



レポート No.2

目次

1. 深層学習（前編 2）	2
1.1. 勾配消失問題	2
1.2. 学習率最適化手法	19
1.3. 過学習	31
1.4. 畳み込みニューラルネットワークの概念	45
1.5. 最新の CNN	55
2. 深層学習（後編 1）	59
2.1. 再帰型ニューラルネットワークの概念	59
2.2. LSTM	73
2.3. GRU (Gated Recurrent Unit)	78
2.4. 双方向 RNN	81
2.5. Seq2Seq RNN での自然言語処理	83
2.6. word2vec	88
2.7. Attention Mechanism	89
3. 深層学習（後編 2）	91
3.1. Tensorflow の実装演習	91
3.2. 強化学習	142
3.3. 参考文献	148

1. 深層学習（前編 2）

1.1. 勾配消失問題

誤差逆伝播法の復習

連鎖率の原理を使い dz/dx を求める

$$z=t^2$$

$$t=x+y$$

$$dz/dx = 2t \cdot 1 = 2t$$

$$= 2(x+y)$$

$$dt/dx = 1$$

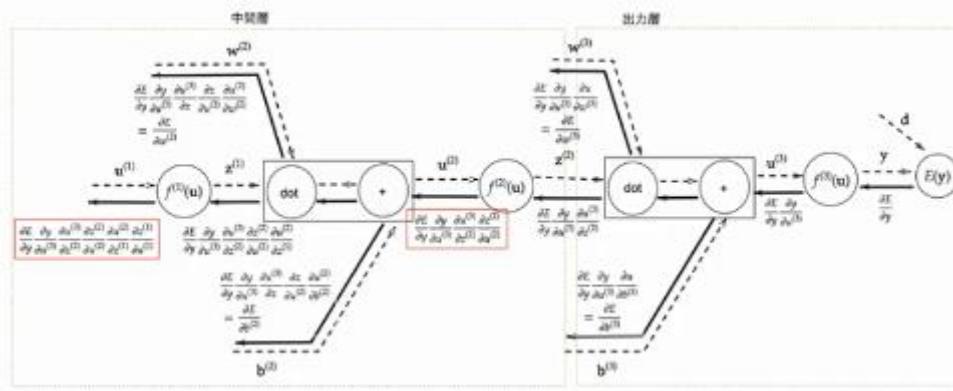
勾配消失問題とは？

誤差逆伝播法が下位層（入力層側）に進んでいくにつれて、

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、

訓練は最適解に収束しなくなる。



活性化関数：シグモイド関数の課題として、大きな値では出力の変化が微小なため、勾配消失問題を引き起こすことがある。

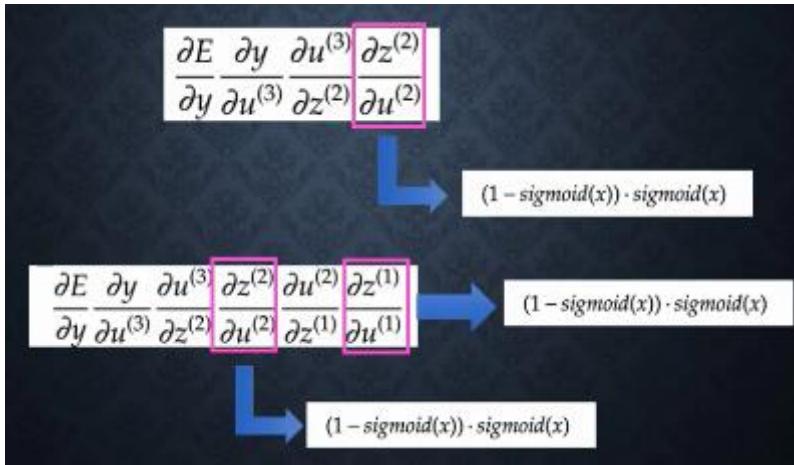
シグモイド関数の微分。入力値がゼロの時に最大値をとる。

その値は、

$$\delta(u)' = (1 - \text{sigmoid}(u)) \cdot \text{sigmoid}(u)$$

$$= 0.5 \times 0.5 = 0.25$$

全体像のピンクの場所



勾配消失の解決方法が 3 つ

- (1) 活性化関数の選択
- (2) 重みの初期値設定
- (3) バッチ正規化 (2015 年～)

1.1.1 活性化関数

活性化関数 : ReLU 関数（もっとも良くつかわれる関数）

勾配消失問題の回避とスペース化に貢献する関数。

1.1.2 初期値の設定方法

重みの初期値設定 : Xavier (ザビエル) :

重みの要素を前の層のノード数の平方根で除算した値

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

重みの初期値設定 : He (エイチイー) :

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て $\sqrt{2}$ を掛け合わせた値

He の初期値を設定する際の活性化関数

- ReLU 関数に特化

重みの初期値にゼロを設定するとどうなるか？

すべての値が同じ値で伝わるためパラメータのチューニングが行われなくなる。

1.1.3 バッチ正規化

2015年～、最近のもの。

バッチ正則化とは？

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

バッチ正規化の使い所

活性化関数に値を渡す前後に、バッチ正規化の処理を含んだ層を加える。

バッチ正規化の効果

(1)計算の高速化（データの正規化によりコンパクトになるため）

(2)勾配消失が起きづらくなる

順伝播でのバッチ正規化の数学的記述

$$1. \mu_t = \frac{1}{N_t} \sum_{i=1}^{N_t} x_{ni}$$
$$2. \sigma_t^2 = \frac{1}{N_t} \sum_{i=1}^{N_t} (x_{ni} - \mu_t)^2$$
$$3. \hat{x}_{ni} = \frac{x_{ni} - \mu_t}{\sqrt{\sigma_t^2 + \theta}}$$
$$4. y_{ni} = \gamma x_{ni} + \beta$$

μ_t : ミニバッチt全体の平均
 σ_t^2 : ミニバッチt全体の標準偏差
 N_t : ミニバッチのインデックス
 \hat{x}_{ni} : 0に値を近づける計算(0を中心とするセンタリング)と正規化を施した値
 γ : スケーリングパラメータ
 β : シフトパラメータ
 y_{ni} : ミニバッチのインデックス値とスケーリングの積にシフトを加算した値(バッチ正規化オペレーションの出力)

\hat{x} が正規化されているところ

γ と β は逆伝播の場合は学習していく。最適な値に変えていく。

ハンズオン（実装演習）

Jupyter 2_2_1_vanishing_gradient Last Checkpoint: 2018/11/24 (autosaved) Logout

File Edit View Insert Cell Kernel Widgets Help Trusted Python 3

vanishing gradient

sigmoid - gauss

```
In [1]: import sys, os
sys.path.append(os.pardir) # 網ディレクトリのファイルをインポートするための設定
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

# mnistをロード
(x_train, d_train), (x_test, d_test) = load_mnist(normalize=True, one_hot_label=True)
train_size = len(x_train)

print("データ読み込み完了")

# 重み初期値補正係数
weight_init = 0.01
# 入力層サイズ
input_layer_size = 784
# 中間層サイズ
hidden_layer_1_size = 40
hidden_layer_2_size = 20

# 出力層サイズ
output_layer_size = 10
# 繰り返し数
iters_num = 2000
# ミニバッチサイズ
batch_size = 100
# 学習率
learning_rate = 0.1
# 描写頻度
plot_interval=10
```

Jupyter 2_2_1_vanishing_gradient Last Checkpoint: 2018/11/24 (autosaved)

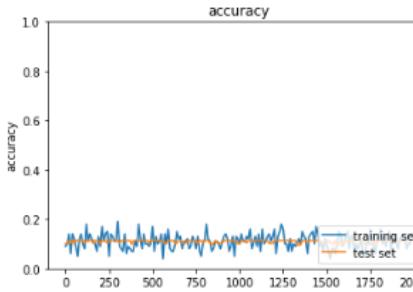
File Edit View Insert Cell Kernel Widgets Help Trusted Python 3

```
print('Generation: %d. 正答率(トレーニング) = %f' % (i+1, accr_train))
print('Generation: %d. 正答率(テスト) = %f' % (i+1, accr_test))

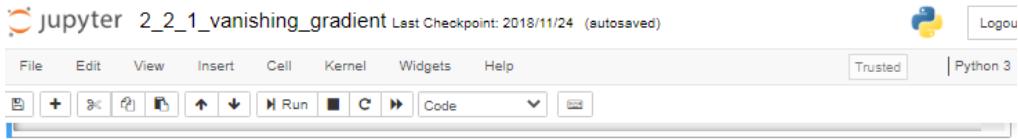
# パラメータに勾配適用
for key in ('W1', 'W2', 'W3', 'b1', 'b2', 'b3'):
    network[key] -= learning_rate * grad[key]

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()
```

Generation: 2000. 正答率(トレーニング) = 0.14
: 2000. 正答率(テスト) = 0.1135



sigmoid 関数で勾配消失が発生



ReLU - gauss

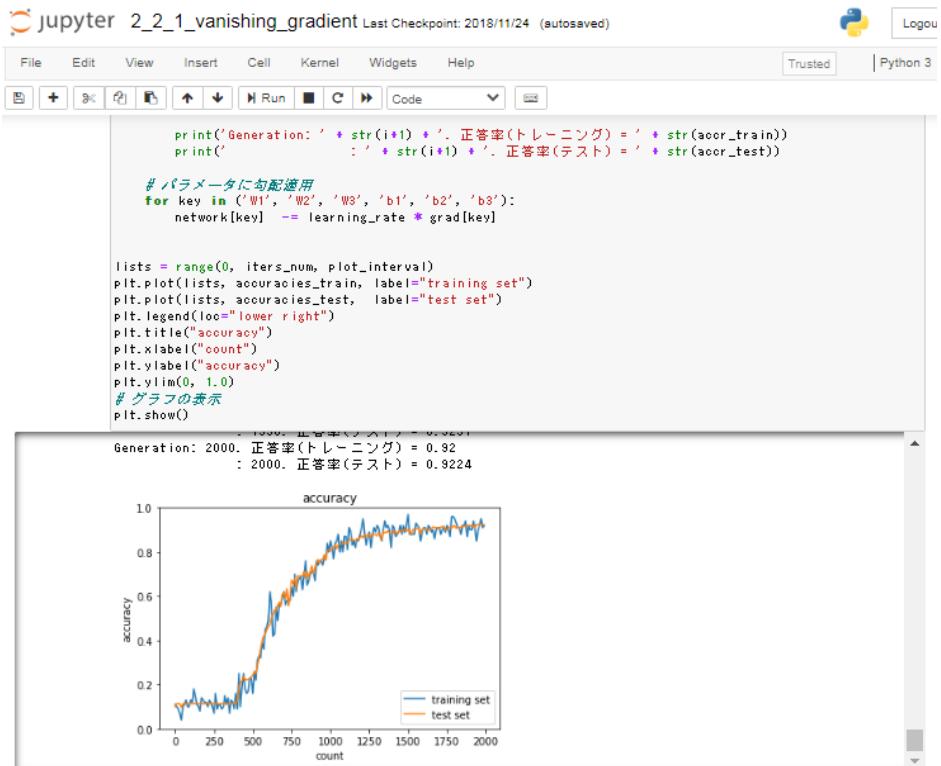
```
In [2]: import sys, os
sys.path.append(os.pardir) # 綱ディレクトリのファイルをインポートするための設定
import numpy as np
from data.mnist import load_mnist
from PIL import Image
import pickle
from common import functions
import matplotlib.pyplot as plt

# mnistをロード
(x_train, d_train), (x_test, d_test) = load_mnist(normalize=True, one_hot_label=True)
train_size = len(x_train)

print("データ読み込み完了")

# 重み初期値補正係数
weight_init = 0.01
# 入力層サイズ
input_layer_size = 784
# 中間層サイズ
hidden_layer_1_size = 40
hidden_layer_2_size = 20

# 出力層サイズ
output_layer_size = 10
# 繰り返し数
iters_num = 2000
# ミニバッチサイズ
batch_size = 100
# 学習率
learning_rate = 0.1
# 描写頻度
plot_interval=10
```



ReLU 関数で、勾配消失が起きにくい状況になった。学習が進んでいる。

Jupyter 2_2_1_vanishing_gradient Last Checkpoint: 2018/11/24 (autosaved) Logout

File Edit View Insert Cell Kernel Widgets Help Trusted Python 3

sigmoid - Xavier

```
In [3]: import sys, os
sys.path.append(os.pardir) # 父ディレクトリのファイルをインポートするための設定
import numpy as np
from data.mnist import load_mnist
from PIL import Image
import pickle
from common import functions
import matplotlib.pyplot as plt

# mnistをロード
(x_train, d_train), (x_test, d_test) = load_mnist(normalize=True, one_hot_label=True)
train_size = len(x_train)

print("データ読み込み完了")

# 入力層サイズ
input_layer_size = 784
# 中間層サイズ
hidden_layer_1_size = 40
hidden_layer_2_size = 20
# 出力層サイズ
output_layer_size = 10
# 繰り返し数
iters_num = 2000
# ミニバッチサイズ
batch_size = 100
# 学習率
learning_rate = 0.1
# 描写頻度
plot_interval=10

# 初期設定
def init_network():
    network = {}

    ##### 変更箇所 #####
    # Xavierの初期値
    network['W1'] = np.random.randn(input_layer_size, hidden_layer_1_size) / (np.sqrt(input_layer_size))
```

jupyter 2_2_1_vanishing_gradient Last Checkpoint: 2018/11/24 (autosaved)

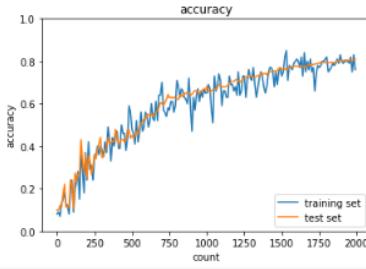
File Edit View Insert Cell Kernel Widgets Help Trusted Python 3

```
print('Generation: ' + str(i+1) + ', 正答率(トレーニング) = ' + str(acor_train))
print('                : ' + str(i+1) + ', 正答率(テスト) = ' + str(acor_test))

# パラメータに勾配適用
for key in ('W1', 'W2', 'W3', 'b1', 'b2', 'b3'):
    network[key] -= learning_rate * grad[key]

lists = range(0, iter_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()
```

Generation: 2000, 正答率(トレーニング) = 0.76
: 2000, 正答率(テスト) = 0.8077



Xavier を用いた初期値で、sigmoid 関数を利用して勾配消失は起きていない

jupyter 2_2_1_vanishing_gradient Last Checkpoint: 2018/11/24 (autosaved) Logout

File Edit View Insert Cell Kernel Widgets Help Trusted Python 3

ReLU - He

```
In [4]: import sys, os
sys.path.append(os.pardir) # 父ディレクトリのファイルをインポートするための設定
import numpy as np
from data.mnist import load_mnist
from PIL import Image
import pickle
from common import functions
import matplotlib.pyplot as plt

# mnistをロード
(x_train, d_train), (x_test, d_test) = load_mnist(normalize=True, one_hot_label=True)
train_size = len(x_train)

print("データ読み込み完了")

# 重み初期値
wieght_init = 0.01
# 入力層サイズ
input_layer_size = 784
# 中間層サイズ
hidden_layer_1_size = 40
hidden_layer_2_size = 20

# 出力層サイズ
output_layer_size = 10
# 繰り返し数
iters_num = 2000
# ミニバッチサイズ
batch_size = 100
# 学習率
learning_rate = 0.1
# 描写頻度
plot_interval=10

# 初期設定
def init_network():
    network = {}
```

Jupyter 2_2_1_vanishing_gradient Last Checkpoint: 2018/11/24 (autosaved)

File Edit View Insert Cell Kernel Widgets Help Trusted Python 3

```
print('Generation: ' + str(i+1) + ' 正答率(トレーニング) = ' + str(acor_train))
print('Generation: ' + str(i+1) + ' 正答率(テスト) = ' + str(acor_test))

# パラメータに勾配適用
for key in ('W1', 'W2', 'W3', 'b1', 'b2', 'b3'):
    network[key] -= learning_rate * grad[key]

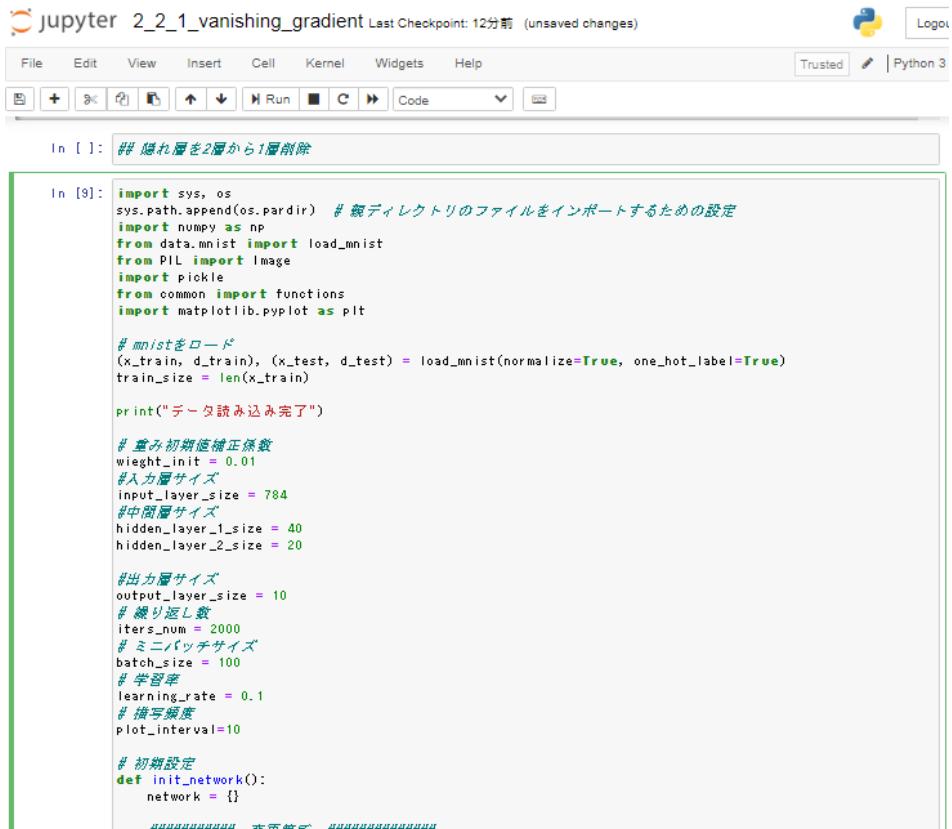
lists = range(0, iter_num, plot_interval)
plt.plot(lists, accuracies_train, label="training set")
plt.plot(lists, accuracies_test, label="test set")
plt.legend(loc="lower right")
plt.title("accuracy")
plt.xlabel("count")
plt.ylabel("accuracy")
plt.ylim(0, 1.0)
# グラフの表示
plt.show()
```

Generation: 2000. 正答率(トレーニング) = 0.94
: 2000. 正答率(テスト) = 0.9563

He を用いた初期値で、ReLU 関数を利用して勾配消失は起きていない

(1) 隠れ層の変更

He を用いた初期値で、ReLU 関数を利用したもので、
隠れ層を 2 層から 1 層に変更。



The screenshot shows a Jupyter Notebook interface with the title "jupyter 2_2_1_vanishing_gradient". The toolbar includes File, Edit, View, Insert, Cell, Kernel, Widgets, Help, Trusted, and Python 3. The code cell (In [1]) contains the following Python script:

```
import sys, os
sys.path.append(os.pardir) # 親ディレクトリのファイルをインポートするための設定
import numpy as np
from data.mnist import load_mnist
from PIL import Image
import pickle
from common import functions
import matplotlib.pyplot as plt

# mnistをロード
(x_train, d_train), (x_test, d_test) = load_mnist(normalize=True, one_hot_label=True)
train_size = len(x_train)

print("データ読み込み完了")

# 重み初期値
weight_init = 0.01
# 入力層サイズ
input_layer_size = 784
# 中間層サイズ
hidden_layer_1_size = 40
hidden_layer_2_size = 20

# 出力層サイズ
output_layer_size = 10
# 繰り返し数
iters_num = 2000
# ミニバッチサイズ
batch_size = 100
# 学習率
learning_rate = 0.1
# 描写頻度
plot_interval=10

# 初期設定
def init_network():
    network = {}

    *******/

    return network
```

