DAY2

Section1：勾配消失問題

◇要点
学習の流れ
1．入力層に値を入力
2．重み、バイアス、活性化関数で計算しながら値が伝わる
3．出力層から値が伝わる
4．出力層から出た値と正解値から、誤差関数を使って誤差を求める
5.誤差を小さくするために重みやバイアスを更新する
6．1～5 の操作を繰り返すことにより、出力値を正解値に近づけていく
計算結果（=誤差）から微分を逆算することで、不要な再帰的計算を避けて微分を算出できる

・連鎖律の原理を使い、dz/dx を求めよ。
z = t2
t = x +y
A.
dz/dt = 2t
dt/dx = y
dz/dx = dz/dt*dt/dx = 2ty
勾配消失問題
誤差逆伝播法が下位層に進んでいくに連れて、勾配がどんどん緩やかになっていく。そのため、勾配降下法による、更新では下位層のパラメータはほとんど変わらず、訓練は最適値に収束しなくなる。

・シグモイド関数を微分した時、入力値が 0 の時に最大値をとる。その値として正しいものを選択肢から選べ。
0.25

勾配消失の解決方
・活性化関数の選択
　ReLU 関数
　今最も使われている活性化関数勾配消失問題の回避とスパース化に貢献することで良い成果をもたらしている
・重みの初期値設定
Xavier
　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
　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

He

He の初期値を設定する際の活性化関数　Relu 関数

初期値の設定方法

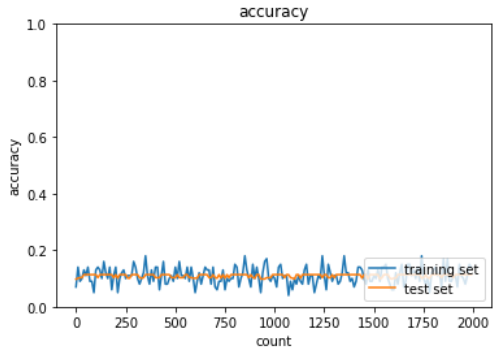
　重みの要素を、前の層のノード数の平方根で除算した値に対し√２をかけ合わせた値

network['W1'] = np.random.randn(input_layer_size , hidden_layer_size) / np.sqrt(input_layer_size) * np.sqrt(2)

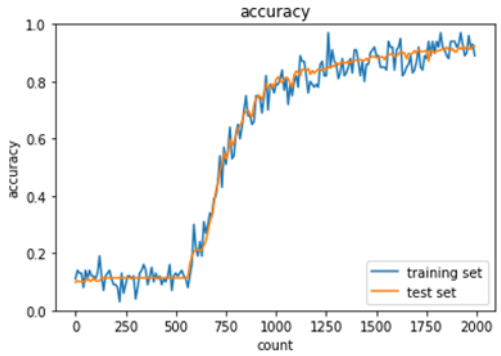
network['W2']=np.random.randn(hidden_layer_size,output_layer_size)/np.sqrt(hidden_layer_size) * np.sqrt(2)

