1]

w = np.random.randn(node_num, node_num) * 1


---


z = np.dot(x, w)
a = sigmoid(z)  # シグモイド関数！
activations[i] = a
```

# ヒストグラム(標準偏差 1)

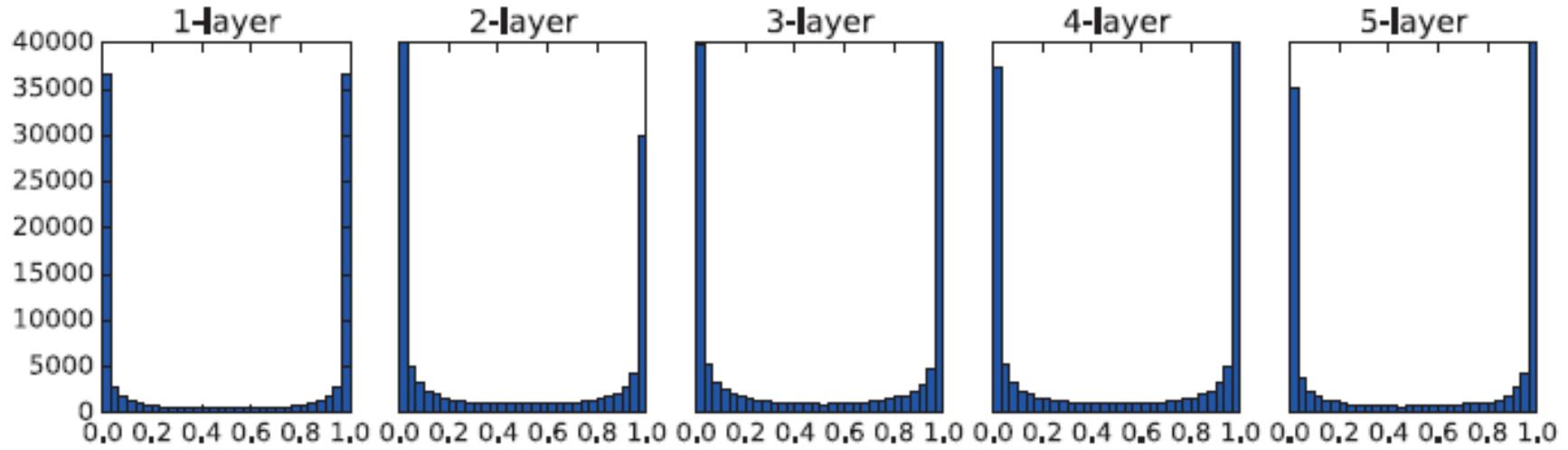


図 6-10 重みの初期値として標準偏差 1 のガウス分布を用いたときの、各層のアクティベーションの分布

勾配消失(gradient vanishing)

```
# w = np.random.randn(node_num, node_num) * 1  
w = np.random.randn(node_num, node_num) * 0.01
```

# ヒストグラム(標準偏差0.01)

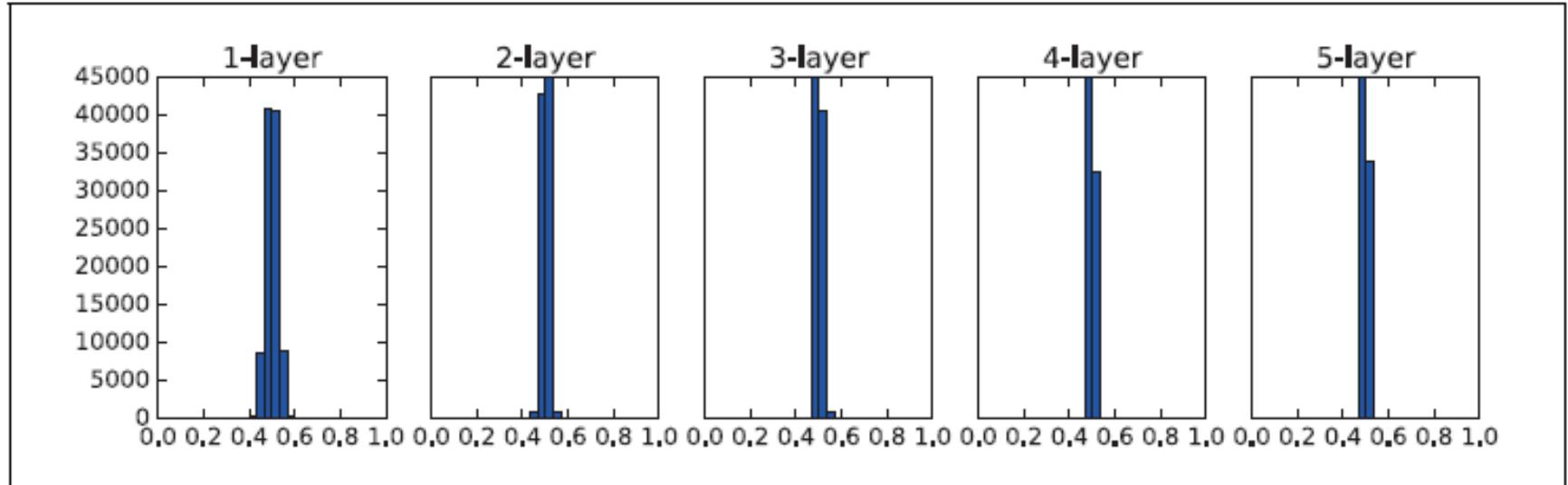


図6-11 重みの初期値として標準偏差 0.01 のガウス分布を用いたときの、各層のアクティベーションの分布