Jupyter 2_2_1_vanishing_gradient Last Checkpoint: 12分前 (unsaved changes)

```
File Edit View Insert Cell Kernel Widgets Help Trusted Python 3
```

```
network = {}

#####
# 变更箇所 #####
#####

# Heの初期値
network['W1'] = np.random.randn(input_layer_size, hidden_layer_1_size) / np.sqrt(input_layer_size) * n
network['W2'] = np.random.randn(hidden_layer_1_size, hidden_layer_2_size) / np.sqrt(hidden_layer_1_size)
network['W3'] = np.random.randn(hidden_layer_2_size, output_layer_size) / np.sqrt(hidden_layer_2_size)
network['b3'] = np.random.randn(hidden_layer_1_size, output_layer_size) / np.sqrt(hidden_layer_2_size)

#####
# 誤差逆伝播
def forward(network, x):
    # W1, W2, W3 = network['W1'], network['W2'], network['W3']
    # b1, b2, b3 = network['b1'], network['b2'], network['b3']
    W1, W3 = network['W1'], network['W3']
    b1, b3 = network['b1'], network['b3']

    #####
# 变更箇所 #####
#####

    hidden_f = functions.relu

    #####
# 誤差逆伝播
    u1 = np.dot(x, W1) + b1
    z1 = hidden_f(u1)
    #u2 = np.dot(z1, W2) + b2
    #z2 = hidden_f(u2)
    #u3 = np.dot(z2, W3) + b3
    u3 = np.dot(z1, W3) + b3
    y = functions.softmax(u3)

    #return z1, z2, y
    return z1, y

# 誤差逆伝播
```

Jupyter 2_2_1_vanishing_gradient Last Checkpoint: 12分前 (unsaved changes)

```
File Edit View Insert Cell Kernel Widgets Help Trusted Python 3
```

```
return z1, y

# 誤差逆伝播
def backward(x, d, z1, z2, y):
    grad = {}

    # W1, W2, W3 = network['W1'], network['W2'], network['W3']
    # b1, b2, b3 = network['b1'], network['b2'], network['b3']
    W1, W3 = network['W1'], network['W3']
    b1, b3 = network['b1'], network['b3']

    #####
# 变更箇所 #####
#####

    hidden_d_f = functions.d_relu

    #####
# 出力層でのデルタ
    delta3 = functions.d_softmax_with_loss(d, y)
    # b3の勾配
    grad['b3'] = np.sum(delta3, axis=0)
    # W3の勾配
    grad['W3'] = np.dot(z2.T, delta3)
    grad['W3'] = np.dot(z1.T, delta3)
    # 2層でのデルタ
    delta2 = np.dot(delta3, W3.T) * hidden_d_f(z2)
    # b2の勾配
    grad['b2'] = np.sum(delta2, axis=0)
    # W2の勾配
    grad['W2'] = np.dot(z1.T, delta2)
    # 1層でのデルタ
    delta1 = np.dot(delta2, W2.T) * hidden_d_f(z1)
    delta1 = np.dot(delta3, W3.T) * hidden_d_f(z1)
    # b1の勾配
    grad['b1'] = np.sum(delta1, axis=0)
    # W1の勾配
    grad['W1'] = np.dot(x.T, delta1)

    return grad

# パラメータの初期化
network = init_network()
```

jupyter 2_2_1_vanishing_gradient Last Checkpoint: 12分前 (unsaved changes) Logout

File Edit View Insert Cell Kernel Widgets Help Trusted Python 3

```
# パラメータの初期化
network = init_network()

accuracies_train = []
accuracies_test = []

# 正答率
def accuracy(x, d):
    z1, z2, y = forward(network, x)
    z1, y = forward(network, 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

for i in range(iters_num):
    # ランダムにバッチを取得
    batch_mask = np.random.choice(train_size, batch_size)
    # ミニバッチに対応する教師訓練画像データを取得
    x_batch = x_train[batch_mask]
    # ミニバッチに対応する訓練正解ラベルデータを取得する
    d_batch = d_train[batch_mask]

    #z1, z2, y = forward(network, x_batch)
    z1, y = forward(network, x_batch)
    #grad = backward(x_batch, d_batch, z1, z2, y)
    grad = backward(x_batch, d_batch, z1, y)

    if (i+1)%plot_interval==0:
        accr_test = accuracy(x_test, d_test)
        accuracies_test.append(accr_test)

        accr_train = accuracy(x_batch, d_batch)
        accuracies_train.append(accr_train)

        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))

    # パラメータに勾配適用
    #for key in ('W1', 'W2', 'W3', 'b1', 'b2', 'b3'):
```

Jupyter 2_2_1_vanishing_gradient Last Checkpoint: 12分前 (unsaved changes) Log

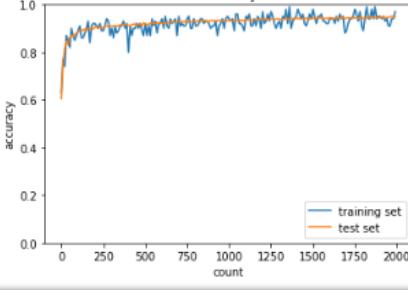
File Edit View Insert Cell Kernel Widgets Help Trusted Python

```
print('Generation: ' + str(i+1) + ', 正答率(トレーニング) = ' + str(acor_train))
print('           : ' + str(i+1) + ', 正答率(テスト) = ' + str(acor_test))

# パラメータに勾配適用
# for key in ('W1', 'W2', 'W3', 'b1', 'b2', 'b3'):
for key in ('W1', 'W2', 'b1', 'b2', 'b3'):
    network[key] -= learning_rate * grad[key]

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()
```

Generation: 2000. 正答率(トレーニング) = 0.9485
: 2000. 正答率(テスト) = 0.9485



層を1層削除してもこのデータでは少し正答率が落ちる程度で
大きな差は見られない。

(2) ReLU で Xavier

The screenshot shows a Jupyter Notebook interface with the title "jupyter 2_2_1_vanishing_gradient". The notebook has a single cell labeled "In [7]". The code in the cell is as follows:

```
## ReLU - Xavier

import sys, os
sys.path.append(os.pardir) # 親ディレクトリのファイルをインポートするための設定
import numpy as np
from data.mnist import load_mnist
from PIL import Image
import pickle
from common import functions
import matplotlib.pyplot as plt

# mnistをロード
(x_train, d_train), (x_test, d_test) = load_mnist(normalize=True, one_hot_label=True)
train_size = len(x_train)

print("データ読み込み完了")

# 重み初期値補正係数
weight_init = 0.01
# 入力層サイズ
input_layer_size = 784
# 中間層サイズ
hidden_layer_1_size = 40
hidden_layer_2_size = 20

# 出力層サイズ
output_layer_size = 10
# 繰り返し数
iters_num = 2000
# ミニバッチサイズ
batch_size = 100
# 学習率
learning_rate = 0.1
# 描写頻度
plot_interval=10

# 初期設定
def init_network():
    network = {}

    ##### 変更箇所 #####
    return network
```

```

jupyter 2_2_1_vanishing_gradient Last Checkpoint: 1分前 (autosaved) Logout
File Edit View Insert Cell Kernel Widgets Help Trusted Python 3










```

#####
Xavierの初期値
#####
network['W1'] = np.random.randn(input_layer_size, hidden_layer_1_size) / (np.sqrt(input_layer_size))
network['W2'] = np.random.randn(hidden_layer_1_size, hidden_layer_2_size) / (np.sqrt(hidden_layer_1_size))
network['W3'] = np.random.randn(hidden_layer_2_size, output_layer_size) / (np.sqrt(hidden_layer_2_size))

#####
network['b1'] = np.zeros(hidden_layer_1_size)
network['b2'] = np.zeros(hidden_layer_2_size)
network['b3'] = np.zeros(output_layer_size)

return network

誤伝播
def forward(network, x):
 W1, W2, W3 = network['W1'], network['W2'], network['W3']
 b1, b2, b3 = network['b1'], network['b2'], network['b3']

#####
誤差逆伝播
#####
hidden_f = functions.relu

#####
u1 = np.dot(x, W1) + b1
z1 = hidden_f(u1)
u2 = np.dot(z1, W2) + b2
z2 = hidden_f(u2)
u3 = np.dot(z2, W3) + b3
y = functions.softmax(u3)

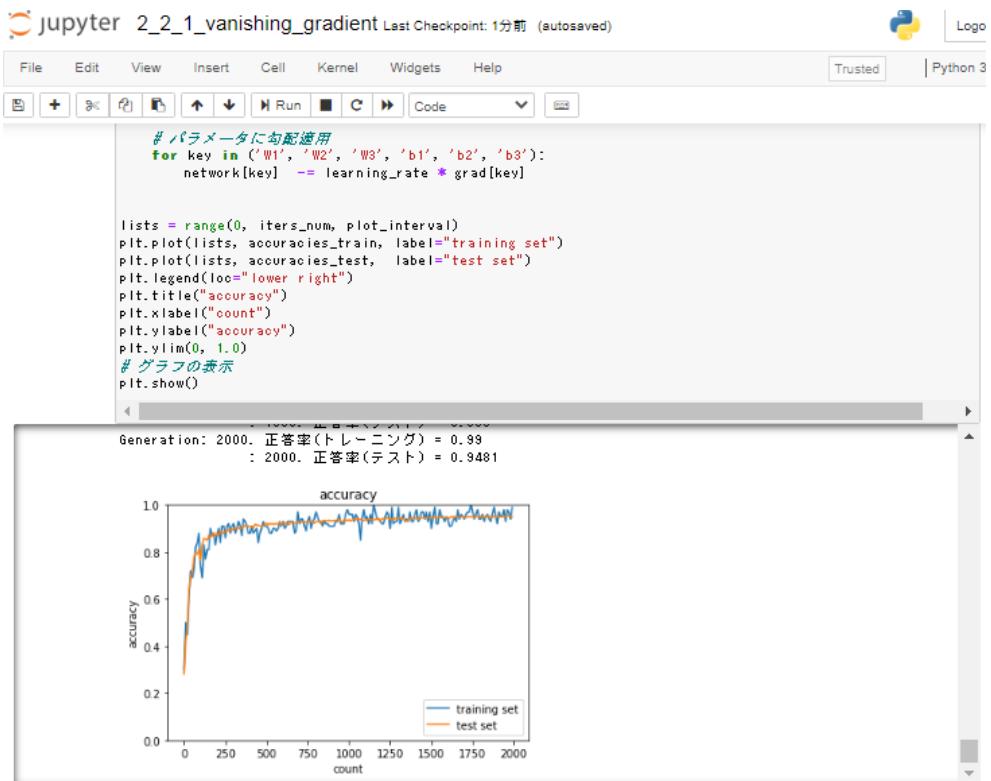
return z1, z2, y

誤差逆伝播
def backward(x, d, z1, z2, y):
 grad = {}

 W1, W2, W3 = network['W1'], network['W2'], network['W3']
 b1, b2, b3 = network['b1'], network['b2'], network['b3']

```


```



Xavier を用いた初期値で、ReLU 関数を利用しても勾配消失は起きていない

(3) シグモイドで He

Jupyter 2_2_1_vanishing_gradient Last Checkpoint: 2018/11/24 (unsaved changes)

File Edit View Insert Cell Kernel Widgets Help Trusted Python 3

In [5]: `## sigmoid - He`

```
In [6]: import sys, os
sys.path.append(os.pardir) # 親ディレクトリのファイルをインポートするための設定
import numpy as np
from data.mnist import load_mnist
from PIL import Image
import pickle
from common import functions
import matplotlib.pyplot as plt

# mnistをロード
(x_train, d_train), (x_test, d_test) = load_mnist(normalize=True, one_hot_label=True)
train_size = len(x_train)

print("データ読み込み完了")

# 入力層サイズ
input_layer_size = 784
# 中間層サイズ
hidden_layer_1_size = 40
hidden_layer_2_size = 20
# 出力層サイズ
output_layer_size = 10
# 繰り返し数
iters_num = 2000
# バッチサイズ
batch_size = 100
# 学習率
learning_rate = 0.1
# 描写頻度
plot_interval=10

# 初期設定
def init_network():
    network = {}

    ##### 変更箇所 #####
    # Heの初期値
    network['W1'] = np.random.randn(input_layer_size, hidden_layer_1_size) / np.sqrt(input_layer_size) * np.sqrt(2)
    network['W2'] = np.random.randn(hidden_layer_1_size, hidden_layer_2_size) / np.sqrt(hidden_layer_1_size) * np.sqrt(2)
    network['W3'] = np.random.randn(hidden_layer_2_size, output_layer_size) / np.sqrt(hidden_layer_2_size) * np.sqrt(2)

    ##### 変更箇所 #####
    network['b1'] = np.zeros(hidden_layer_1_size)
    network['b2'] = np.zeros(hidden_layer_2_size)
    network['b3'] = np.zeros(output_layer_size)

    return network

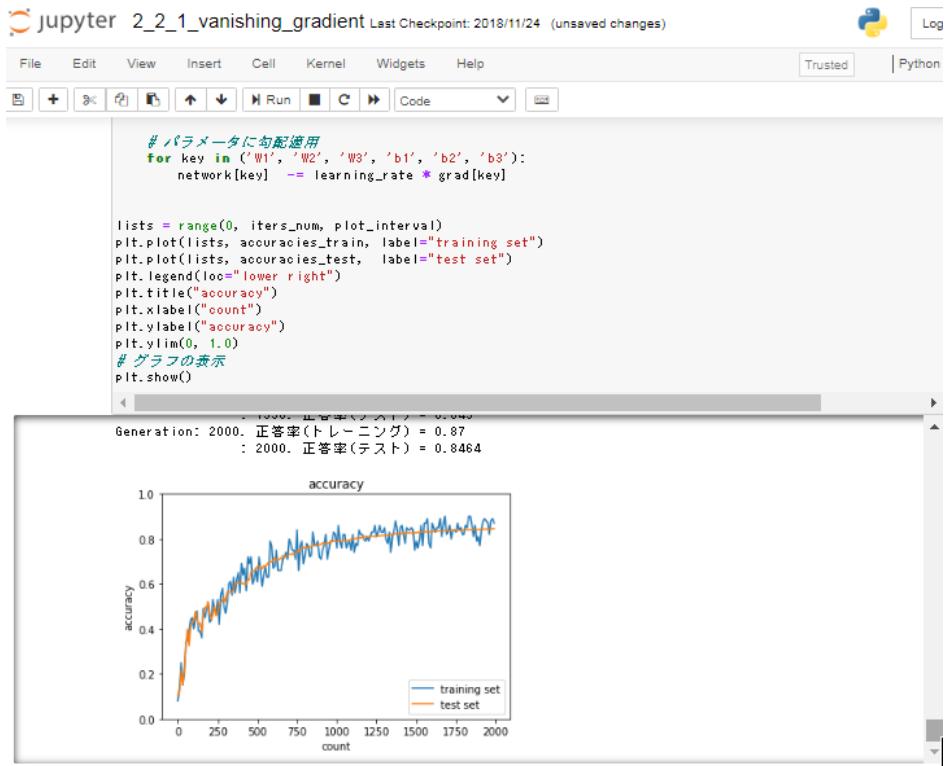
# 前伝播
def forward(network, x):
    W1, W2, W3 = network['W1'], network['W2'], network['W3']
    b1, b2, b3 = network['b1'], network['b2'], network['b3']
    hidden_f = functions.sigmoid

    v1 = np.dot(x, W1) + b1
    z1 = hidden_f(v1)
    v2 = np.dot(z1, W2) + b2
    z2 = hidden_f(v2)
    v3 = np.dot(z2, W3) + b3
    y = functions.softmax(v3)

    return z1, z2, y

# 逆差伝播
def backward(x, d, z1, z2, y):
    grad = []

    W1, W2, W3 = network['W1'], network['W2'], network['W3']
    b1, b2, b3 = network['b1'], network['b2'], network['b3']
    hidden_d_f = functions.d_sigmoid
```



He を用いた初期値で、sigmoid 関数を利用して勾配消失は起きにくくなる

データの集め方：ユーザーがデータを登録するためのプロダクトを作ると良い

1.2. 学習率最適化手法

学習率の復習

学習率が大きい場合：最適値にいつまでもたどり着かず、発散

学習率が小さい場合：発散はないが、小さすぎると終息まで時間がかかる

大域局所最適値に収束しづらくなる

学習率の決め方：大きめに設定し、徐々に小さくする

パラメータ毎に学習率を可変させる

1.2.1 モメンタム

【モメンタム】	【勾配降下法】
$V_t = \mu V_{t-1} - \epsilon \nabla E$	$\mathbf{w}^{(t+1)} = \mathbf{w}^{(t)} - \epsilon \nabla E$
$\mathbf{w}^{(t+1)} = \mathbf{w}^{(t)} + V_t$	
慣性: μ	
モメンタム	勾配降下法
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	誤差をパラメータで微分したものと学習率の積を減算する

慣性の値：ハイパーパラメータ 0.05～0.09 の間が一般的

モメンタムのメリット

- ・局所的最適解にならず、大域的最適解になる。
- ・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。

$V_t = \mu V_{t-1} - \epsilon \nabla E$
<code>self.v[key] = self.momentum * self.v[key] - self.learning_rate * grad[key]</code>
$\mathbf{w}^{(t+1)} = \mathbf{w}^{(t)} + V_t$
<code>params[key] += self.v[key]</code>
慣性: μ

1.2.2 AdaGrad

勾配のゆるやかな斜面に対して有効、最適値に近づける

【AdaGrad】	【勾配降下法】
$h_0 = \theta$	$\mathbf{w}^{(t+1)} = \mathbf{w}^{(t)} - \epsilon \nabla E$
$h_t = h_{t-1} + (\nabla E)^2$	
$\mathbf{w}^{(t+1)} = \mathbf{w}^{(t)} - \epsilon \frac{1}{\sqrt{h_t + \theta}} \nabla E$	
AdaGrad	勾配降下法
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	誤差をパラメータで微分したものと学習率の積を減算する

問題：学習率が徐々に少なくなるので、鞍点問題が起きる事があった。

この問題を解決するのは RMSProp

```


$$h_0 = \theta$$

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)

```

1.2.3 RMSProp

【RMSProp】		【勾配降下法】
$h_t = \alpha h_{t-1} + (1 - \alpha) (\nabla E)^2$		$\mathbf{w}^{(t+1)} = \mathbf{w}^{(t)} - \epsilon \nabla E$
$\mathbf{w}^{(t+1)} = \mathbf{w}^{(t)} - \epsilon \frac{1}{\sqrt{h_t + \theta}} \nabla E$		
RMSProp		勾配降下法
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		誤差をパラメータで微分したものと学習率の積を減算する