シグモイド関数　初期値の加工なし

勾配消失問題が発生

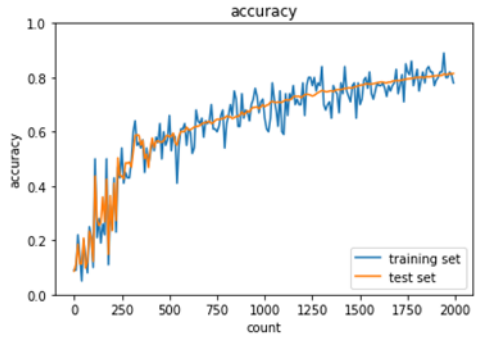


ReLU 関数　初期値の加工なし



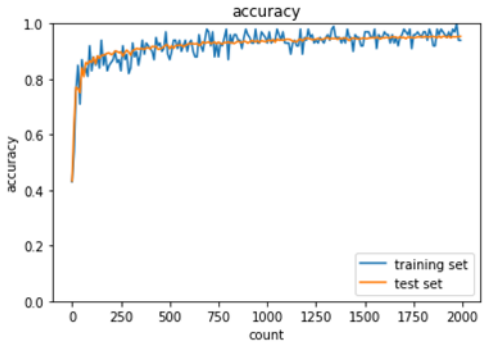
学習が進んでいる。

シグモイド関数　Xavier の初期値



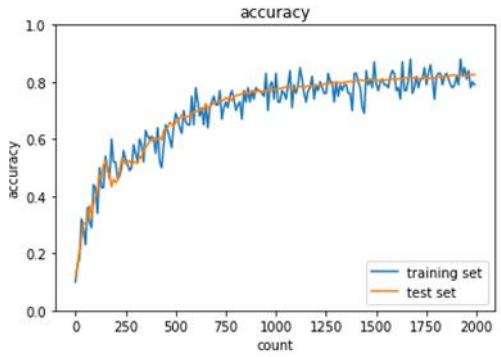
勾配消失問題は発生しない

ReLU 関数、He の初期値



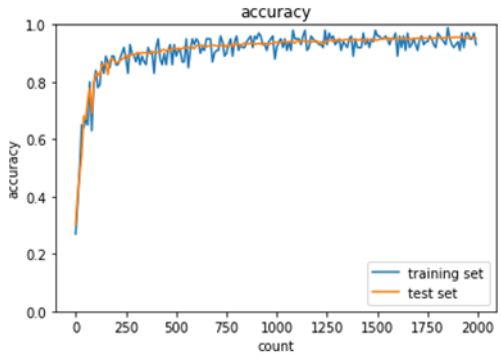
He の初期値の効果は高い。

シグモイド関数　He の初期値



Xavier の初期値と比較して、大きな違いは見受けられない。

ReLU 関数　Xavier の初期値



He の初期値と同様

・重みの初期値に 0 を設定すると、どのような問題が発生するか。

A.パラメータのチューニングが行われない

・バッチ正規化

ミニバッチ単位で入力値のデータの偏りを抑制する手法。

活性化関数に値を渡す前後にバッチ正規化の層を加える。

活性化関数の前の場合、バッチ正規化層の入力は $u = w z + b$ 又は z

・一般的に考えられるバッチ正規化の効果を 2 点挙げよ。

A.計算が早い。勾配消失問題が発生しにくくなる。

例題チャレンジ

data_x[i:i_end], data_t[i:i_end

Section2：学習率最適化手法

◇要点

誤差を最小にするネットワークを作成するため、勾配降下法を用いてパラメータを最適化

$$\mathbf{w}^{(t+1)} = \mathbf{w}^{(t)} - \varepsilon \nabla E$$

$$\nabla E = \frac{\partial E}{\partial \mathbf{w}} = \left[\frac{\partial E}{\partial w_1} \cdots \frac{\partial E}{\partial w_M} \right]$$

学習率ε

学習率が大きすぎた場合、最小値にいつまでもたどり着かず発散する。

学習率が小さすぎた場合、発散することはないが、収束するまでに時間がかかる。

勾配降下法の種類

・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。

$$V_t = \mu V_{t-1} - \epsilon \nabla E$$

```
self.v[key] = self.momentum* self.v[key] -self.learning_rate* grad[key]
```

$$\mathbf{w}^{(t+1)} = \mathbf{w}^{(t)} + V_t$$

```
params[key] += self.v[key]
```

慣性：u

・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

メリットは、勾配の緩やかな斜面に対して、最適値に近づける

課題は、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。

$$h_0 = \theta$$

```
self.h[key] = np.zeros_like(val)
```

$$h_t = h_{t-1} + (\nabla E)^2$$

```
self.h[key] += grad[key] * grad[key]
```

$$\mathbf{w}^{(t+1)} = \mathbf{w}^{(t)} - \epsilon \frac{1}{\sqrt{h_t} + \theta} \nabla E$$

```
params[key] -= self.learning_rate* grad[key] / (np.sqrt(self.h[key]) + 1e-7)
```

・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

メリット

・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

$$h_t = \alpha h_{t-1} + (1-\alpha)\ (\nabla E)^2$$

self.h[key] *= self.decay_rate

self.h[key] += (1 -self.decay_rate) * grad[key] * grad[key]

$$\mathbf{w}^{(t+1)} = \mathbf{w}^{(t)} - \epsilon \frac{1}{\sqrt{h_t} + \theta} \nabla E$$

params[key] -= self.learning_rate* grad[key] / (np.sqrt(self.h[key]) + 1e-7)

・Adam

・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

メリット

モメンタムおよび RMSProp のメリットをはらんだアルゴリズム

2_1_network_modified.ipynb

準備

Googleドライブのマウント

```
[1] from google.colab import drive
    drive.mount('/content/drive')

    Mounted at /content/drive
```

sys.pathの設定

```
[14] sys.path.append('/content/drive/My Drive/Colab Notebooks/DNN_code_colab_lesson_1_2')
```

network modified

layers

```
import numpy as np
from common import layers
from collections import OrderedDict
from common import functions
from data.mnist import load_mnist
import matplotlib.pyplot as plt


# ReLU layer
class Relu:
    def __init__(self):
        self.mask = None

    def forward(self, x):
        # mask.shape = x.shape
```

```
        return out

    def backward(self, dout):
        # Trueの箇所を0にする
        dout[self.mask] = 0
        dx = dout

        return dx

# Affine layer(全結合 layer)
class Affine:

    def __init__(self, W, b):
        self.W =W
        self.b = b

        self.x = None
        self.original_x_shape = None
        # 重み・バイアスパラメータの微分
        self.dW = None
        self.db = None

    def forward(self, x):
        out = np.dot(self.x, self.W) + self.b
```

```
[15]    self.db = np.sum(dout, axis=0)

        return dx
```