表現力の制限  
(expressive force-restricted)

# ヒストグラム(Xavierの初期値)

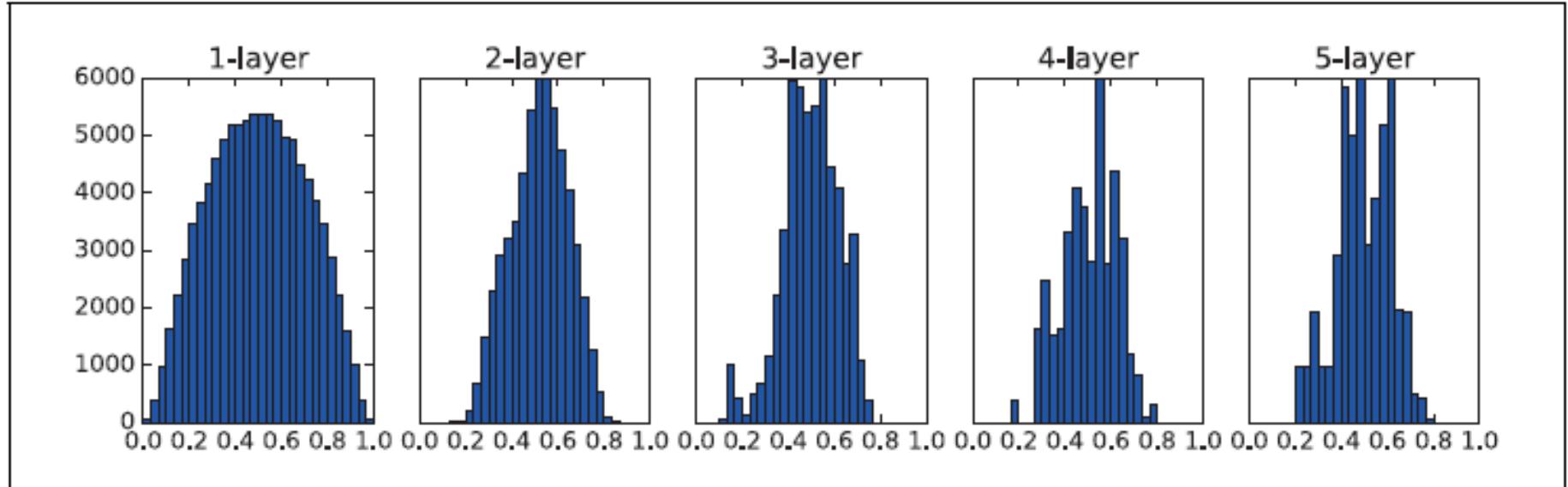


図 6-13 重みの初期値として「Xavier の初期値」を用いたときの、各層のアクティベーションの分布

$$\text{Xavierの初期値} = \frac{1}{\sqrt{n}}$$

効率的に学習

! ReLUを除く

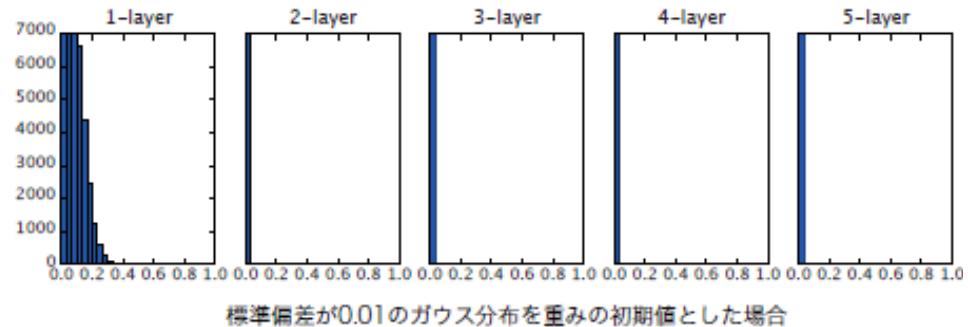
# ReLUの重みの初期値

「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す。  
sigmoid 関数や tanh 関数は左右対称で中央付近が線形関数として見なせるので、「Xavier の初期値」が適しています。一方、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ています。