ハイパーパラメータの調整が少なくてすむ。

局所的最適解にはならず、大域的最適解となる。

```


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

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

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

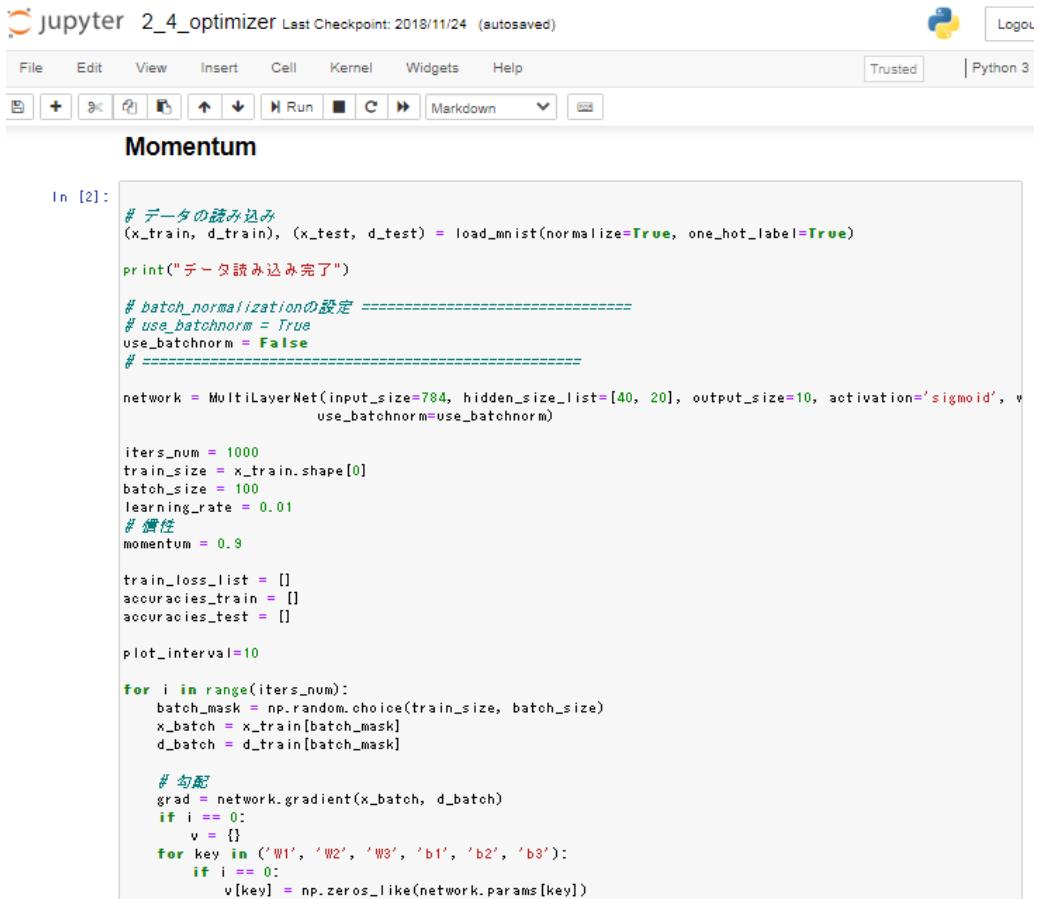
params[key] -= self.learning_rate * grad[key] / (np.sqrt(self.h[key]) + 1e-7)

```

1.2.4 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
- 上記をそれぞれ含んだ最適化アルゴリズムとなっている。
- モメンタムと RMSProp のメリットを含む。
 - ハイパーパラメータの調整が少ない。

ハンズオン（実装演習）



The screenshot shows a Jupyter Notebook interface with the title "jupyter 2_4_optimizer Last Checkpoint: 2018/11/24 (autosaved)". The notebook has tabs for File, Edit, View, Insert, Cell, Kernel, Widgets, Help, Trusted, and Python 3. Below the tabs is a toolbar with icons for file operations like Open, Save, and Run, and a dropdown for Markdown. The main area is titled "Momentum". A code cell labeled "In [2]" contains the following Python code:

```
# 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")

# batch_normalizationの設定 =====
# use_batchnorm = True
use_batchnorm = False
# =====

network = MultiLayerNet(input_size=784, hidden_size_list=[40, 20], output_size=10, activation='sigmoid', w_init_std=0.01, use_batchnorm=use_batchnorm)

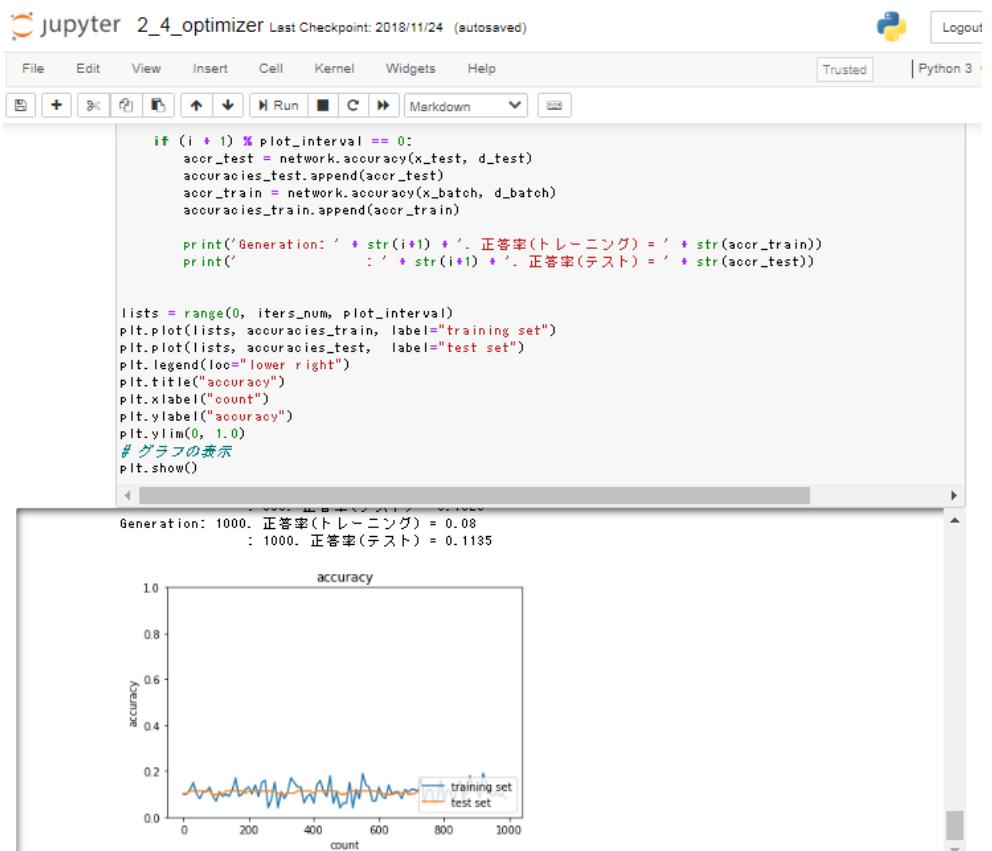
iters_num = 1000
train_size = x_train.shape[0]
batch_size = 100
learning_rate = 0.01
# 構造
momentum = 0.9

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)
    if i == 0:
        v = {}
    for key in ('W1', 'W2', 'W3', 'b1', 'b2', 'b3'):
        if i == 0:
            v[key] = np.zeros_like(network.params[key])
        else:
            v[key] = momentum * v[key] + (1 - momentum) * grad[key]
    network.params = v
```



Momentum で学習率を 0.01⇒0.0001

The screenshot shows a Jupyter Notebook interface with the title "Momentum". The code in cell [6] is as follows:

```
# 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")

# batch_normalizationの設定 =====
# use_batchnorm = True
use_batchnorm = False
# =====

network = MultiLayerNet(input_size=784, hidden_size_list=[40, 20], output_size=10, activation='sigmoid', v
use_batchnorm=use_batchnorm)

iters_num = 1000
train_size = x_train.shape[0]
batch_size = 100
learning_rate = 0.0001
# 慣性
momentum = 0.9

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)
    if i == 0:
        v = {}
    for key in ('W1', 'W2', 'W3', 'b1', 'b2', 'b3'):
        if i == 0:
            v[key] = np.zeros_like(network.params[key])
        else:
            v[key] = momentum * v[key] + (1 - momentum) * grad[key]
    network.params = v
```

jupyter 2_4_optimizer Last Checkpoint: 2018/11/24 (unsaved changes)  

File Edit View Insert Cell Kernel Widgets Help Trusted Python 3

```
File Edit View Insert Cell Kernel Widgets Help Trusted Python 3
```

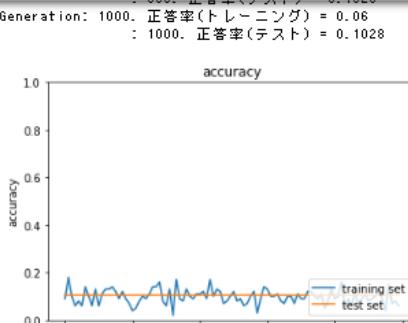
Code

```
plot_interval = 100
acor_train = network.accuracy(x_train, d_train)
accuracies_train.append(acor_train)
acor_test = network.accuracy(x_test, d_test)
accuracies_test.append(acor_test)
acor_train = network.accuracy(x_batch, d_batch)
accuracies_train.append(acor_train)

print('Generation: ' + str(i+1) + '. 正答率(トレーニング) = ' + str(acor_train))
print('          : ' + str(i+1) + '. 正答率(テスト) = ' + str(acor_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")
plt.ylim(0, 1.0)
# グラフの表示
plt.show()
```

Generation: 1000. 正答率(トレーニング) = 0.06
: 1000. 正答率(テスト) = 0.1028



Momentum で学習率を $0.01 \Rightarrow 0.0001$ に変えても大きく変わらない。

AdaGrad の実装

Jupyter 2_4_optimizer Last Checkpoint: 数秒前 (autosaved)

File Edit View Insert Cell Kernel Widgets Help Trusted Python 3 Logout

MomentumをもとにAdaGradを作つてみよう

$\theta = 1e-4$ とする

```
In [7]: # AdaGradをつくりよう
# 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")

# batch_normalizationの設定 =====
# use_batchnorm = True
use_batchnorm = False
# =====

network = MultiLayerNet(input_size=784, hidden_size_list=[40, 20], output_size=10, activation='sigmoid', \
    use_batchnorm=use_batchnorm)

iters_num = 1000
# iters_num = 500 # 处理を短縮

train_size = x_train.shape[0]
batch_size = 100
learning_rate = 0.01

# AdaGradでは不要
# =====

momentum = 0.9

# =====

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

plot_interval=10

for i in range(iters_num):
    # ここから下はAdaGradの部分
    # =====
```

Jupyter 2_4_optimizer Last Checkpoint: 数秒前 (autosaved)

File Edit View Insert Cell Kernel Widgets Help Trusted Python 3 Logout

```
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)
    if i == 0:
        h = {}
    for key in ('W1', 'W2', 'W3', 'b1', 'b2', 'b3'):

        # 変更しよう
        # =====
        if i == 0:
            h[key] = np.full_like(network.params[key], 1e-4)
        else:
            h[key] += np.square(grad[key])
        network.params[key] -= learning_rate * grad[key] / (np.sqrt(h[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)

        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")
plt.ylim(0, 1.0)
```

jupyter 2_4_optimizer Last Checkpoint: 数秒前 (autosaved)

File Edit View Insert Cell Kernel Widgets Help Trusted Logo Python 3

```
acor_test = network.accuracy(x_test, d_test)
accuracies_test.append(acor_test)
acor_train = network.accuracy(x_batch, d_batch)
accuracies_train.append(acor_train)

print('Generation: ' + str(i+1) + '. 正答率(トレーニング) = ' + str(acor_train))
print('           : ' + str(i+1) + '. 正答率(テスト) = ' + str(acor_test))

lists = range(0, iter_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()
```

Generation: 1000. 正答率(トレーニング) = 0.04
 : 1000. 正答率(テスト) = 0.098

The figure is a line graph titled "accuracy". The x-axis is labeled "count" and ranges from 0 to 1000 with major ticks every 200 units. The y-axis is labeled "accuracy" and ranges from 0.0 to 1.0 with major ticks every 0.2 units. There are two data series: "training set" represented by a blue line, and "test set" represented by an orange line. The training set line shows high-frequency oscillations between 0.0 and 0.2. The test set line starts at approximately 0.05 and increases steadily, reaching about 0.1 at the end of the 1000 iterations.

Jupyter 2_4_optimizer Last Checkpoint: 5分前 (autosaved) Logout

File Edit View Insert Cell Kernel Widgets Help Trusted Python 3 C

Run Markdown

RSMprop

```
In [4]:  
# 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")  
  
# batch_normalizationの設定 =====  
# use_batchnorm = True  
use_batchnorm = False  
# =====  
  
network = MultiLayerNet(input_size=784, hidden_size_list=[40, 20], output_size=10, activation='sigmoid', w_init_std=0.01, use_batchnorm=use_batchnorm)  
  
iters_num = 1000  
train_size = x_train.shape[0]  
batch_size = 100  
learning_rate = 0.01  
decay_rate = 0.99  
  
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)  
    if i == 0:  
        h = {}  
    for key in ('W1', 'W2', 'W3', 'b1', 'b2', 'b3'):  
        if i == 0:  
            h[key] = np.zeros_like(network.params[key])
```

jupyter 2_4_optimizer Last Checkpoint: 5分前 (autosaved)

File Edit View Insert Cell Kernel Widgets Help Trusted Python 3

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

    print('Generation: ' + str(i+1) + ', 正答率(トレーニング) = ' + str(accor_train))
    print('                : ' + str(i+1) + ', 正答率(テスト) = ' + str(accor_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")
plt.ylim(0, 1.0)
# グラフの表示
plt.show()
```

Generation: 1000. 正答率(トレーニング) = 0.98
: 1000. 正答率(テスト) = 0.9449

The graph displays two data series: 'training set' (blue line) and 'test set' (orange line). Both series show an overall increasing trend in accuracy over 1000 iterations. The training set accuracy reaches approximately 0.9449 at iteration 1000, while the test set accuracy reaches approximately 0.9449.

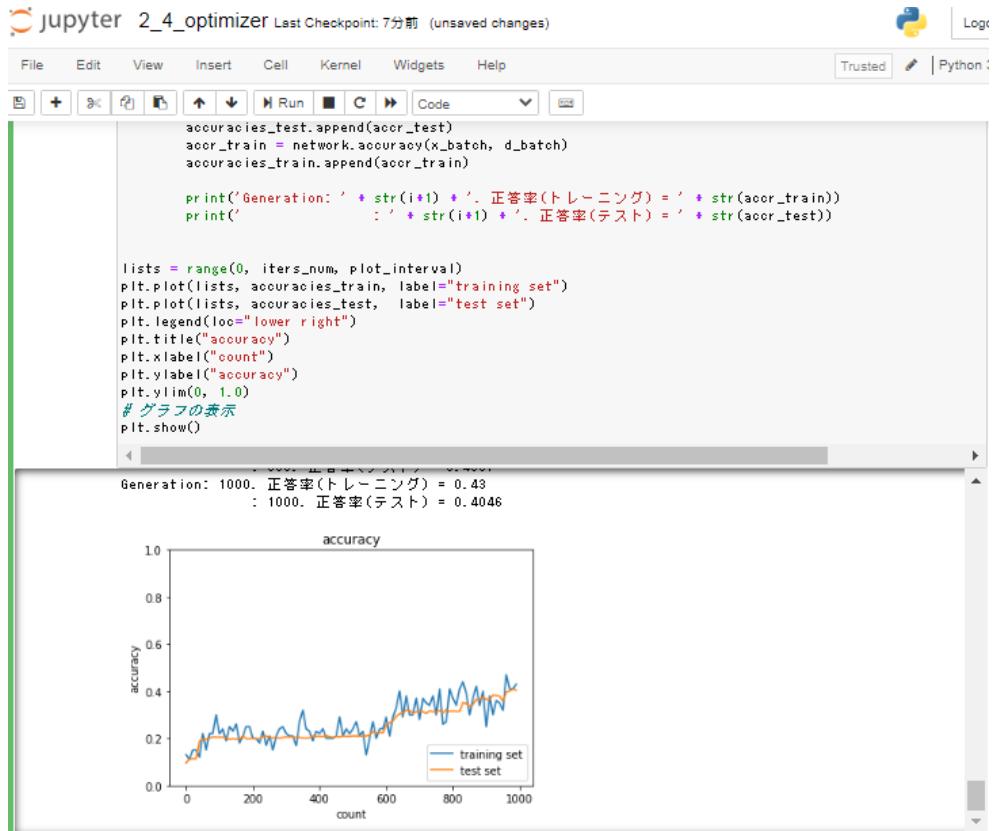
Jupyter 2_4_optimizer Last Checkpoint: 7分前 (unsaved changes)  Logo

File Edit View Insert Cell Kernel Widgets Help Trusted | Python 3

Run Cell Code

RSMprop

```
In [9]:  
# 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")  
  
# batch_normalizationの設定 =====  
# use_batchnorm = True  
use_batchnorm = False  
# =====  
  
network = MultiLayerNet(input_size=784, hidden_size_list=[40, 20], output_size=10, activation='sigmoid', v  
use_batchnorm=use_batchnorm)  
  
iters_num = 1000  
train_size = x_train.shape[0]  
batch_size = 100  
learning_rate = 0.001  
decay_rate = 0.99  
  
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)  
    if i == 0:  
        h = {}  
    for key in ('W1', 'W2', 'W3', 'b1', 'b2', 'b3'):  
        if i == 0:  
            h[key] = np.zeros_like(network.params[key])
```



学習率を $0.01 \Rightarrow 0.001$ に変更すると、学習の収束が遅くなる。

1.3. 過学習

過学習の復習

テスト誤差と訓練誤差とで学習曲線が剥離する。

特定の訓練サンプルに対して特化して学習している。

過学習の原因

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が高い（層数、ノード数、パラメータの値、）

→正則化手法を利用して自由度を制約し、過学習を抑制する。

リッジ回帰：ハイパープラメータを大きな値に設定すると、

すべての重みの限りなくゼロに近づく

1.3.1 L1 正則化、L2 正則化

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。

学習させていくと、重みにばらつきが発生。

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。

→誤差に対して、正則化項を加算することで、重みを抑制

過学習が起こりそうな重みの大きさ以下で、重みをコントロールし、

かつ重みの大きさがばらつくようにする

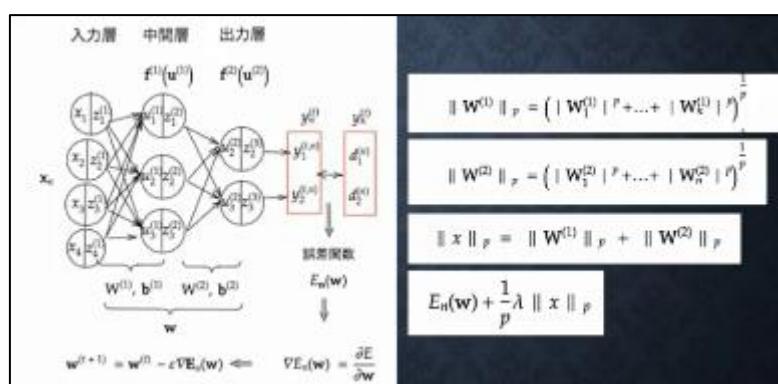
$$E_n(\mathbf{w}) + \frac{1}{p} \lambda \| \mathbf{x} \|_p$$

:誤差関数に、 p ノルムを加える

$$\| \mathbf{x} \|_p = \left(|x_1|^p + \dots + |x_n|^p \right)^{\frac{1}{p}}$$

: p ノルムの計算

$p = 1$ の場合、L1正則化と呼ぶ。
 $p = 2$ の場合、L2正則化と呼ぶ。



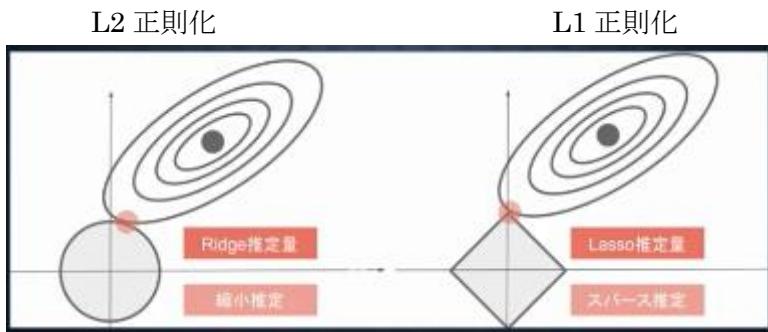
$$\|x\|_p = \left(|x_1|^p + \dots + |x_n|^p \right)^{\frac{1}{p}}$$

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

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