two layer network class

```
class TwoLayerNet:
    '''
    input_size: 入力層のノード数
    hidden_size: 隠れ層のノード数
    output_size: 出力層のノード数
    weight_init_std: 重みの初期化方法
    '''
    def __init__(self, input_size, hidden_size, output_size, weight_init_std = 0.01):
        # 重みの初期化
        self.params = {}
        self.params['W1'] = weight_init_std * np.random.randn(input_size, hidden_size)
        self.params['W2'] = weight_init_std * np.random.randn(hidden_size, output_size)
        self.params['b1'] = np.zeros(hidden_size)
        self.params['b2'] = np.zeros(output_size)

        # レイヤの生成
```

```
        self.lastLayer = layers.SoftmaxWithLoss()

    # 順伝播
    def predict(self, x):
        for layer in self.layers.values():
            x = layer.forward(x)

        return x

    # 誤差
    def loss(self, x, d):
        y = self.predict(x)
        return self.lastLayer.forward(y, d)

    # 精度
    def accuracy(self, x, d):
        y = self.predict(x)
        y = np.argmax(y, axis=1)
        if d.ndim != 1 : d = np.argmax(d, axis=1)

        accuracy = np.sum(y == d) / float(x.shape[0])
        return accuracy

    # 勾配
    def gradient(self, x, d):
        # forward
```

```
[16]    dout = self.lastLayer.backward(dout)

        layers = list(self.layers.values())
        layers.reverse()
        for layer in layers:
            dout = layer.backward(dout)

        # 設定
        grad = {}
        grad['W1'], grad['b1'] = self.layers['Affine1'].dW, self.layers['Affine1'].db
        grad['W2'], grad['b2'] = self.layers['Affine2'].dW, self.layers['Affine2'].db

        return grad
```

1_4_1_mnist_sample modified

```
# データの読み込み
(x_train, d_train), (x_test, d_test) = load_mnist(normalize=True, one_hot_label=True)

print("データ読み込み完了")

network = TwoLayerNet(input_size=784, hidden_size=40, output_size=10)
```

```
train_loss_list = []
accuracies_train = []
accuracies_test = []

plot_interval=10

for i in range(iters_num):
    batch_mask = np.random.choice(train_size, batch_size)
    x_batch = x_train[batch_mask]
    d_batch = d_train[batch_mask]

    # 勾配
    grad = network.gradient(x_batch, d_batch)

    for key in ('W1', 'W2', 'b1', 'b2'):
        network.params[key] -= learning_rate * grad[key]

    loss = network.loss(x_batch, d_batch)
    train_loss_list.append(loss)

    if (i + 1) % plot_interval == 0:
        accr_test = network.accuracy(x_test, d_test)
        accuracies_test.append(accr_test)
        accr_train = network.accuracy(x_batch, d_batch)
        accuracies_train.append(accr_train)
```

```
lists = range(0, iters_num, plot_interval)
plt.plot(lists, accuracies_train, label="training set")
plt.plot(lists, accuracies_test,  label="test set")
plt.legend(loc="lower right")
plt.title("accuracy")
plt.xlabel("count")
plt.ylabel("accuracy")
plt.ylim(0, 1.0)
# グラフの表示
plt.show()
```

```
データ読み込み完了
Generation: 10. 正答率(トレーニング) = 0.13
              : 10. 正答率(テスト) = 0.1034
Generation: 20. 正答率(トレーニング) = 0.15
              : 20. 正答率(テスト) = 0.1605
Generation: 30. 正答率(トレーニング) = 0.36
              : 30. 正答率(テスト) = 0.4031
Generation: 40. 正答率(トレーニング) = 0.47
              : 40. 正答率(テスト) = 0.3991
Generation: 50. 正答率(トレーニング) = 0.39
              : 50. 正答率(テスト) = 0.4231
Generation: 60. 正答率(トレーニング) = 0.47
              : 60. 正答率(テスト) = 0.4765
Generation: 70. 正答率(トレーニング) = 0.53
              : 70. 正答率(テスト) = 0.5729
Generation: 80. 正答率(トレーニング) = 0.65
```

```
Generation: 120. 正答率(トレーニング) = 0.82
               : 120. 正答率(テスト) = 0.7539
Generation: 130. 正答率(トレーニング) = 0.77
               : 130. 正答率(テスト) = 0.7873
Generation: 140. 正答率(トレーニング) = 0.79
               : 140. 正答率(テスト) = 0.797
Generation: 150. 正答率(トレーニング) = 0.78
               : 150. 正答率(テスト) = 0.8222
```

```
               : 190. 正答率(テスト) = 0.8408
Generation: 200. 正答率(トレーニング) = 0.9
               : 200. 正答率(テスト) = 0.8348
Generation: 210. 正答率(トレーニング) = 0.87
               : 210. 正答率(テスト) = 0.8403
Generation: 220. 正答率(トレーニング) = 0.88
               : 220. 正答率(テスト) = 0.8577
Generation: 230. 正答率(トレーニング) = 0.9
               : 230. 正答率(テスト) = 0.866
Generation: 240. 正答率(トレーニング) = 0.92
               : 240. 正答率(テスト) = 0.8697
Generation: 250. 正答率(トレーニング) = 0.86
               : 250. 正答率(テスト) = 0.8712
Generation: 260. 正答率(トレーニング) = 0.84
               : 260. 正答率(テスト) = 0.8695
Generation: 270. 正答率(トレーニング) = 0.9
               : 270. 正答率(テスト) = 0.8721
Generation: 280. 正答率(トレーニング) = 0.91
               : 280. 正答率(テスト) = 0.8805
Generation: 290. 正答率(トレーニング) = 0.88
```