それは、Kaiming He らが推奨する初期値——その名も「He の初期値」<sup>[10]</sup> です。「He の初期値」は、前層のノードの数が  $n$  個の場合、 $\sqrt{\frac{2}{n}}$  を標準偏差とするガウス分布を用います。「Xavier の初期値」が  $\sqrt{\frac{1}{n}}$  であったことを考えると、ReLU の場合は負の領域が 0 になるため、より広がりを持たせるために倍の係数が必要になると（直感的には）解釈できます。

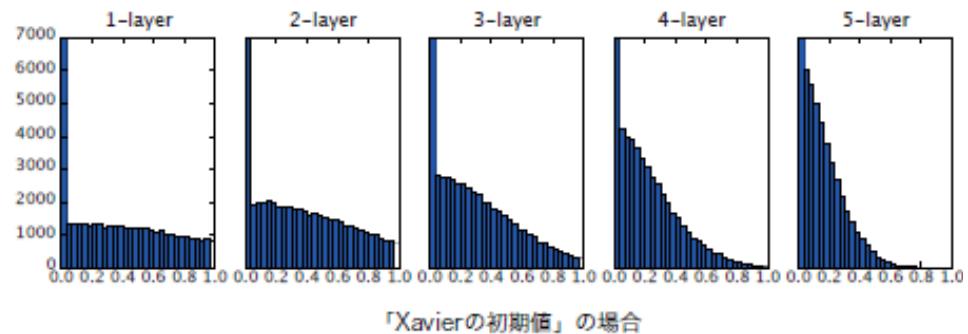
[10] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun (2015) : Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification. In 1026–1034.