```
weight_decay += weight_decay_lambda
* np.sum(np.abs(network.params['W' + str(idx)]))

loss = network.loss(x_batch, d_batch) + weight_decay
```



L2 パラメータ正則化

```
def ridge(param, grad, rate):
    grad += rate * param
```

rate は L2 正則化の係数、grad は誤差の勾配

L2 ノルムは 2 乗だが、係数 2 は正則化の係数に吸収されても変わらないため、param となる

L1 パラメータ正則化

```
def lasso(param, grad, rate):
    x = np.sign(param)
    grad += rate * x
```

rate は L1 正則化の係数、grad は誤差の勾配

L1 ノルムは $|param|$ なので、その勾配が誤差の勾配に加えられる。

Sign は符号関数：0 より大きい値は 1、0 は 0、マイナスは -1 が返る。

データ集合の拡張

```
def random_crop(image, crop_size):
    h,w,_=image.shape
    crop_h, cron_v = crop_size

    top = np.random.randint(0, h-cron_h)
    left = np.random.randint(0, w-cron_w)
    bottom = top + cron_h
    right = left + cron_w
    image = image[top:bottom, left:right, :]
    return image
```

image の形式は縦幅、横幅、チャンネル

1.3.2 ドロップアウト

ノード数が多い場合で過学習が多い。

ドロップアウトは、ランダムにノードを削除して学習させる

データ量を変化させずに、異なるモデルを学習させていくことになる

DL フレームワークにより少し違いが出てくる。

ハンズオン（実装演習）

jupyter 2_5_overfitting Last Checkpoint: 2018/11/24 (autosaved)  Logo

File Edit View Insert Cell Kernel Widgets Help Trusted Python 3

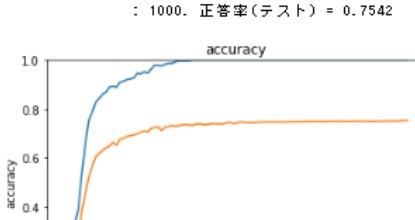
Code

```
acor_train = network.accuracy(x_train, d_train)
acor_test = network.accuracy(x_test, d_test)
accuracies_train.append(acor_train)
accuracies_test.append(acor_test)

print('Generation: ' + str(i+1) + '. 正答率(トレーニング) = ' + str(acor_train))
print(' : ' + str(i+1) + '. 正答率(テスト) = ' + str(acor_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")
plt.ylim(0, 1.0)
# グラフの表示
plt.show()
```

Generation: 990. 正答率(トレーニング) = 1.0
: 990. 正答率(テスト) = 0.7525
Generation: 1000. 正答率(トレーニング) = 1.0
: 1000. 正答率(テスト) = 0.7542



過学習が起きている。

jupyter 2_5_overfitting Last Checkpoint: 2018/11/24 (autosaved)

File Edit View Insert Cell Kernel Widgets Help Trusted Python 3

weight decay

L2

```
In [2]: from common import optimizer

(x_train, d_train), (x_test, d_test) = load_mnist(normalize=True)

print("データ読み込み完了")

# 通常学習を再現するために、学習データを削減
x_train = x_train[:300]
d_train = d_train[:300]

network = MultiLayerNet(input_size=784, hidden_size_list=[100, 100, 100, 100, 100, 100], output_size=10)

iters_num = 1000
train_size = x_train.shape[0]
batch_size = 100
learning_rate=0.01

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

plot_interval=10
hidden_layer_num = network.hidden_layer_num

# 正則化強度設定 =====
weight_decay_lambda = 0.1
# =====
```

jupyter 2_5_overfitting Last Checkpoint: 2018/11/24 (autosaved)

File Edit View Insert Cell Kernel Widgets Help Trusted Python 3

```
# 正則化強度設定 =====
weight_decay_lambda = 0.1
# =====

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]

    grad = network.gradient(x_batch, d_batch)
    weight_decay = 0

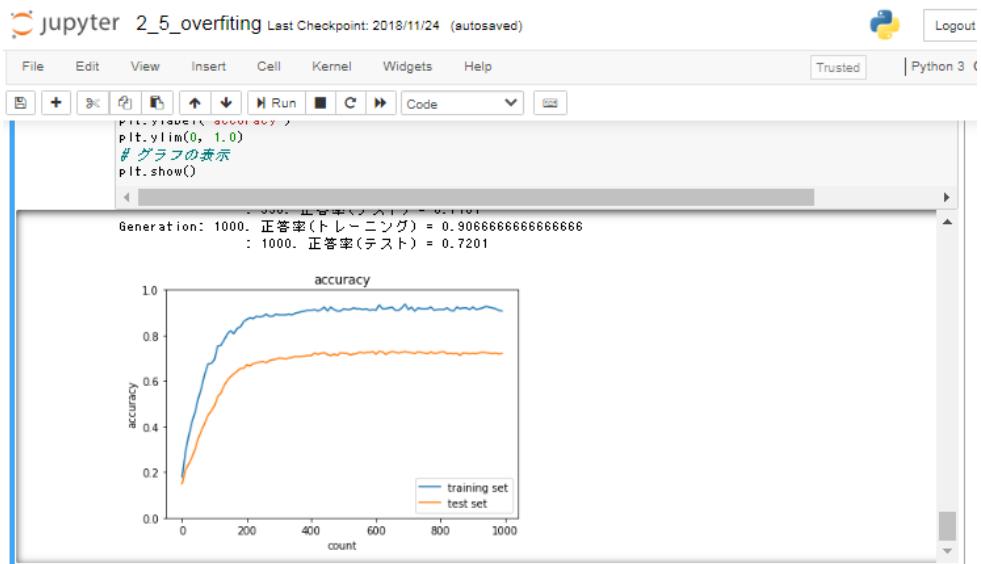
    for idx in range(1, hidden_layer_num+1):
        grad['W' + str(idx)] = network.layers['Affine' + str(idx)].dW + weight_decay_lambda * network.params['W' + str(idx)]
        grad['b' + str(idx)] = network.layers['Affine' + str(idx)].db
        network.params['W' + str(idx)] -= learning_rate * grad['W' + str(idx)]
        network.params['b' + str(idx)] -= learning_rate * grad['b' + str(idx)]
        weight_decay += 0.5 * weight_decay_lambda * np.sqrt(np.sum(network.params['W' + str(idx)] ** 2))

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

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

        print('Generation: ' + str(i+1) + ', 正答率(トレーニング) = ' + str(accur_train))
        print('                 : ' + str(i+1) + ', 正答率(テスト) = ' + str(accur_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")
    plt.ylim(0, 1.0)
    # グラフの表示
    plt.show()
```



L2 正則化で少し過学習が減った

```

In [8]: (x_train, d_train), (x_test, d_test) = load_mnist(normalize=True)
print("データ読み込み完了")

# 過学習を再現するために、学習データを削減
x_train = x_train[:300]
d_train = d_train[:300]

network = MultiLayerNet(input_size=784, hidden_size_list=[100, 100, 100, 100, 100, 100], output_size=10)

iters_num = 1000
train_size = x_train.shape[0]
batch_size = 100
learning_rate=0.1

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

plot_interval=10
hidden_layer_num = network.hidden_layer_num

# 正則化強度設定 =====
weight_decay_lambda = 0.005
# =====

```

```

jupyter 2_5_overfitting Last Checkpoint: 2018/11/24 (autosaved) Logout
File Edit View Insert Cell Kernel Widgets Help Trusted Python 3



```

正則化強度設定 =====
weight_decay_lambda = 0.005
=====

for i in range(iterations_num):
 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)
 weight_decay = 0

 for idx in range(1, hidden_layer_num+1):
 grad['W' + str(idx)] = network.layers['Affine' + str(idx)].dW * weight_decay_lambda * np.sign(network.params['W' + str(idx)])
 grad['b' + str(idx)] = network.layers['Affine' + str(idx)].db
 network.params['W' + str(idx)] -= learning_rate * grad['W' + str(idx)]
 network.params['b' + str(idx)] -= learning_rate * grad['b' + str(idx)]
 weight_decay += weight_decay_lambda * np.sum(np.abs(network.params['W' + str(idx)]))

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

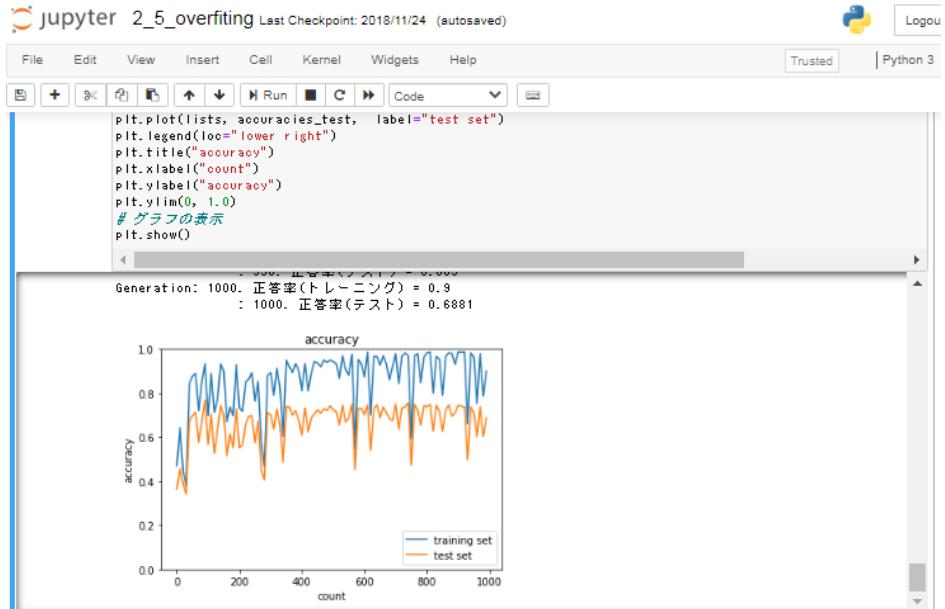
 if (i+1) % plot_interval == 0:
 accr_train = network.accuracy(x_train, d_train)
 accr_test = network.accuracy(x_test, d_test)
 accuracies_train.append(accr_train)
 accuracies_test.append(accr_test)

 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, iterations_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()

```


```



L1 正則化、最初のものよりは過学習が減った

jupyter 2_5_overfitting Last Checkpoint: 2018/11/24 (unsaved changes)

File Edit View Insert Cell Kernel Widgets Help Trusted Python Log

weight decay

L2

```
In [9]: from common import optimizer
(x_train, d_train), (x_test, d_test) = load_mnist(normalize=True)

print("データ読み込み完了")

# 過学習を再現するために、学習データを削減
x_train = x_train[:300]
d_train = d_train[:300]

network = MultiLayerNet(input_size=784, hidden_size_list=[100, 100, 100, 100, 100], output_size=10)

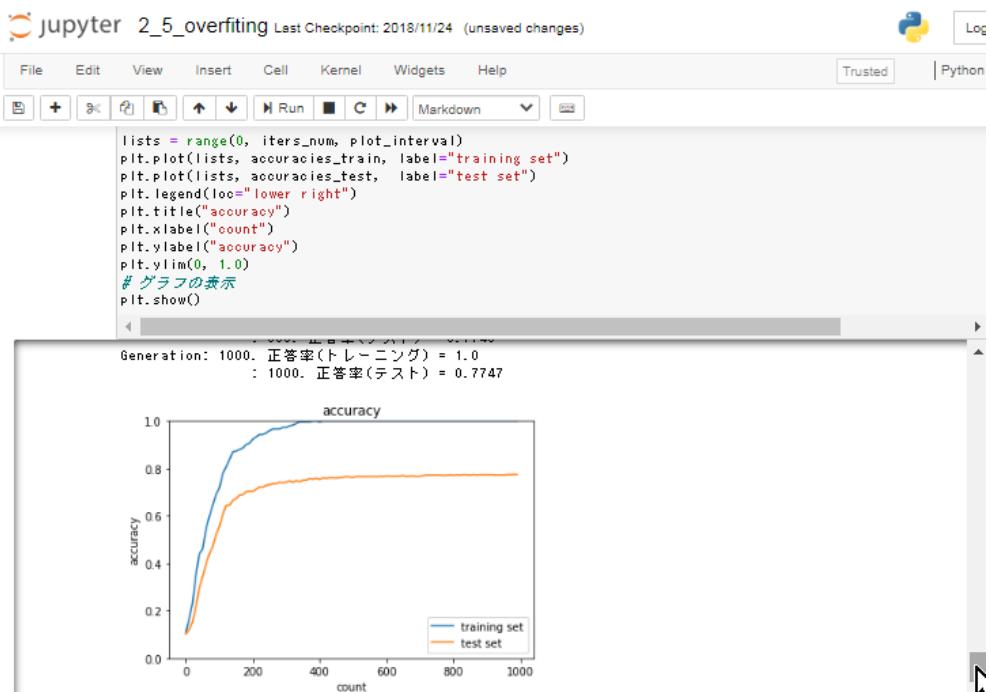
iters_num = 1000
train_size = x_train.shape[0]
batch_size = 100
learning_rate=0.01

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

plot_interval=10
hidden_layer_num = network.hidden_layer_num

# 正則化強度設定 =====
weight_decay_lambda = 0.0001
# =====

for i in range(iters_num):
```



weight_decay_lambda 正則化強度設定を小さくすると、過学習が抑えられない。

jupyter 2_5_overfitting Last Checkpoint: 2018/11/24 (unsaved changes)

File Edit View Insert Cell Kernel Widgets Help Trusted Python

weight decay

L2

```
In [9]: from common import optimizer
(x_train, d_train), (x_test, d_test) = load_mnist(normalize=True)
print("データ読み込み完了")

# 過学習を防ぐために、学習データを削減
x_train = x_train[:300]
d_train = d_train[:300]

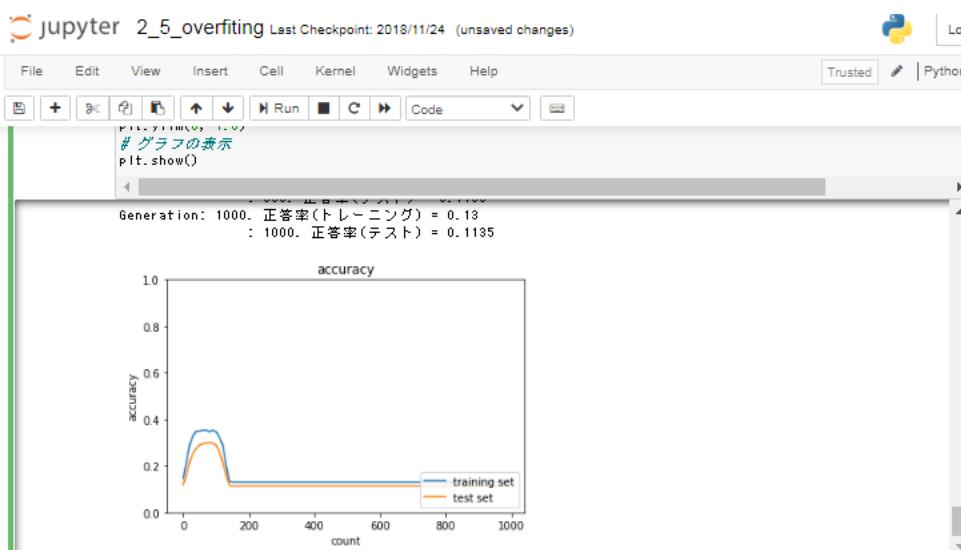
network = MultiLayerNet(input_size=784, hidden_size_list=[100, 100, 100, 100, 100, 100], output_size=10)

iters_num = 1000
train_size = x_train.shape[0]
batch_size = 100
learning_rate=0.01

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

plot_interval=10
hidden_layer_num = network.hidden_layer_num

# 正則化強度設定 =====
weight_decay_lambda = 0.5
# =====
```



weight_decay_lambda 正則化強度設定を大きくすると、学習できなくなる。

jupyter 2_5_overfitting Last Checkpoint: 2018/11/24 (autosaved) Logout

File Edit View Insert Cell Kernel Widgets Help Trusted Python 3 C

Dropout

```
In [4]: 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
```

```
In [5]: from common import optimizer
(x_train, d_train), (x_test, d_test) = load_mnist(normalize=True)

print("データ読み込み完了")

#過学習を再現するために、学習データを削減
x_train = x_train[:300]
d_train = d_train[:300]

# ドロップアウト設定 -----
use_dropout = True
dropout_ratio = 0.15
# -----

network = MultiLayerNet(input_size=784, hidden_size_list=[100, 100, 100, 100, 100, 100], output_size=10,
                        weight_decay_lambda=weight_decay_lambda, use_dropout=use_dropout, dropout_ratio=dropout_ratio)
optimizer = optimizer.SGD(learning_rate=0.01)
# optimizer = optimizer.Momentum(learning_rate=0.01, momentum=0.9)
# optimizer = optimizer.Adam(learning_rate=0.01)
```

jupyter 2_5_overfitting Last Checkpoint: 2018/11/24 (autosaved) Logout

File Edit View Insert Cell Kernel Widgets Help Trusted Python 3

```
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")
plt.ylim(0, 1.0)
# グラフの表示
plt.show()
```

Generation: 1000. 正答率(トレーニング) = 0.85
: 1000. 正答率(テスト) = 0.6631

Dropoutにより、最初のものよりは過学習が減った

jupyter 2_5_overfitting Last Checkpoint: 2018/11/24 (unsaved changes)

File Edit View Insert Cell Kernel Widgets Help Trusted Python 3

```
In [12]: from common import optimizer
(x_train, d_train), (x_test, d_test) = load_mnist(normalize=True)

print("データ読み込み完了")

# 過学習を再現するために、学習データを削減
x_train = x_train[:300]
d_train = d_train[:300]

# ドロップアウト設定 =====
use_dropout = True
dropout_ratio = 0.5
# =====

network = MultiLayerNet(input_size=784, hidden_size_list=[100, 100, 100, 100, 100], output_size=10,
                         weight_decay_lambda=weight_decay_lambda, use_dropout=use_dropout, dropout_ratio=dropout_ratio)
optimizer = optimizer.SGD(learning_rate=0.01)
# optimizer = optimizer.Momentum(learning_rate=0.01, momentum=0.9)
# optimizer = optimizer.AdaGrad(learning_rate=0.01)
# optimizer = optimizer.Adam()

iters_num = 1000
train_size = x_train.shape[0]
batch_size = 100

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]