- 準備
- Googleドライブのマウント
- sys.pathの設定
- vanishing gradient
- sigmoid - gauss
- ReLU - gauss
- sigmoid - Xavier
- ReLU - He

Section3：過学習

◇要点

過学習

　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

原因

・パラメータの数が多い

・パラメータの値が適切でない

・ノードが多い

ネットワークの自由度(層数、ノード数、パラメータの値)が高い

正則化

ネットワークの自由度(層数、ノード数、パラメータの値)を制約すること

正則化手法を利用して過学習を抑制する

正則化手法

・L 1 正則化、L2 正則化

・ドロップアウト

機械学習で使われる線形モデル(線形回帰,主成分分析…etc)の正則化は、
モデルの重みを制限することで可能となる。
前述の線形モデルの正則化手法の中にリッジ回帰という手法があり、
その特徴として正しいものを選択しなさい。

(a)ハイパーパラメータを大きな値に設定すると、すべての重みが限りなく0に近づく
(b)ハイパーパラメータを0に設定すると、非線形回帰となる。
(c)バイアス項についても、正則化される
(d)リッジ回帰の場合、隠れ層に対して正則化項を加える

Weight decay(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。

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

適用方法

1. ある層上の複数の重みに対して p ノルムを計算する。
2. 他の層に対しても同様の計算をする。
3. すべての層の p ノルムを足し算する。
4. 3.で求めた p ノルムに係数をかけて、誤差関数に加える。

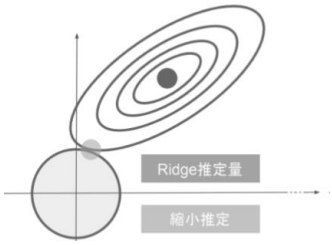
$$E_n(\mathbf{w}) + \frac{1}{p}\lambda \| x \|_p$$

weight_decay+= weight_decay_lambda
*np.sum(np.abs(network.params['W' + str(idx)]))
loss = network.loss(x_batch, d_batch) + weight_decay

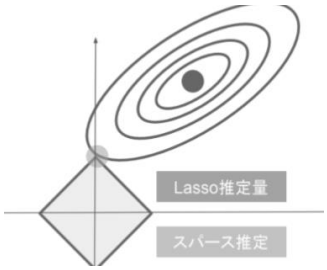
$$\| x \|_p = \left(| x_1 |^p + \ldots + | x_n |^p \right)^{\frac{1}{p}}$$ p = 1 の場合、L1 正則化 p = 2 の場合、L2 正則化

np.sum(np.abs(network.params['W' + str(idx)]))

L1 正則化を表しているグラフ



L 2 正則化を表しているグラフ



5. L2 パラメータ正則化

深層学習において、過学習の抑制・汎化性能の向上のために正則化が用いられる。そのひとつに、L2 ノルム正則化（Ridge, Weigh Decay）がある。以下は L2 正則化を適用した場合に、パラメータの更新を行うプログラムである。あるパラメータ param と正則化がないときにそのパラメータに伝播される誤差の勾配 grad が与えられたとする。
最終的な勾配を計算する（え）にあはてはまるのはどれか。ただし rate は L2 正則化の係数を表すとする。

```
def ridge(param, grad, rate):
    """
    param: target parameter
    grad: gradients to param
    rate: ridge coefficient
    """
    grad += rate * （え）
```