# ReLUの重みの初期値

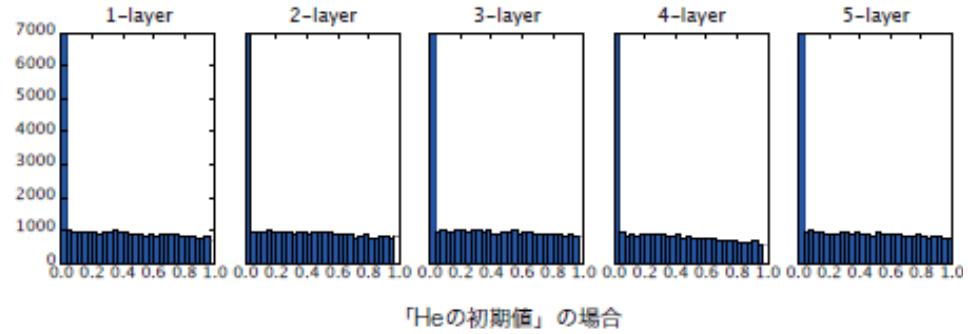
標準差 = 0.01



Xavierの初期値



Heの初期値



# MNISTデータセットによる重み初期値の比較

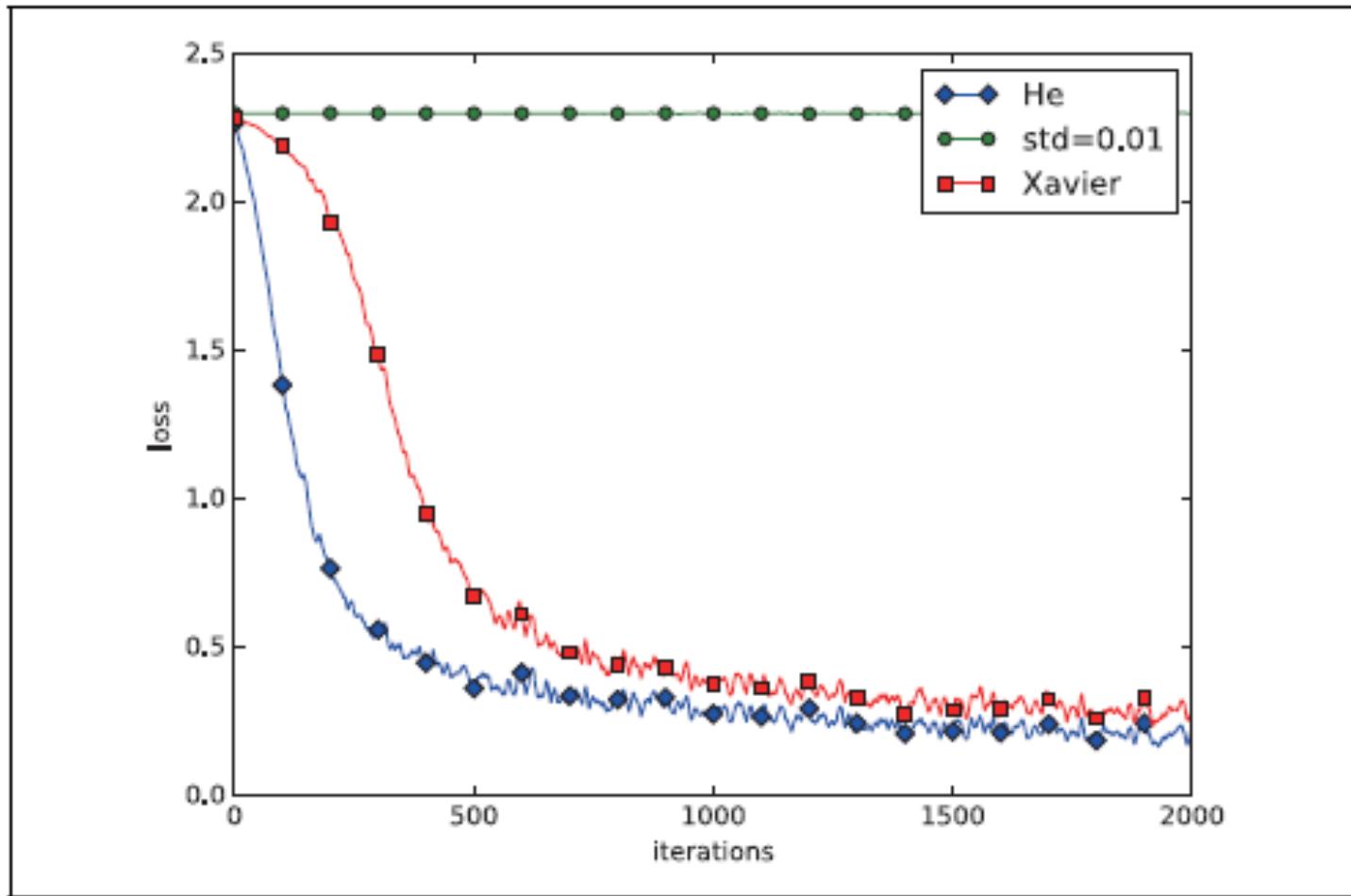


図 6-15 MNIST データセットに対する「重みの初期値」による比較：横軸は学習の繰り返し回数 (iterations)、縦軸は損失関数の値 (loss)

## 6.3 Batch Normalization

---

To be continued...