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

    acor_train = network.accuracy(x_train, d_train)
    accuracies_train.append(acor_train)

    print('Generation: ' + str(i+1) + '. 正答率(トレーニング) = ' + str(acor_train))
    print('           : ' + str(i+1) + '. 正答率(テスト) = ' + str(acor_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")
    plt.ylim(0, 1.0)
    # グラフの表示
    plt.show()
```

Generation: 1000. 正答率(トレーニング) = 0.13
: 1000. 正答率(テスト) = 0.1135

Dropout の dropout_ratio を大きくすると、学習ができなくなる

jupyter 2_5_overfitting Last Checkpoint: 2018/11/24 (unsaved changes)

File Edit View Insert Cell Kernel Widgets Help Trusted Python 3

```
In [1]: from common import optimizer
(x_train, d_train), (x_test, d_test) = load_mnist(normalize=True)

print("データ読み込み完了")

# 過学習を防ぐために、学習データを削減
x_train = x_train[:300]
d_train = d_train[:300]

# ドロップアウト設定 -----
use_dropout = True
dropout_ratio = 0.015
# -----

network = MultiLayerNet(input_size=784, hidden_size_list=[100, 100, 100, 100, 100, 100], output_size=10,
                         weight_decay_lambda=weight_decay_lambda, use_dropout=use_dropout, dropout_ratio=dropout_ratio)
optimizer = optimizer.SGD(learning_rate=0.01)
# optimizer = optimizer.Momentum(learning_rate=0.01, momentum=0.9)
# optimizer = optimizer.AdaGrad(learning_rate=0.01)
# optimizer = optimizer.Adam()

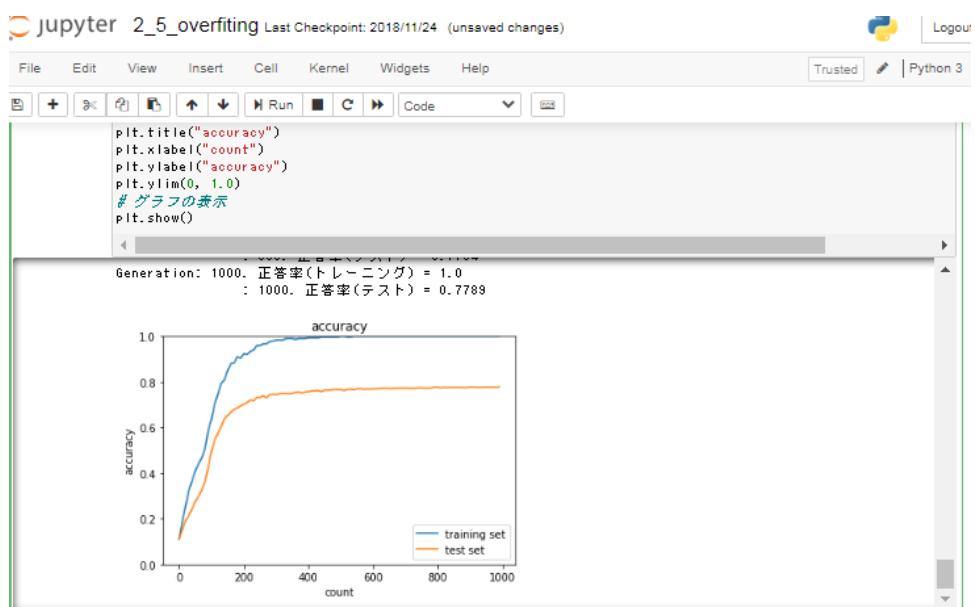
iters_num = 1000
train_size = x_train.shape[0]
batch_size = 100

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

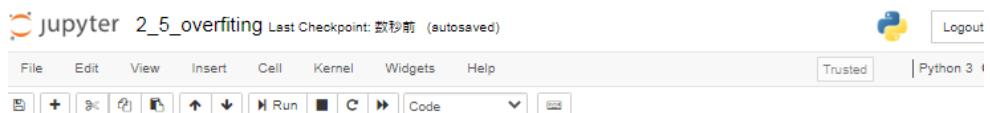
plot_interval=10

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]

    grad = network.gradient(x_batch, d_batch)
    optimizer.update(network.params, grad)
```



Dropout の dropout_ratio を小さくすると、過学習が抑えられなくなる。



```
In [6]: from common import optimizer
(x_train, d_train), (x_test, d_test) = load_mnist(normalize=True)

print("データ読み込み完了")

# 過学習を再現するために、学習データを削減
x_train = x_train[:300]
d_train = d_train[:300]

# ドロップアウト設定 =====
use_dropout = True
dropout_ratio = 0.08
# =====

network = MultiLayerNet(input_size=784, hidden_size_list=[100, 100, 100, 100, 100], output_size=10,
                         use_dropout=use_dropout, dropout_ratio=dropout_ratio)

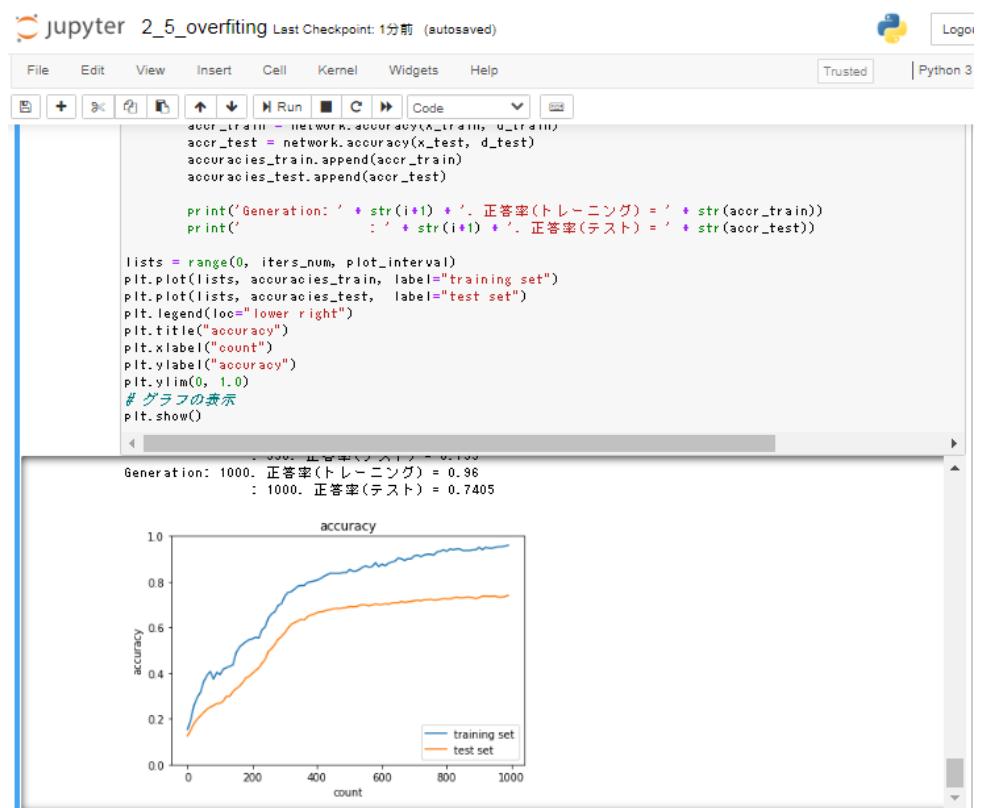
iters_num = 1000
train_size = x_train.shape[0]
batch_size = 100
learning_rate=0.01

train_loss_list = []
accuracies_train = []
accuracies_test = []
hidden_layer_num = network.hidden_layer_num

plot_interval=10

# 正則化強度設定 =====
weight_decay_lambda=0.004
# =====

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]
```



Dropout+L1により、最初のものよりは過学習が減った

1.4. 署み込みニューラルネットワークの概念

CNN の構造の例

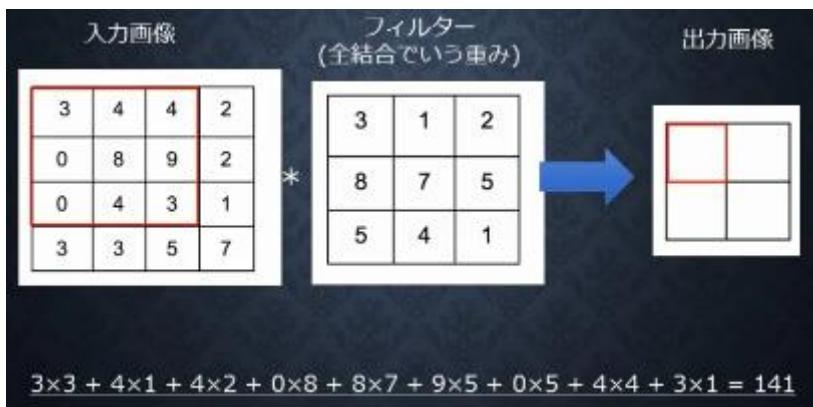
入力層（入力画像）→署み込み層→署み込み層→プーリング層→
→署み込み層→署み込み層→プーリング層→全結合層→出力層（出力画像）

1.4.1 署み込み層

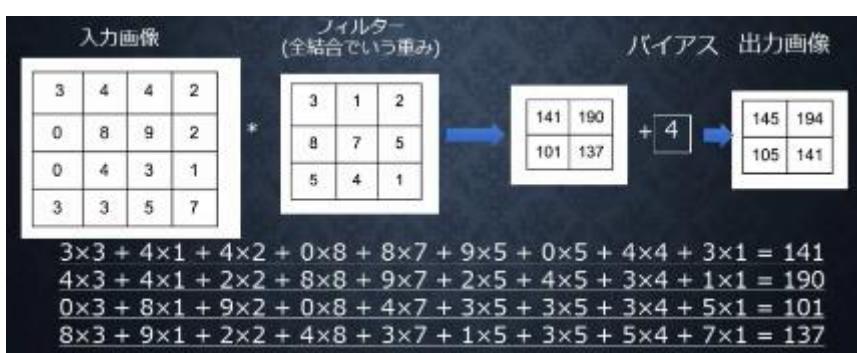


画像の場合、縦、横、チャンネルの3次元のデータをそのまま学習し、次の層に伝える
3次元の空間情報も学習できるような層が署み込み層である。

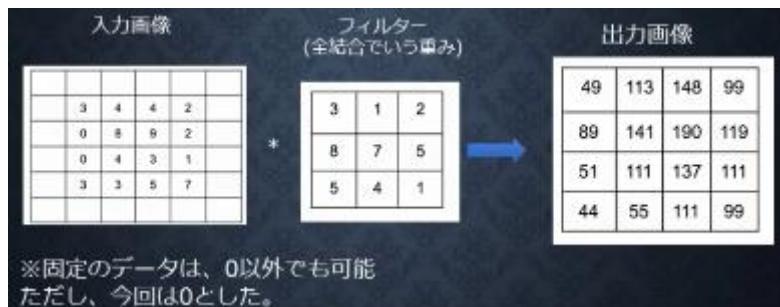
1.4.1.1 バイアス



バイアスで値が加えられる。

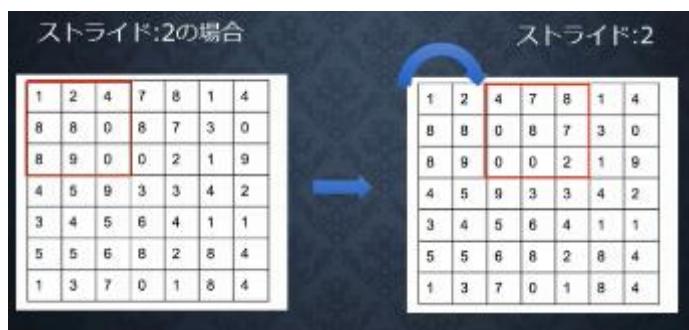


1.4.1.2 パディング



ゼロ以外でも可能だが、入力画像以外の特徴になってしまふので、ゼロが多い。

1.4.1.3 ストライド



ストライドの分だけ飛んでフィルターがかかる。右方向だけでなく下方向にも同じ。

1.4.1.4 チャンネル



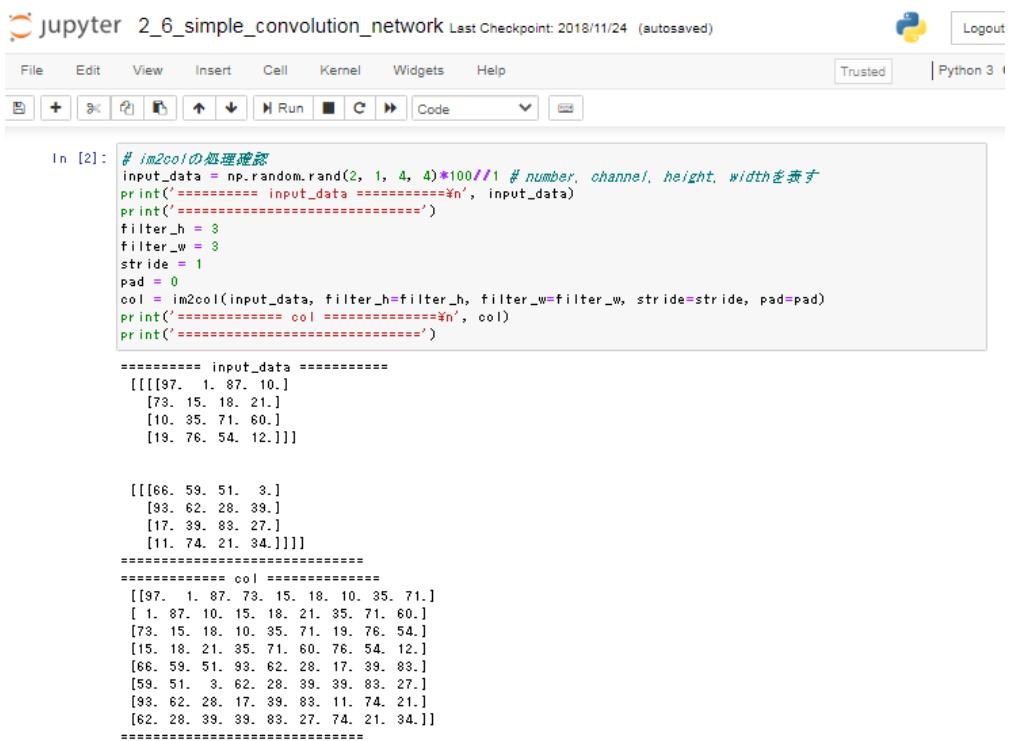
チャンネル数分のフィルターがかかる

全結合で画像を学習した場合

全結合のデメリット

- ・画像の場合、縦横チャンネルの3次元だが、1次元で処理される。
- ・RGBの各チャンネル間の関連性が学習に反映されない。

ハンズオン（実装演習）



The screenshot shows a Jupyter Notebook interface with the title "jupyter 2_6_simple_convolution_network Last Checkpoint: 2018/11/24 (autosaved)". The toolbar includes File, Edit, View, Insert, Cell, Kernel, Widgets, Help, Trusted, Python 3, and Logout. The code cell (In [2]) contains Python code for testing the im2col function:

```
# im2colの処理確認
input_data = np.random.rand(2, 1, 4, 4)*100//1 # number, channel, height, widthを表す
print('===== input_data =====\n', input_data)
print('=====')
filter_h = 3
filter_w = 3
stride = 1
pad = 0
col = im2col(input_data, filter_h=filter_h, filter_w=filter_w, stride=stride, pad=pad)
print('===== col =====\n', col)
print('=====')
```

The output shows the input data as a 2x1x4x4 array and the resulting column vector col:

```
===== input_data =====
[[[97.  1.  87. 10.]
 [73. 15. 18. 21.]
 [10. 35. 71. 60.]
 [19. 76. 54. 12.]]]

===== col =====
[[[66. 59. 51. 3.]
 [93. 62. 28. 39.]
 [17. 39. 88. 27.]
 [11. 74. 21. 34.]]]

=====
```

im2col 関数の処理の確認、2次元配列になる。

im2col 関数のメリットは、行列計算に落としこむことで、多くのライブラリで活用できる

image to column

```
import sys, os
sys.path.append(os.pardir)
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

# 画像データを2次元配列に変換
...
input_data: 入力値
filter_h: フィルターの高さ
filter_w: フィルターの横幅
stride: ストライド
pad: パディング
...
def im2col(input_data, filter_h, filter_w, stride=1, pad=0):
    # N: number, C: channel, H: height, W: width
    N, C, H, W = input_data.shape
    out_h = (H + 2 * pad - filter_h)//stride + 1
    out_w = (W + 2 * pad - filter_w)//stride + 1

    img = np.pad(input_data, [(0,0), (0,0), (pad, pad), (pad, pad)], 'constant')
    col = np.zeros((N, C, filter_h, filter_w, out_h, out_w))

    for y in range(filter_h):
        y_max = y * stride * out_h
        for x in range(filter_w):
            x_max = x * stride * out_w
            col[:, :, y, x, :, :] = img[:, :, y:y_max:stride, x:x_max:stride]

    #col = col.transpose(0, 4, 5, 1, 2, 3) # (N, C, filter_h, filter_w, out_h, out_w) -> (N, filter_w, out_
    col = col.reshape(N * out_h * out_w, -1)
    return col
```

transpose の処理をコメントアウトして実行する

Jupyter 2_6_simple_convolution_network Last Checkpoint: 2018/11/24 (unsaved changes)

File Edit View Insert Cell Kernel Widgets Help Trusted Python 3

```
In [9]: # im2colの処理確認
input_data = np.random.rand(2, 1, 4, 4)*100//1 # number, channel, height, widthを表す
print(' ===== input_data =====\n', input_data)
print(' =====')
filter_h = 3
filter_w = 3
stride = 1
pad = 0
col = im2col(input_data, filter_h=filter_h, filter_w=filter_w, stride=stride, pad=pad)
print(' ===== col =====\n', col)
print(' =====')

=====
input_data =====
[[[37. 92. 6. 42.]
 [21. 40. 8. 83.]
 [56. 50. 77. 24.]
 [70. 78. 26. 52.]]

 [[48. 57. 80. 58.]
 [60. 69. 63. 62.]
 [36. 97. 46. 83.]
 [46. 33. 28. 3.]]]

=====
col =====
[[37. 92. 21. 40. 92. 6. 40. 8. 6.]
 [42. 8. 83. 21. 40. 56. 50. 40. 8.]
 [50. 77. 8. 83. 77. 24. 56. 50. 70.]
 [78. 50. 77. 78. 26. 77. 24. 26. 52.]
 [48. 57. 60. 69. 57. 80. 69. 63. 80.]
 [58. 63. 62. 60. 69. 36. 97. 69. 68.]
 [97. 46. 63. 62. 46. 63. 36. 97. 46.]
 [33. 97. 46. 33. 28. 46. 83. 28. 3.]]]
```

```

jupyter 2_6_simple_convolution_network Last Checkpoint: 2018/11/24 (unsaved changes)
File Edit View Insert Cell Kernel Widgets Help Trusted Python 3 Logout

File + Run C Code
out_h = (H * 2 * pad - filter_h)//stride + 1
out_w = (W * 2 * pad - filter_w)//stride + 1
col = col.reshape(N, out_h, out_w, C, filter_h, filter_w).transpose(0, 3, 4, 5, 1, 2) # (N, filter_h, filter_w, out_h, out_w, C)
img = np.zeros((N, C, H * 2 * pad + stride - 1, W * 2 * pad + stride - 1))
for y in range(filter_h):
    y_max = y * stride * out_h
    for x in range(filter_w):
        x_max = x * stride * out_w
        img[:, :, y:y_max:stride, x:x_max:stride] += col[:, :, y, x, :, :]
return img[:, :, pad:H * pad, pad:W * pad]

```

col2imの処理を確認しよう
・im2colの確認で出力したcolをimageに変換して確認しよう

```

In [11]: # col2imの処理確認
imout = col2im(col, (2,1,4), filter_h=filter_h, filter_w=filter_w, stride=stride, pad=pad)
print('===== image =====', imout)
print('=====')

===== image =====
[[[ 37. 134. 29. 83.]
 [ 90. 268. 104. 133.]
 [123. 213. 96. 85.]
 [ 56. 74. 96. 52.]]