(1) np.sum(param**2)
(2) np.sum(param)
(3) param**2
(4) param

6. L1 パラメータ正則化

以下は L1 ノルム正則化（Lasso）を適用した場合に、パラメータの更新を行うプログラムである。あるパラメータ param と正則化がないときにそのパラメータに伝播される誤差の勾配 grad が与えられたとする。
最終的な勾配を計算する（お）にあはてはまるのはどれか。ただし rate は L1 正則化の係数を表すとする。

```
def lasso(param, grad, rate):
    """
    param: target parameter
    grad: gradients to param
    rate: lasso coefficient
    """
    x = （お）
    grad += rate * x
```

(1) np.maximum(param, 0)
(2) np.minimum(param, 0)
(3) np.sign(param)
(4) np.abs(param)

7. データ集合の拡張

画像認識などにおいて、精度向上や汎化性能の向上のためにデータ拡張が行われることが多い。データ拡張には、画像を回転・反転させるなど様々な種類がある。以下は画像をランダムに切り取る処理を行うプログラムである。これは画像中の物体の位置を移動させるなどの意味がある。
（か）にはてはまるのはどれか。

```
def random_crop(image, crop_size):
    """
    image: (height, width, channel)
    crop_size: (crop_height, crop_width)

    height >= crop_height, width >= crop_width
    """
    h, w, _ = image.shape
    crop_h, crop_w = crop_size

    # 切り取る位置をランダムに決める
    top = np.random.randint(0, h - crop_h)
    left = np.random.randint(0, w - crop_w)
    bottom = top + crop_h
    right = left + crop_w

    image = （か）
    return image
```

(1) image[:, top:bottom, left:right]
(2) image[:, bottom:top, right:left]
(3) image[bottom:top, right:left, :]
(4) image[top:bottom, left:right, :]