 [[ 48. 115. 123. 62.]
 [166. 196. 309. 82.]
 [131. 239. 260. 109.]
 [ 36. 180. 74.  3.]]]
=====
```

com2im で元の画像に復元できる

1.4.2 プーリング層

Max プーリング、Avarage プーリング



例 1) サイズ 6x6 の入力画像を、サイズ 2x2 のフィルタで畳み込んだ時、

ストライドとパディングが 1 とすると

出力画像のサイズは 7x7

公式がある

$$OH = (H + 2p - FH) / S + 1 = (6 + 2 \times 1 - 2) / 1 + 1 = 7$$

$$OW = (W + 2p - FW) / S + 1 = (6 + 2 \times 1 - 2) / 1 + 1 = 7$$

FH: filter_height、FW: filter_width、p:パディング、S:ストライド

例 2) サイズ 5x5 の入力画像を、サイズ 3x3 のフィルタで畳み込んだ時、

ストライドは 2 とパディングが 1 とすると

出力画像のサイズは 3x3

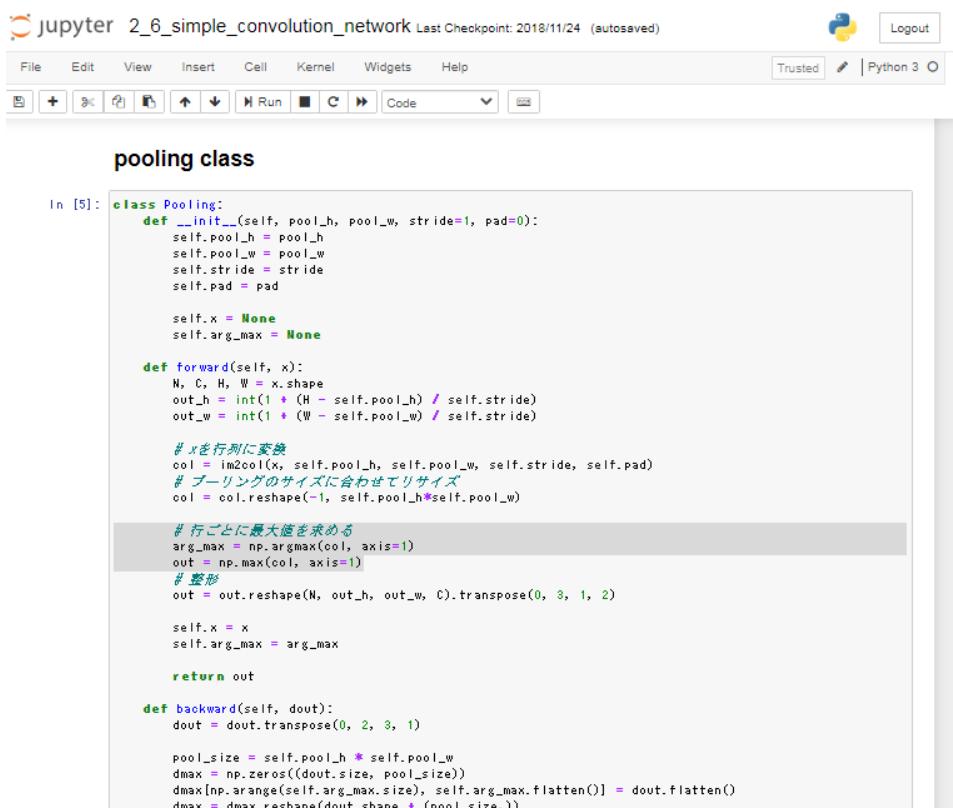
公式がある

$$OH = (H + 2p - FH) / S + 1 = (5 + 2 \times 1 - 3) / 2 + 1 = 3$$

$$OW = (W + 2p - FW) / S + 1 = (5 + 2 \times 1 - 3) / 2 + 1 = 3$$

FH: filter_height、FW: filter_width、p:パディング、S:ストライド

ハンズオン（実装演習）



The screenshot shows a Jupyter Notebook interface with the title "jupyter 2_6_simple_convolution_network". The notebook has a single cell labeled "In [5]:" containing Python code for a "Pooling class". The code defines a class "Pooling" with methods for initialization, forward pass, and backward pass. The forward pass involves calculating output dimensions, creating a column vector from the input, finding the maximum value in each column, and reshaping the result. The backward pass involves calculating gradients based on the pool size and the input gradient. A portion of the code for finding the maximum value in each column is highlighted in gray.

```
In [5]: class Pooling:
    def __init__(self, pool_h, pool_w, stride=1, pad=0):
        self.pool_h = pool_h
        self.pool_w = pool_w
        self.stride = stride
        self.pad = pad

        self.x = None
        self.argmax = None

    def forward(self, x):
        N, C, H, W = x.shape
        out_h = int(1 + (H - self.pool_h) // self.stride)
        out_w = int(1 + (W - self.pool_w) // self.stride)

        # xを行列に変換
        col = im2col(x, self.pool_h, self.pool_w, self.stride, self.pad)
        # ブーリングのサイズに合わせてリサイズ
        col = col.reshape(-1, self.pool_h * self.pool_w)

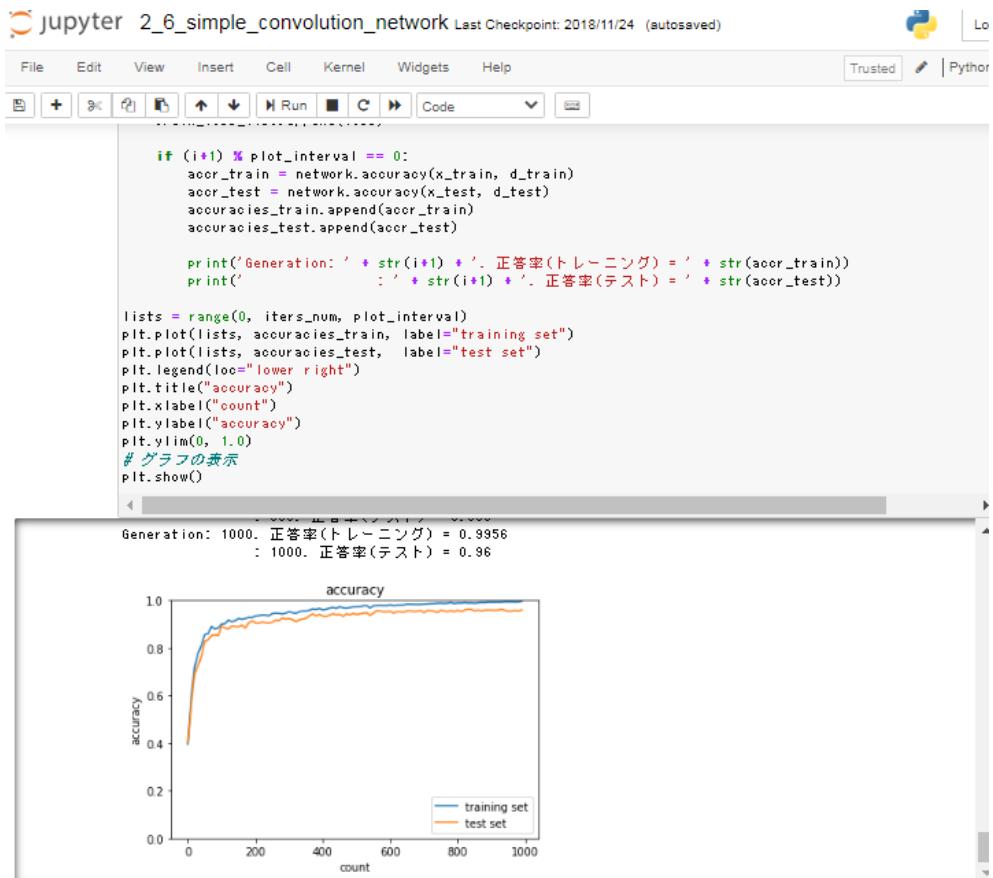
        # 行ごとに最大値を求める
        arg_max = np.argmax(col, axis=1)
        out = np.max(col, axis=1)
        # 整形
        out = out.reshape(N, out_h, out_w, C).transpose(0, 3, 1, 2)

        self.x = x
        self.argmax = arg_max

        return out

    def backward(self, dout):
        dout = dout.transpose(0, 2, 3, 1)

        pool_size = self.pool_h * self.pool_w
        dmax = np.zeros((dout.size, pool_size))
        dmax[np.arange(dout.argmax.size), self.argmax.flatten()] = dout.flatten()
        dmax = dmax.reshape(dout.shape + (pool_size,))
```



Goodfellow 他著「Deep Learning」

<http://www.deeplearningbook.org/>

大項目	中項目	小項目	細項目	対応チャプター (Goodfellow著 "Deep Learning")
応用数学	線形代数	特異値分解		2.8
	確率・統計	一般的な確率分布	ペレヌーイの分布	3.9
			マルチヌーイの分布	3.9
			ガウス分布	3.9
	情報理論	ベイズ則		3.11
		情報理論		3.13

- ・講義の内容を理解していればシラバスの範囲（筆記試験）は問題ありません。
- ・誤差逆伝播法で用いる偏微分などはシラバスには含まれませんが追加。

機械学習	機械学習の基礎	学習アルゴリズム	タスクT 性能指標P 経験E	5.1 5.1 5.1 5.2 5.3 5.3 5.3 5.5 5.5 5.7
		能力・過剰適合・過少適合 ハイパーパラメータ 検証集合	学習データ、検証データ、テストデータ ホールドアウト法 k分割交差検証法 条件付き対数尤度と平均二乗誤差	5.2 5.3 5.3 5.5
		最尤推定	最尤法の特性 教師あり学習アルゴリズム 教師なし学習アルゴリズム	5.5 5.7 5.7 5.7
		教師あり学習アルゴリズム	主成分分析 k平均クラスタリング	5.8 5.8
		確率的勾配降下法 深層学習の発展を促す課題	次元のない 局所一様と平滑化	5.9 5.11 5.11
		性能指標 データの追加収集の判断 ハイパーパラメータの選択	手動でのハイパーパラメータ調整 グリッドサーチ ランダムサーチ モデルに基づくハイパーパラメータの最適化	11.1 11.3 11.4.1 11.4.3 11.4.4 11.4.5
		実用的な方法論		
・特別な技術ではなく一般的な話のまとめ。				

順伝播型ネットワーク	線形問題と非線形問題	コスト関数	最尤推定による条件付き分布の学習 条件付き統計量の学習	6序文 6.2.1 6.2.1
		出力ユニット	ガウス出力分布のための線形ユニット ペリスロイ出力分布のためのシグモイドユニット マルチヌイー出力分布のためのソフトマックスユニット	6.2.2 6.2.2 6.2.2
		隠れユニット	ReLUとその一般化 ロジスティックシグモイドとハイパボリックタンジェント その他の隠れユニット (RBF, ソフトプラス, Hard)	6.3 6.3 6.3
		アーキテクチャの設計	万能近似定理と深さ	6.4
		誤差逆伝搬法およびその他の微分アルゴリズム	計算グラフ 微積分の連鎖率	6.5.1 6.5.2
			誤差逆伝搬のための連鎖率の再起的な適用	6.5.3
			全結合 MLP での誤差逆伝搬法	6.5.4
			シンボル間の微分	6.5.5
			一般的な誤差逆伝播法	6.5.6
・Day 1 の講義で扱った最も基本的な内容です。基本だけに重要。予習資料を手打ち、手計算で繰り返しましょう。				

深層モデルのための正則化	パラメータノルムペナルティー L2 パラメータ正則化	7.1.1	講義
	L1 正則化	7.1.2	講義
	条件付き最適化としてのノルムペナルティ	7.2	演習問題
	正則化と制約不足問題	7.3	演習問題
	データ集合の拡張	7.4	講義
	ノイズに対する頑健性	7.5	演習問題
	半教師あり学習	7.6	演習問題
	マルチタスク学習	7.7	演習問題
	早期終了	7.8	演習問題
	パラメータ拘束とパラメータ共有	7.9	演習問題
	スパース表現	7.10	講義
	バギングやその他のアンサンブル手法	7.11	演習問題
	ドロップアウト	7.12	講義

- ・講義では正則化の流れを最もスタンダードな手法を用いて演習しました。
- ・講義で扱っていない内容は演習問題のキーワードやシラバスから自身でのまとめ学習をしてください。（9 Action 検索力向上で後述）

深層モデルのための最適化	学習と純粹な最適化の差異	経験損失最小化	8.1.1	講義
		代理損失関数と早期終了	8.1.2	講義
		バッチアルゴリズムとミニバッチアルゴリズム	8.1.3	講義
	ニューラルネットワーク最適化 課題	悪条件	8.2.1	講義(day1)
		局所値	8.2.2	講義(day1)
		プラトー、鞍点、その他平坦な領域	8.2.3	講義
	基礎的なアルゴリズム	崖と勾配爆発	8.2.4	演習問題
		長期依存性	8.2.5	講義
		不正確な勾配	8.2.6	講義
	基礎的なアルゴリズム	確率的勾配降下法	8.3.1	講義
		モメンタム	8.3.2	講義
		ネステロフのモメンタム	8.3.3	演習問題
	パラメータの初期化戦略		8.4	講義
	適応的な学習率を持つアルゴリズム	AdaGrad	8.5.1	講義
		RMSprop	8.5.2	講義
		Adam	8.5.3	講義
	二次手法の近似	ニュートン法	8.6.1	演習問題
		共役勾配	8.6.2	演習問題
		BFGS	8.6.3	演習問題
	最適化戦略とメタアルゴリズム	バッチ正規化	8.7.1	講義
	教師あり事前学習		8.7.4	ソース演習問題

- ・ディープラーニングの基礎的なアルゴリズムです。

畳み込みネットワーク 畳み込み処理			9.1	講義
ブーリング	最重要		9.3	講義
構造出力			9.6	講義
データの種類			9.7	講義
効率的な畳み込みアルゴリズム			9.8	講義
ランダムあるいは教師なし特徴量			9.9	講義
画像認識の有名なモデル	VGG	例題参照	演習問題	
	AlexNet	例題参照	講義	
	GoogLeNet	例題参照	演習問題	
	Resnet	例題参照	演習問題	
特徴量の転移			15.2	演習問題
画像の局在化、検知、セグメンテーション		例題参照	演習問題	

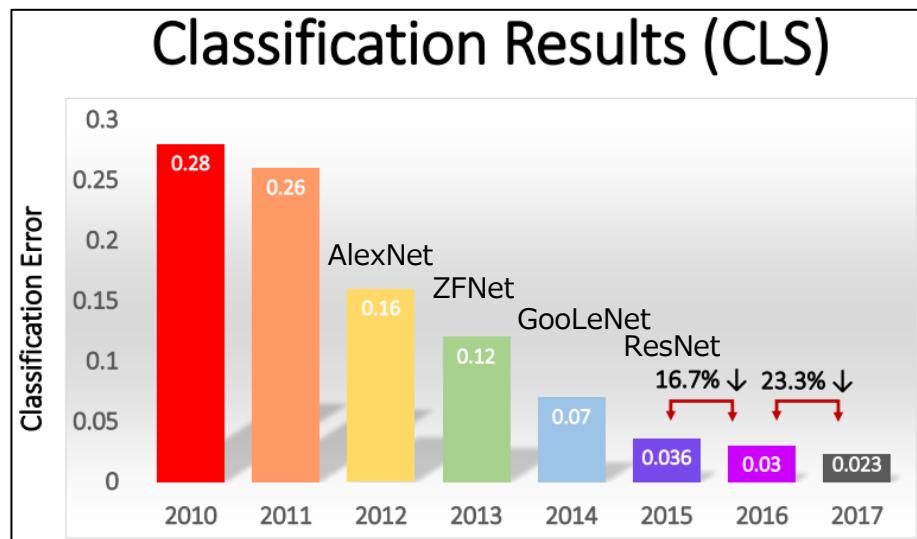
回帰結合型ニューラルネットワークと再帰的ネットワーク	回帰結合型のニューラルネットワーク教師強制と出力回帰のあるネットワーク	10.2	講義
	回帰結合型ネットワークにおける勾配計算 (BPTT)	10.2	講義
	有向グラフィカルモデルとしての回帰結合型のネットワーク	10.2	講義
	RNNを使った文脈で条件付けされた系列モデリング	10.2	講義
双方向 RNN		10.3	演習問題
Encoder-Decoder と Sequence-to-Sequence		10.4	演習問題
深層回帰結合型のネットワーク		10.5	講義
再帰型ニューラルネットワーク		10.6	講義
長期依存性の課題		10.7	講義
エコーステートネットワーク		10.8	講義
複数時間スケールのための Leaky ユニットとその他の手法	時間方向にスキップ接続を追加	10.9	講義
	Leakyユニットと異なる時間スケールのスペクトル接続の削除	10.9	講義
ゲート付きRNN	LSTM	10.10, 10.11	演習問題
	GRU	10.11	演習問題
長期依存性の最適化	勾配のクリッピング	10.12	演習問題
自然言語処理とRNN		12.4.2	講義
メモリネットワーク	Attention	12.4.5.1	講義

- Day3の内容。CNNの応用系で時系列データの扱いを学びます。
- TensorFlow,Keras (4日目) やChainerでも演習して理解を深めましょう。

生成モデル	識別モデルと生成モデル	例題参照	講義
	オートエンコーダ	VAE	20.10.3 講義
	GAN	DCGAN	20.10.4 演習問題
強化学習	方策勾配法	例題参照	講義
	価値反復法	DQN	例題参照 演習問題

- ・大変難易度の高い分野です。講義では紹介にとどめます。
- ・「**09 Action**」で論文などもご紹介しますので実践が近道です。

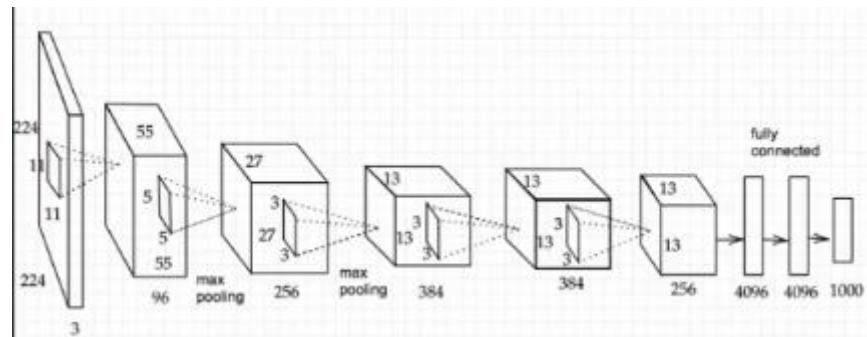
1.5. 最新の CNN



ILSVRC のクラス分類タスクの精度の推移[9]

1.5.1 AlexNet

2012 年のコンペにて 2 位に大差をつけた



5層の畳み込み層およびブーリング層など

3層の全結合層

過学習を防ぐため、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。

1.5.2 ZfNet

The screenshot shows a PDF document titled 'Visualizing and Understanding Convolutional Networks' by Matthew D. Zeiler and Rob Fergus. The document is dated Nov 28, 2013. The page number is 1/11. The abstract discusses the challenges in understanding deep convolutional network models, including the availability of large training sets, powerful GPU implementations, and better regularization strategies like Dropout. It also mentions an ablation study and sensitivity analysis. The document is available at arxiv.org/pdf/1311.2901.pdf.

ZfNet の論文を確認した。[10]

AlexNet の問題点を改善している。

- 最初の畳み込み層のフィルタサイズを 11 から 7 に縮小
- ストライドを 4 から 2 に縮小

1.5.3 GooLeNet

The screenshot shows a PDF document titled "Going deeper with convolutions" by Christian Szegedy et al. The document is viewed on a browser window with a dark theme. The page includes author names, affiliations, and a short abstract. The abstract discusses the Inception architecture's design for increasing depth and width while keeping computational budget constant. The page number is 1/12. On the left, a vertical sidebar displays the document ID "iv:1409.4842v1 [cs.CV] 17 Sep 2014". On the right, there are zoom controls (+, -, ×).

iv:1409.4842v1 [cs.CV] 17 Sep 2014

Going deeper with convolutions

Christian Szegedy
Google Inc.

Wei Liu
University of North Carolina, Chapel Hill

Yangqing Jia
Google Inc.

Pierre Sermanet
Google Inc.

Scott Reed
University of Michigan

Dragomir Anguelov
Google Inc.

Dumitru Erhan
Google Inc.

Vincent Vanhoucke
Google Inc.

Andrew Rabinovich
Google Inc.

Abstract

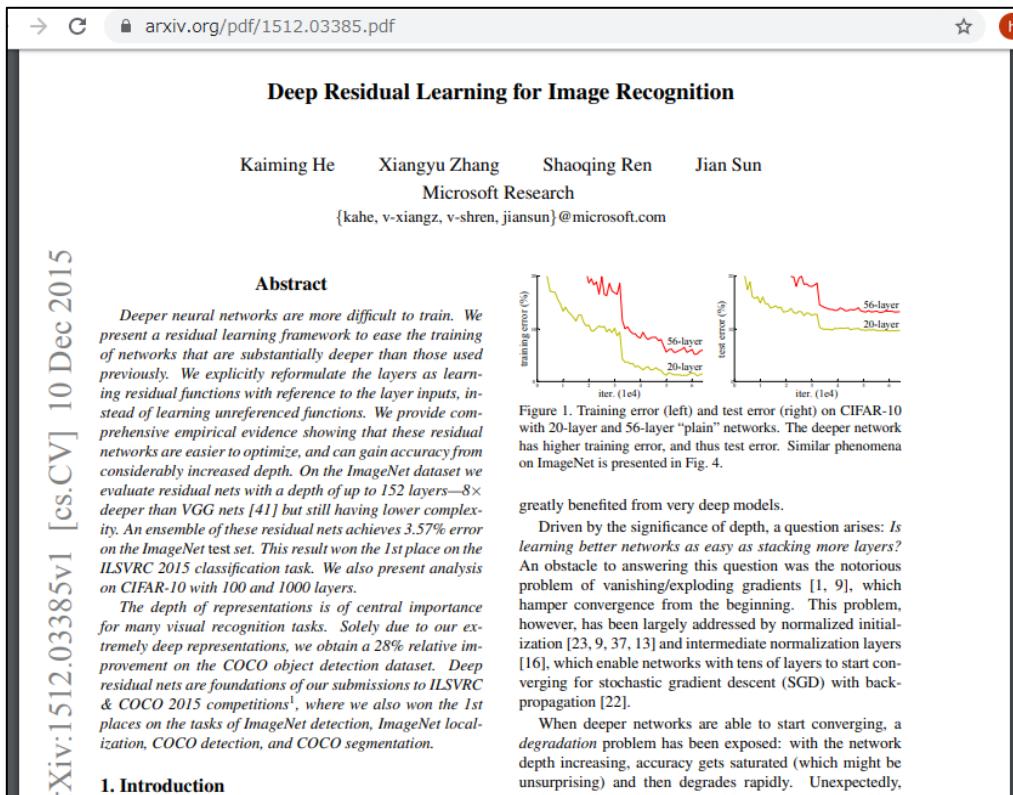
We propose a deep convolutional neural network architecture codenamed Inception, which was responsible for setting the new state of the art for classification and detection in the ImageNet Large-Scale Visual Recognition Challenge 2014 (ILSVRC14). The main hallmark of this architecture is the improved utilization of the computing resources inside the network. This was achieved by a carefully crafted design that allows for increasing the depth and width of the network while keeping the computational budget constant. To optimize quality, the architectural decisions were based on the Hebbian principle and the intuition of multi-scale processing. One particular incarnation used in our submission for ILSVRC14 is called GoogLeNet, a 22 layers deep network, the quality of which is assessed in the context of classification and detection.

1 Introduction

GooLeNet の論文を確認した。[11]

複数の畳み込み層や pooling 層から構成される Inception モジュールと呼ばれる小さなネットワークを定義し、これを通常の畳み込み層のように重ねていくことで 1 つの大きな CNN を作り上げている

1.5.4 ResNet



ResNet の論文を確認した。[12]

通常のネットワークのように、処理ブロックによる変換を単純に次の層に渡していくのではなく、その処理ブロックへの入力をショートカットする residual モジュールを作り、このモジュールを複数積み重ねることにより深いモデルを作った。

2. 深層学習（後編 1）

2.1. 再帰型ニューラルネットワークの概念

2.1.1 RNN 全体像

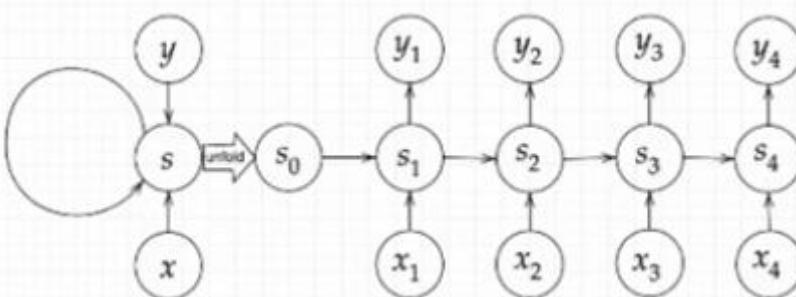
2.1.1.1 RNN とは

時系列データに対応可能なニューラルネットワーク

2.1.1.2 時系列データ

時間的順序を追って一定間隔ごとに観察され、
かつ相互に統計的依存関係が認められるようなデータ
例) 音声データ、テキストデータ…

2.1.1.3 RNN について



$$\begin{aligned} u^t &= W_{(in)}x^t + W z^{t-1} + b \\ z^t &= f(W_{(in)}x^t + W z^{t-1} + b) \\ v^t &= W_{(out)}z^t + c \\ y^t &= g(W_{(out)}z^t + c) \end{aligned}$$