ドロップアウト　ランダムにノードを削除して学習させること

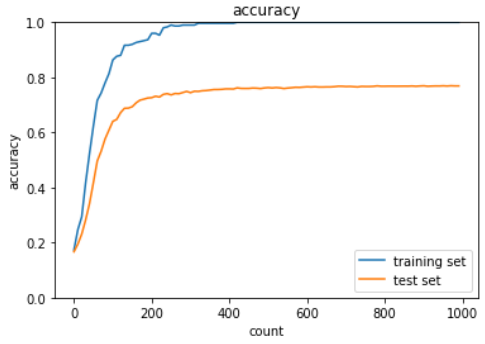
過学習の課題

・ノードの数が多い

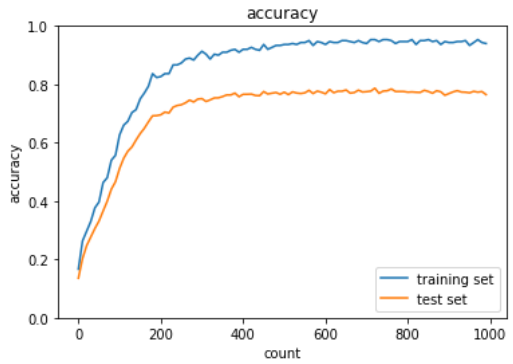
メリット

・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

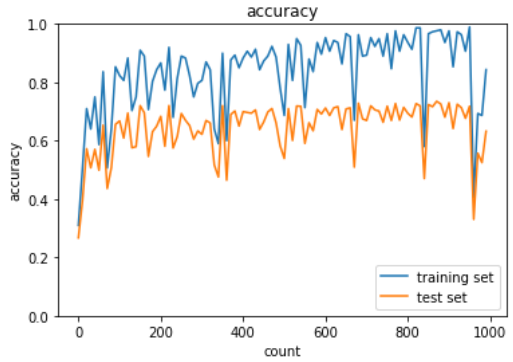
overfiting



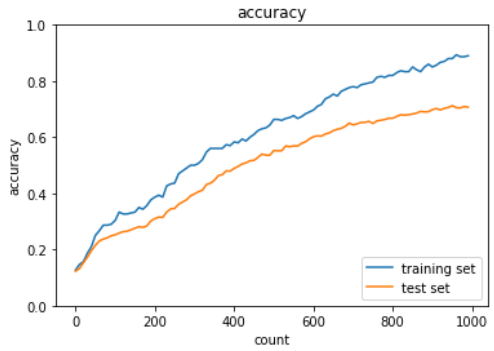
weight decay　L2



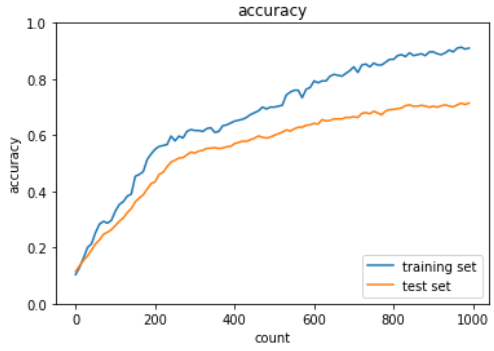
L1



Dropout



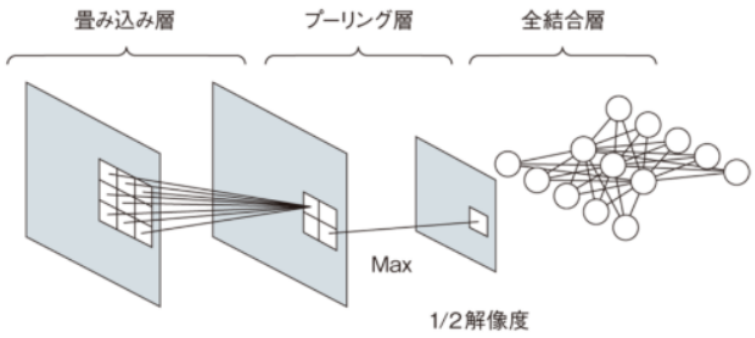
Dropout + L1



Section4：畳み込みニューラルネットワークの概念

◇要点

CNN の構造図



・畳み込み層

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。

結論: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。

畳み込みの概念

　フィルターをかける、バイアスを足し合わせる→ 出力画像が出せる

・パディング

　ゼロなど固定データで枠をつける。

・スライド

　スライド 2 の場合、決められた領域をまとめて 2 マス移動

・チャンネル

　全結合でいう重み

全結合で画像を学習した際の課題

全結合層のデメリット画像の場合、縦、横、チャンネルの 3 次元データだが、1 次元のデータとして処理される。

↓

RGB の各チャンネル間の関連性が、学習に反映されないということ

・プーリング層

入力画像⇒対象領域の MAX 値または平均値⇒出力値

・サイズ 6×6 の入力画像を、サイズ 2×2 のフィルタで畳み込んだ時の出力画像のサイズを答えよ。

　なおストライドとパディングは 1 とする。

7×7

Section5：最新の CNN

◇要点

AlexNet のモデルの説明

タイトル：深い畳み込みニューラルネットワークを用いたイメージネットの分類

年代：2012 年の論文

URL：

https://papers.nips.cc/paper/4824-imagenet-classification-with-deep-convolutional-neural-networks.pdf

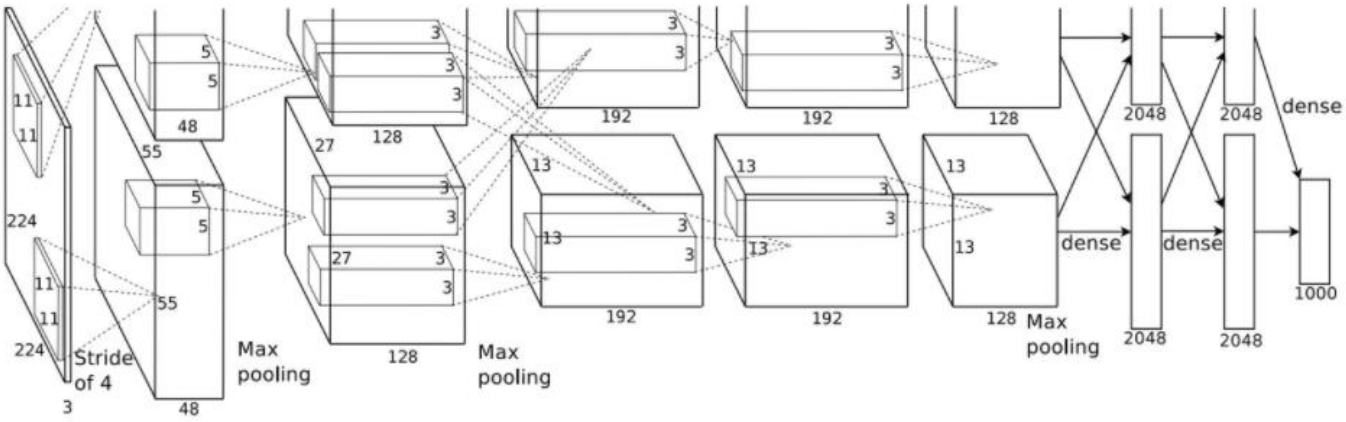
作者：Alex Krizhevsky & Ilya Sutskever & Geoffrey E. Hinton