```
u[:,t+1] = np.dot(X, W_in) + np.dot(z[:,t].reshape(1, -1), W)
```

```
z[:,t+1] = functions.sigmoid(u[:,t+1])
```

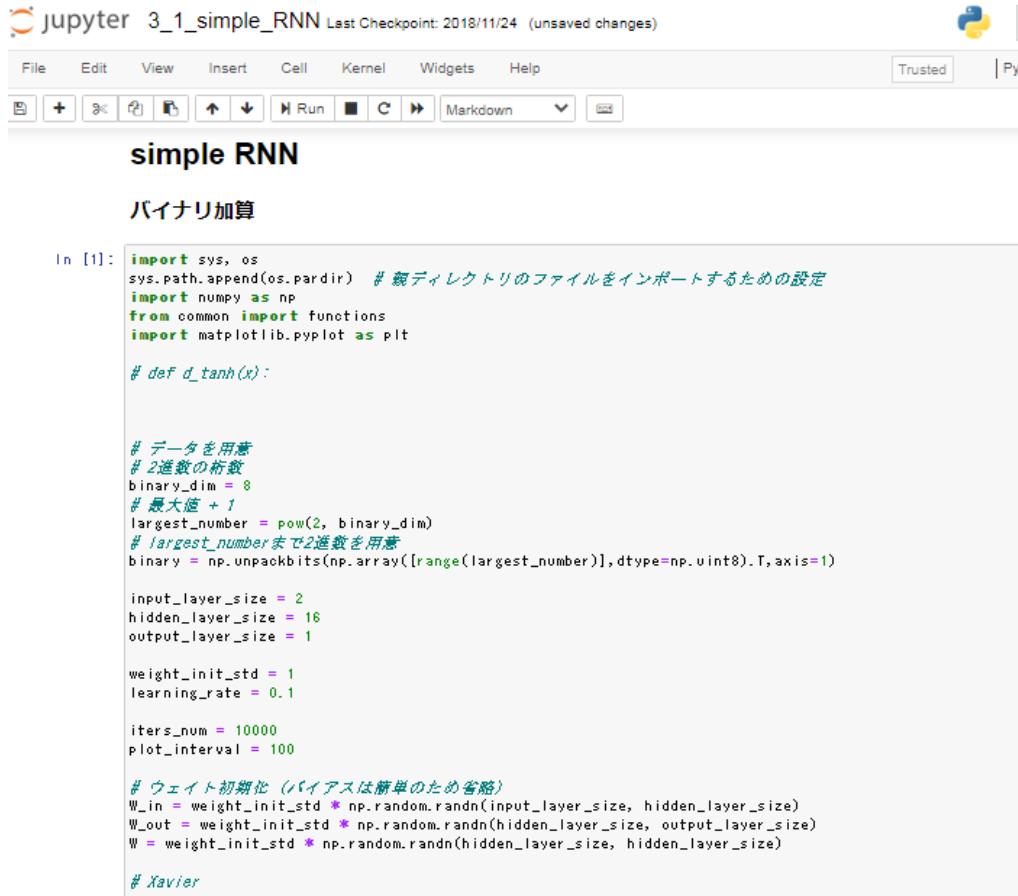
重み 1 ($W_{(in)}$) : 入力から現在の中間層を定義する際にかけられる重み

重み 2 ($W_{(out)}$) : 中間層から出力を定義する際にかけられる重み

重み 3 (W) : 中間層から次の中間層に渡される際にかけられる重み

RNN の特徴：初期の状態と過去の時間 $t-1$ の状態を保持し、
そこから次の時間での t を再帰的に求める再帰構造が必要になる

ハンズオン（実装演習）



The screenshot shows a Jupyter Notebook interface with the title "simple RNN". The code in cell [1] is as follows:

```
import sys, os
sys.path.append(os.pardir) # 窓ディレクトリのファイルをインポートするための設定
import numpy as np
from common import functions
import matplotlib.pyplot as plt

# def d_tanh(x):

# データを用意
# 2進数の桁数
binary_dim = 8
# 最大値 + 1
largest_number = pow(2, binary_dim)
# largest_numberまで2進数を用意
binary = np.unpackbits(np.array([range(largest_number)]), dtype=np.uint8).T, axis=1

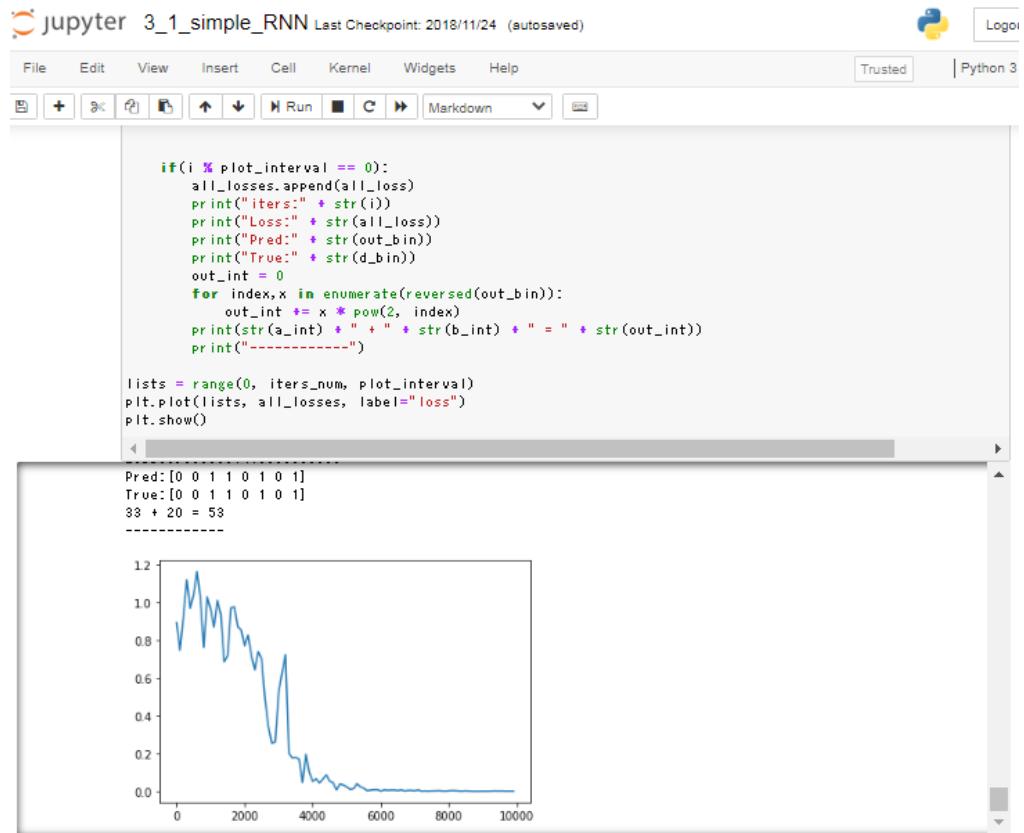
input_layer_size = 2
hidden_layer_size = 16
output_layer_size = 1

weight_init_std = 1
learning_rate = 0.1

iters_num = 10000
plot_interval = 100

# ウェイト初期化（バイアスは簡単のため省略）
W_in = weight_init_std * np.random.randn(input_layer_size, hidden_layer_size)
W_out = weight_init_std * np.random.randn(hidden_layer_size, output_layer_size)
W = weight_init_std * np.random.randn(hidden_layer_size, hidden_layer_size)

# Xavier
```



weight_init_std を 1 から 0.001 に変更

The screenshot shows a Jupyter Notebook interface with a Python 3 kernel. The code cell has been modified to set weight_init_std to 0.001. The output shows the same initial data as before, followed by the new code execution. The plot shows a similar rapid decrease in loss, reaching near-zero values by iteration 4,000.

```

In [1]: import sys, os
sys.path.append(os.pardir) # 親ディレクトリのファイルをインポートするための設定
import numpy as np
from common import functions
import matplotlib.pyplot as plt

# def d_tanh(x):

# データを用意
# 2進数の桁数
binary_dim = 8
# 最大値 + 1
largest_number = pow(2, binary_dim)
# largest_numberまで2進数を用意
binary = np.unpackbits(np.array([range(largest_number)]), dtype=np.uint8).T, axis=1

input_layer_size = 2
hidden_layer_size = 16
output_layer_size = 1

weight_init_std = 0.001
learning_rate = 0.1

iters_num = 10000
plot_interval = 100

# ウェイト初期化 (バイアスは簡単のため省略)
W_in = weight_init_std * np.random.randn(input_layer_size, hidden_layer_size)
W_out = weight_init_std * np.random.randn(hidden_layer_size, output_layer_size)
W = weight_init_std * np.random.randn(hidden_layer_size, hidden_layer_size)

# Xavier

```

jupyter 3_1_simple_RNN Last Checkpoint: 2018/11/24 (unsaved changes)

File Edit View Insert Cell Kernel Widgets Help Trusted Python 3

```
        out_int += x * pow(2, index)
        print(str(a_int) + " + " * str(b_int) + " = " * str(out_int))
        print("-----")
```

```
lists = range(0, iters_num, plot_interval)
plt.plot(lists, all_losses, label="loss")
plt.show()
```

```
Pred:[1 1 1 1 1 1 1]
True:[1 1 0 0 0 1 0]
101 + 85 = 255
-----

```

学習されていない。

逆に weight_init_std を大きくする、1 から 5 に変更

jupyter 3_1_simple_RNN Last Checkpoint: 2018/11/24 (unsaved changes)

File Edit View Insert Cell Kernel Widgets Help Trusted Python 3

simple RNN

バイナリ加算

```
In [1]: import sys, os
sys.path.append(os.pardir) # 父ディレクトリのファイルをインポートするための設定
import numpy as np
from common import functions
import matplotlib.pyplot as plt

# def d_tanh(x):
```

```
# データを用意
# 2進数の桁数
binary_dim = 8
# 最大値 + 1
largest_number = pow(2, binary_dim)
# largest_numberまで2進数を用意
binary = np.unpackbits(np.array([range(largest_number)]).T, axis=1)

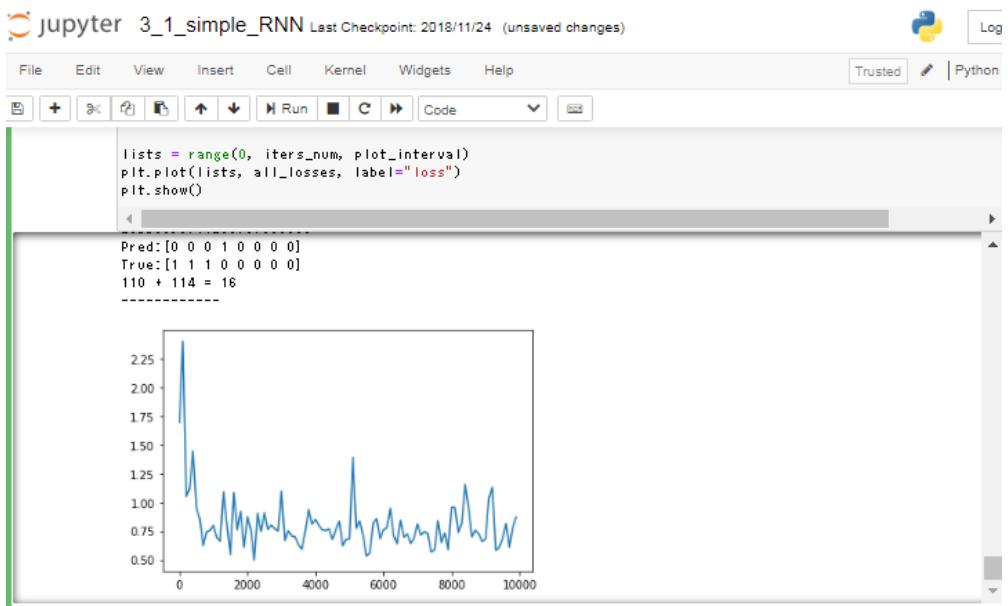
input_layer_size = 2
hidden_layer_size = 16
output_layer_size = 1

weight_init_std = 5
learning_rate = 0.1

iters_num = 10000
plot_interval = 100
```

```
# ウエイト初期化(バイアスは簡単のため省略)
W_in = weight_init_std * np.random.randn(input_layer_size, hidden_layer_size)
W_out = weight_init_std * np.random.randn(hidden_layer_size, output_layer_size)
W = weight_init_std * np.random.randn(hidden_layer_size, hidden_layer_size)
```

```
# Xavier
```



学習が速くなった

初期値を Xavier で与える

```

In [8]: import sys, os
sys.path.append(os.pardir) # 父ディレクトリのファイルをインポートするための設定
import numpy as np
from common import functions
import matplotlib.pyplot as plt

# def d_tanh(x):

# データを用意
# 2進数の桁数
binary_dim = 8
# 最大値 + 1
largest_number = pow(2, binary_dim)
# largest_numberまで2進数を用意
binary = np.unpackbits(np.array([range(largest_number)]), dtype=np.uint8).T, axis=1

input_layer_size = 2
hidden_layer_size = 16
output_layer_size = 1

weight_init_std = 1
learning_rate = 0.1

iter_num = 10000
plot_interval = 100

# ウエイト初期化 (バイアスは簡単のため省略)
# W_in = weight_init_std * np.random.randn(input_layer_size, hidden_layer_size)
# W_out = weight_init_std * np.random.randn(hidden_layer_size, output_layer_size)
# W = weight_init_std * np.random.randn(hidden_layer_size, hidden_layer_size)