過学習を防ぐ施策・サイズ 4096 の全結合層の出力にドロップアウトを使用している

・モデルの構造

　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 層の全結合層から構成される。

・過学習を防ぐ施策　サイズ 4096 の全結合層の出力にドロップアウトを使用している



AlexNet の特徴

・活性化関数 ReLU

・Max Pooling

・GPU の活用

・Data Augmentation

・Dropout

ReLU

ReLU は Sigmoid 関数とは異なり、階段状ではなく、しかも x=0 において、右微分係数と左微分係数が異なる、微分可能でない関数。

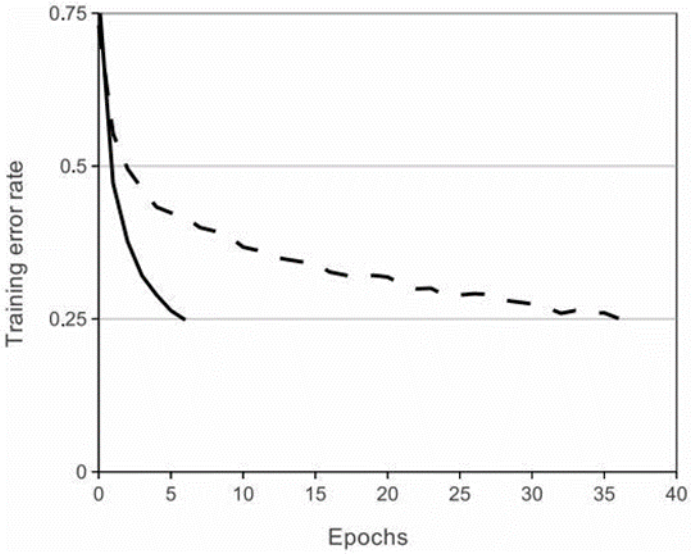
ReLU の課題

1.初期値の与え方によってはうまく学習できない

　Sigmoid 関数は最大で 1 だが、ReLU は∞なので、平均 0.5 の重みでニューロンが 100 あれば次の層に 50 の強さ、その次の層には 5000 の強さになって届く。初期値はニューロンの数によって決める。

2.学習率を調整しないと一気にシナプスの重みが負になってしまい、学習が止まってしまう

　Sigmoid 関数はどこまででも勾配があるが、ReLU は 0 を下回ると勾配がなくなってしまい、学習が行われなくなってしまう。


0.75
0.5
0.25
0
Training error rate
0
5
10
15
20
25
30
35
40
Epochs

2_5_simple_convolution_network.ipynb

準備

Googleドライブのマウント

sys.pathの設定

simple convolution network

image to column

[try] im2colの処理を確認しよう

col2imの処理を確認しよう

simple convolution network class

2_7_double_comvolution_network.ipynb

File Edit View Insert Runtime Tools Help All changes saved

Comment Share

+ Code + Text

RAM Disk Editing

準備

Googleドライブのマウント

```
from google.colab import drive
drive.mount('/content/drive')
```

```
Mounted at /content/drive
```

sys.pathの設定

```
import sys, os
sys.path.append('/content/drive/My Drive/Colab Notebooks/UNK_code_colab_lesson_1_2')
```

double_comvolution_network

```
import pickle
import numpy as np
from collections import OrderedDict
from common import layers
from common import optimizer
from data.mnist import load_mnist
import matplotlib.pyplot as plt

class DoubleConvNet:
    """ conv - relu - pool - conv - relu - pool - affine - relu - affine - softmax
    def __init__(self, input_dim=(1, 28, 28),
                 conv_param_1={'filter_num':10, 'filter_size':2, 'pad':1, 'stride':1},
                 conv_param_2={'filter_num':20, 'filter_size':3, 'pad':1, 'stride':1},
                 hidden_size=100, output_size=10, weight_init_std=0.01):
        conv_output_size_1 = (input_dim[1] - conv_param_1['filter_size']) + 2 * conv_param_1['pad']) / conv_param_1['stride'] + 1
        conv_output_size_2 = (conv_output_size_1 / 2 - conv_param_2['filter_size']) + 2 * conv_param_2['pad']) / conv_param_2['stride'] + 1
        pool_output_size = int(conv_param_2['filter_num'] * (conv_output_size_2 / 2) * (conv_output_size_2 / 2))
        # 重みの初期化
        self.params = {}
```

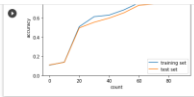
```
        self.params['b3'] = np.zeros(hidden_size)
        self.params['W4'] = weight_init_std * np.random.randn(hidden_size, output_size)
        self.params['b4'] = np.zeros(output_size)
        # レイヤの生成
        self.layers = OrderedDict()
        self.layers['Conv1'] = layers.Convolution(self.params['W1'], self.params['b1'], conv_param_1['stride'], conv_param_1['pad'])
        self.layers['Relu1'] = layers.Relu()
        self.layers['Pool1'] = layers.Pooling(pool_h=2, pool_w=2, stride=2)
        self.layers['Conv2'] = layers.Convolution(self.params['W2'], self.params['b2'], conv_param_2['stride'], conv_param_2['pad'])
        self.layers['Relu2'] = layers.Relu()
        self.layers['Pool2'] = layers.Pooling(pool_h=2, pool_w=2, stride=2)
        self.layers['Affine1'] = layers.Affine(self.params['W3'], self.params['b3'])
        self.layers['Relu3'] = layers.Relu()
        self.layers['Affine2'] = layers.Affine(self.params['W4'], self.params['b4'])
        self.last_layer = layers.SoftmaxWithLoss()

    def predict(self, x):
        for key in self.layers.keys():
            x = self.layers[key].forward(x)
        return x

    def loss(self, x, d):
        y = self.predict(x)
        return self.last_layer.forward(y, d)
```

```
        tx = x[i*batch_size:(i+1)*batch_size]
        td = d[i*batch_size:(i+1)*batch_size]
        y = self.predict(tx)
        y = np.argmax(y, axis=1)
        acc += np.sum(y == td)

    return acc / x.shape[0]

def gradient(self, x, d):
    # forward
    self.loss(x, d)

    # backward
    dout = 1
    dout = self.last_layer.backward(dout)
    layers = list(self.layers.values())

    layers.reverse()
    for layer in layers:
        dout = layer.backward(dout)

    # 設定
    grad = {}
    grad['W1'], grad['b1'] = self.layers['Conv1'].dW, self.layers['Conv1'].db
    grad['W2'], grad['b2'] = self.layers['Conv2'].dW, self.layers['Conv2'].db
```

```
# データの読み込み
(x_train, d_train), (x_test, d_test) = load_mnist(flatten=False)

print("データ読み込み完了")
# 処理に時間がかかる場合はデータを削減
x_train, d_train = x_train[:5000], d_train[:5000]
x_test, d_test = x_test[:1000], d_test[:1000]


network = DoubleConvNet(input_dim=(1,28,28),
                        conv_param_1={'filter_num':10, 'filter_size':2, 'pad':1, 'stride':1},
                        conv_param_2={'filter_num':20, 'filter_size':3, 'pad':1, 'stride':1},
                        hidden_size=100, output_size=10, weight_init_std=0.01)

optimizer = optimizer.Adam()

# 時間がかかるため100に設定
iters_num = 100
# iters_num = 1000
train_size = x_train.shape[0]
batch_size = 100

train_loss_list = []
accuracies_train = []
accuracies_test = []
```

```
    batch_mask = np.random.choice(train_size, batch_size)
    x_batch = x_train[batch_mask]
    d_batch = d_train[batch_mask]

    grad = network.gradient(x_batch, d_batch)
    optimizer.update(network.params, grad)
    loss = network.loss(x_batch, d_batch)
    train_loss_list.append(loss)

    if (i+1) % plot_interval == 0:
        accr_train = network.accuracy(x_train, d_train)
        accr_test = network.accuracy(x_test, d_test)
        accuracies_train.append(accr_train)
        accuracies_test.append(accr_test)

        print("Generation: " + str(i+1) + ". 正答率(トレーニング) = " + str(accr_train))
        print("                : " + str(i+1) + ". 正答率(テスト) = " + str(accr_test))

lists = range(0, iters_num, plot_interval)
plt.plot(lists, accuracies_train, label="training set")
plt.plot(lists, accuracies_test,  label="test set")
plt.legend(loc="lower right")
plt.title("accuracy")
plt.xlabel("count")
plt.ylabel("accuracy")
```

```
                : 10. 正答率(テスト) = 0.11
Generation: 20. 正答率(トレーニング) = 0.1302
                : 20. 正答率(テスト) = 0.137
Generation: 30. 正答率(トレーニング) = 0.5008
                : 30. 正答率(テスト) = 0.488
Generation: 40. 正答率(トレーニング) = 0.8136
                : 40. 正答率(テスト) = 0.555
Generation: 50. 正答率(トレーニング) = 0.8298
                : 50. 正答率(テスト) = 0.598
Generation: 60. 正答率(トレーニング) = 0.8844
                : 60. 正答率(テスト) = 0.654
Generation: 70. 正答率(トレーニング) = 0.7574
                : 70. 正答率(テスト) = 0.702
Generation: 80. 正答率(トレーニング) = 0.7626
                : 80. 正答率(テスト) = 0.745
Generation: 90. 正答率(トレーニング) = 0.7856
                : 90. 正答率(テスト) = 0.764
Generation: 100. 正答率(トレーニング) = 0.777
```



[try] DoubleConvNetをアレンジしよう

pooling層を一つ減らしたものに変えてみよう

conv - relu - conv - relu - pool - affine - relu - affine - softmax