# Xavier
W_in = np.random.randn(input_layer_size, hidden_layer_size) / (np.sqrt(input_layer_size))
W_out = np.random.randn(hidden_layer_size, output_layer_size) / (np.sqrt(hidden_layer_size))
W = np.random.randn(hidden_layer_size, hidden_layer_size) / (np.sqrt(hidden_layer_size))

```

```

print(str(a_int) + " * " + str(b_int) + " = " + str(out_int))
print("-----")

lists = range(0, iter_num, plot_interval)
plt.plot(lists, all_losses, label="loss")
plt.show()

```

Pred:[0 1 0 0 1 0 0 0]
True:[0 1 0 0 1 0 0 0]
59 + 13 = 72

最初の物とほぼ同じ

初期値を He で与える

```

import sys, os
sys.path.append(os.pardir) # 絶対ディレクトリのファイルをインポートするための設定
import numpy as np
from common import functions
import matplotlib.pyplot as plt

# def d_tanh(x):

# データを用意
# 2進数の桁数
binary_dim = 8
# 最大値 + 1
largest_number = pow(2, binary_dim)
# largest_numberまで2進数を用意
binary = np.unpackbits(np.array([range(largest_number)]), dtype=np.uint8).T, axis=1

input_layer_size = 2
hidden_layer_size = 16
output_layer_size = 1

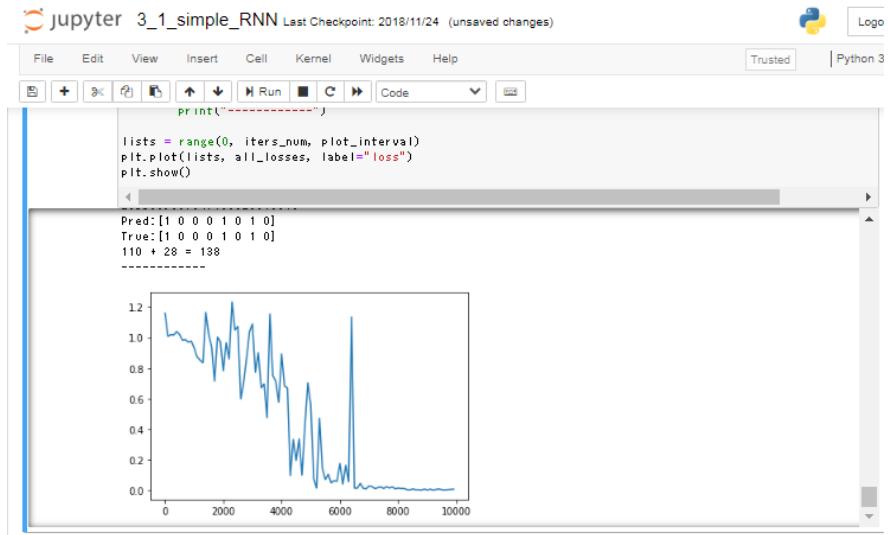
weight_init_std = 1
learning_rate = 0.1

iter_num = 10000
plot_interval = 100

# ウェイト初期化 (ハイアスは簡単のため省略)
# W_in = weight_init_std * np.random.randn(input_layer_size, hidden_layer_size)
# W_out = weight_init_std * np.random.randn(hidden_layer_size, output_layer_size)
# W = weight_init_std * np.random.randn(hidden_layer_size, hidden_layer_size)

# He
W_in = np.random.randn(input_layer_size, hidden_layer_size) / (np.sqrt(input_layer_size)) * np.sqrt(2)
W_out = np.random.randn(hidden_layer_size, output_layer_size) / (np.sqrt(hidden_layer_size)) * np.sqrt(2)
W = np.random.randn(hidden_layer_size, hidden_layer_size) / (np.sqrt(hidden_layer_size)) * np.sqrt(2)

```



中間層の活性化関数を Sigmoid から ReLU に変更。逆伝播の部分も。

The screenshot shows a Jupyter Notebook interface with a Python 3 kernel. The code cell contains a large block of Python code for an RNN. The code includes sections for forward pass, backward pass (gradient calculation), and weight update. The code uses various functions like `functions.sigmoid`, `functions.relu`, and `functions.d_mean_squared_error`. The `# 逆伝播` section is highlighted in purple, showing the computation of gradients for the backward pass. The `# 重み更新` section at the bottom shows the update logic for weights `W_in` and `W_out`.

```

# 時系列全体の誤差
all_loss = 0

# 時系列ループ
for t in range(binary_dim):
    # 入力層
    X = np.array([a_bin[-t-1], b_bin[-t-1]]).reshape(1, -1)
    # 時刻tにおける正解データ
    dd = np.array([d_bin[binary_dim - t - 1]])

    u[:,t+1] = np.dot(X, W_in) * np.dot(z[:,t].reshape(1, -1), W)
    z[:,t+1] = functions.sigmoid(u[:,t+1])
    z[:,t+1] = functions.relu(u[:,t+1])

    y[:,t] = functions.sigmoid(np.dot(z[:,t+1].reshape(1, -1), W_out))

    #誤差
    loss = functions.mean_squared_error(dd, y[:,t])

    delta_out[:,t] = functions.d_mean_squared_error(dd, y[:,t]) * functions.d_sigmoid(y[:,t])

    all_loss += loss
    out_bin[binary_dim - t - 1] = np.round(y[:,t])

    for t in range(binary_dim)[-1:]:
        X = np.array([a_bin[-t-1], b_bin[-t-1]]).reshape(1, -1)

    # delta[t, t] = (np.dot(delta[t, t+1].T, W_D) + np.dot(delta_out[:, t].T, W_out.D)) * functions.d_sigmoid(u[:, t+1])
    delta[:,t] = (np.dot(delta[:,t+1].T, W_T) + np.dot(delta_out[:,t].T, W_out.T)) * functions.d_relu(u[:,t+1])

    # 重み更新
    W_out_grad += np.dot(z[:,t+1].reshape(-1,1), delta_out[:,t].reshape(-1,1))
    W_grad += np.dot(z[:,t].reshape(-1,1), delta[:,t].reshape(1,-1))
    W_in_grad += np.dot(X.T, delta[:,t].reshape(1,-1))

    # 重み適用
    W_in -= learning_rate * W_in_grad
    W_out -= learning_rate * W_out_grad

```

jupyter 3_1_simple_RNN Last Checkpoint: 3分前 (autosaved)

File Edit View Insert Cell Kernel Widgets Help

Trusted Python 3

```

if(i % plot_interval == 0):
    all_losses.append(all_loss)
    print("iters:" + str(i))
    print("Loss:" + str(all_loss))
    print("Pred:" + str(out_bin))
    print("True:" + str(d_bin))
    out_int = 0
    for index,x in enumerate(reversed(out_bin)):
        out_int += x * pow(2, index)
    print(str(a_int) + " + " + str(b_int) + " = " + str(out_int))
    print("-----")

lists = range(0, iters_num, plot_interval)
plt.plot(lists, all_losses, label="loss")
plt.show()

```

```

Pred:[0 0 0 0 0 0 0]
True:[1 0 1 0 1 1 1 0]
121 + 53 = 0
-----

```

勾配爆発する

中間層の活性化関数を tanh に変更。逆伝播の部分も。

d_tanh は関数定義する。

jupyter 3_1_simple_RNN Last Checkpoint: 1分前 (unsaved changes)

File Edit View Insert Cell Kernel Widgets Help

Trusted Python 3

バイナリ加算

```

In [1]: import sys, os
sys.path.append(os.pardir) # 祖ディレクトリのファイルをインポートするための設定
import numpy as np
from common import functions
import matplotlib.pyplot as plt

# def d_tanh(x):
#     return 1/(np.cosh(x) ** 2)

# データを用意
# 2進数の桁数
binary_dim = 8
# 最大値 + 1
largest_number = pow(2, binary_dim)
# largest_numberまで2進数を用意
binary = np.unpackbits(np.array([range(largest_number)]), dtype=np.uint8).T, axis=1

input_layer_size = 2
hidden_layer_size = 16
output_layer_size = 1

weight_init_std = 1
learning_rate = 0.1

iters_num = 10000
plot_interval = 100

```

jupyter 3_1_simple_RNN Last Checkpoint: 1分前 (unsaved changes)

File Edit View Insert Cell Kernel Widgets Help Trusted Python 3 | Logout

```
# 時系列ループ
for t in range(binary_dim):
    # 入力層
    X = np.array([a_bin[-t-1], b_bin[-t-1]]).reshape(1, -1)
    # 隠層における正解データ
    dd = np.array([d_bin[binary_dim - t - 1]])

    u[:,t+1] = np.dot(X, W_in) + np.dot(z[:,t].reshape(1, -1), W)
    z[:,t+1] = functions.sigmoid(u[:,t+1])
    # z[:,t+1] = functions.relu(u[:,t+1])
    z[:,t+1] = np.tanh(u[:,t+1])

    y[:,t] = functions.sigmoid(np.dot(z[:,t+1].reshape(1, -1), W_out))

    # バックプロパゲーション
    loss = functions.mean_squared_error(dd, y[:,t])

    delta_out[:,t] = functions.d_mean_squared_error(dd, y[:,t]) * functions.d_sigmoid(y[:,t])

    all_loss += loss

    out_bin[binary_dim - t - 1] = np.round(y[:,t])

# 逆伝播
for t in range(binary_dim)[::-1]:
    X = np.array([a_bin[-t-1], b_bin[-t-1]]).reshape(1, -1)

    delta[:,t] = (np.dot(delta[t+1], T) * D + np.dot(de/ta_out[:,t], T) * d_out * D) * functions.d_sigmoid(u[:,t+1])
    de/ta[:,t] = (np.dot(de/ta[:,t+1], T) * D + np.dot(de/ta_out[:,t], T) * d_out * D) * functions.d_relu(u[:,t+1])
    delta[:,t] = (np.dot(de/ta[:,t+1], T) * W) * np.dot(delta_out[:,t], W_out.T) * d_tanh(u[:,t+1])
```

jupyter 3_1_simple_RNN Last Checkpoint: 1分前 (unsaved changes)

File Edit View Insert Cell Kernel Widgets Help Trusted Python 3 | Logout

```
all_losses.append(all_loss)
print("iters:" + str(i))
print("Loss:" + str(all_loss))
print("Pred:" + str(out_bin))
print("True:" + str(d_bin))
out_int = 0
for index,x in enumerate(reversed(out_bin)):
    out_int += x * pow(2, index)
    print(str(x) + " + " + str(b_int) + " = " + str(out_int))
    print("-----")

lists = range(0, iters_num, plot_intervals)
plt.plot(lists, all_losses, label="loss")
plt.show()
```

Pred:[1 1 1 1 0 1 1]
True:[1 1 0 0 0 1 1]
121 + 74 = 251

勾配がなくなる。

2.1.2 BPTT

2.1.2.1 BPTT とは

誤差逆伝播法の一種。

誤差逆伝播法を復習。

2.1.2.2 BPTT の数学的記述

$$\frac{\partial E}{\partial W_{(in)}} = \frac{\partial E}{\partial u^t} \left[\frac{\partial u^t}{\partial W_{(in)}} \right]^T = \delta^t [x^t]^T$$
$$\frac{\partial E}{\partial W_{(out)}} = \frac{\partial E}{\partial v^t} \left[\frac{\partial v^t}{\partial W_{(out)}} \right]^T = \delta^{out,t} [z^t]^T$$
$$\frac{\partial E}{\partial W} = \frac{\partial E}{\partial u^t} \left[\frac{\partial u^t}{\partial W} \right]^T = \delta^t [z^{t-1}]^T$$
$$\frac{\partial E}{\partial b} = \frac{\partial E}{\partial u^t} \frac{\partial u^t}{\partial b} = \delta^t$$
$$\frac{\partial E}{\partial c} = \frac{\partial E}{\partial v^t} \frac{\partial v^t}{\partial c} = \delta^{out,t}$$

誤差をパラメータで微分。 2つのバイアス。

$\frac{\partial E}{\partial W_{(in)}} = \frac{\partial E}{\partial u^t} \left[\frac{\partial u^t}{\partial W_{(in)}} \right]^T = \delta^t [x^t]^T$
<code>np.dot(X.T, delta[:,t].reshape(1,-1))</code>
$\frac{\partial E}{\partial W_{(out)}} = \frac{\partial E}{\partial v^t} \left[\frac{\partial v^t}{\partial W_{(out)}} \right]^T = \delta^{out,t} [z^t]^T$
<code>np.dot(z[:,t+1].reshape(-1,1), delta_out[:,t].reshape(-1,1))</code>
$\frac{\partial E}{\partial W} = \frac{\partial E}{\partial u^t} \left[\frac{\partial u^t}{\partial W} \right]^T = \delta^t [z^{t-1}]^T$
<code>np.dot(z[:,t].reshape(-1,1), delta[:,t].reshape(1,-1))</code>

内積をとっていく。

$$\frac{\partial E}{\partial b} = \frac{\partial E}{\partial u^t} \frac{\partial u^t}{\partial b} = \delta^t$$

コード記載なし（本講座のsimple RNNコードでは簡略化のため省略）

$$\frac{\partial E}{\partial c} = \frac{\partial E}{\partial v^t} \frac{\partial v^t}{\partial c} = \delta^{out,t}$$

コード記載なし（本講座のsimple RNNコードでは簡略化のため省略）

誤差をバイアスで微分したもの、のちに確認。

u に関する確認

$$\begin{aligned} u^t &= W_{(in)}x^t + W z^{t-1} + b \\ z^t &= f(W_{(in)}x^t + W z^{t-1} + b) \\ v^t &= W_{(out)}z^t + c \\ y^t &= g(W_{(out)}z^t + c) \end{aligned}$$

前のかくれ層に対して W をかけている。

$$u^t = W_{(in)}x^t + W z^{t-1} + b$$

```
u[:,t+1] = np.dot(X, W_in) + np.dot(z[:,t].reshape(1, -1), W)
```

$$z^t = f(W_{(in)}x^t + W z^{t-1} + b)$$

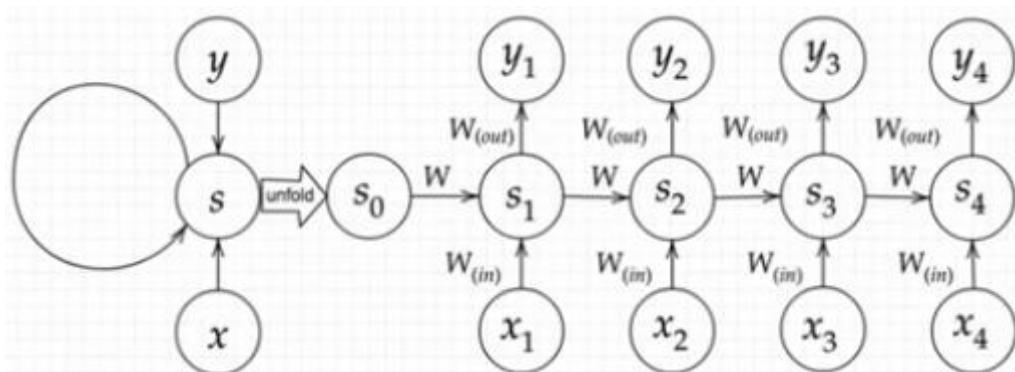
```
z[:,t+1] = functions.sigmoid(u[:,t+1])
```

$$v^t = W_{(out)}z^t + c$$

```
np.dot(z[:,t+1].reshape(1, -1), W_out)
```

$$y^t = g(W_{(out)}z^t + c)$$

```
y[:,t] = functions.sigmoid(np.dot(z[:,t+1].reshape(1, -1), W_out))
```



中間層の出力に Sigmoid 活性化関数を用いる。(出力層も)

$$z_1 = \text{Sigmoid}(S_0W + x_1W(\text{in}) + b)$$

$$y_1 = \text{Sigmoid}(z_1W(\text{out}) + c)$$

$$\frac{\partial E}{\partial u^t} = \frac{\partial E}{\partial v^t} \frac{\partial v^t}{\partial u^t} = \frac{\partial E}{\partial v^t} \frac{\partial \{W_{(out)}f(u^t) + c\}}{\partial u^t} = f'(u^t) W_{(out)}^T \delta^{out,t} = \delta^t$$

$$\delta^{t-1} = \frac{\partial E}{\partial u^{t-1}} = \frac{\partial E}{\partial u^t} \frac{\partial u^t}{\partial u^{t-1}} = \delta^t \left\{ \frac{\partial u^t}{\partial z^{t-1}} \frac{\partial z^{t-1}}{\partial u^{t-1}} \right\} = \delta^t \{ Wf'(u^{t-1}) \}$$

$$\delta^{t-z-1} = \delta^{t-z} \{ Wf'(u^{t-z-1}) \}$$

```
delta[:,t] = (np.dot(delta[:,t+1].T, W.T) + np.dot(delta_out[:,t].T, W_out.T)) * functions.d_sigmoid(u[:,t+1])
```

パラメータの更新、誤差逆伝播法とほぼ同じように。

$$W_{(in)}^{t+1} = W_{(in)}^t - \epsilon \frac{\partial E}{\partial W_{(in)}} = W_{(in)}^t - \epsilon \sum_{z=0}^{T_t} \delta^{t-z} [x^{t-z}]^T$$

$$W_{(out)}^{t+1} = W_{(out)}^t - \epsilon \frac{\partial E}{\partial W_{(out)}} = W_{(in)}^t - \epsilon \delta^{out,t} [z^t]^T$$

$$W^{t+1} = W^t - \epsilon \frac{\partial E}{\partial W} = W_{(in)}^t - \epsilon \sum_{z=0}^{T_t} \delta^{t-z} [z^{t-z-1}]^T$$

$$b^{t+1} = b^t - \epsilon \frac{\partial E}{\partial b} = b^t - \epsilon \sum_{z=0}^{T_t} \delta^{t-z}$$

$$c^{t+1} = c^t - \epsilon \frac{\partial E}{\partial c} = c^t - \epsilon \delta^{out,t}$$

```

W_{(in)}^{t+1} = W_{(in)}^t - \epsilon \frac{\partial E}{\partial W_{(in)}} = W_{(in)}^t - \epsilon \sum_{z=0}^{T_t} \delta^{t-z} [x^{t-z}]^T
W_in -= learning_rate * W_in_grad

```

```

W_{(out)}^{t+1} = W_{(out)}^t - \epsilon \frac{\partial E}{\partial W_{(out)}} = W_{(in)}^t - \epsilon \delta^{out,t} [z^t]^T
W_out -= learning_rate * W_out_grad

```

```

W^{t+1} = W^t - \epsilon \frac{\partial E}{\partial W} = W_{(in)}^t - \epsilon \sum_{z=0}^{T_t} \delta^{t-z} [z^{t-z-1}]^T
W -= learning_rate * W_grad

```

2.1.2.3 BPTT の全体像

誤差の書き方で、下記のようにも書かれる。

$$\begin{aligned} E^t &= \text{loss}(y^t, d^t) \\ &= \text{loss}\left(g(W_{(out)}z^t + c), d^t\right) \\ &= \text{loss}\left(g\left(W_{(out)}\underline{f\left(W_{(in)}x^t + W z^{t-1} + b\right)} + c\right), d^t\right) \end{aligned}$$

↓

$$\begin{aligned} &W_{(in)}x^t + W z^{t-1} + b \\ &W_{(in)}x^t + W f(u^{t-1}) + b \\ &W_{(in)}x^t + W f\left(W_{(in)}x^{t-1} + W z^{t-2} + b\right) + b \end{aligned}$$

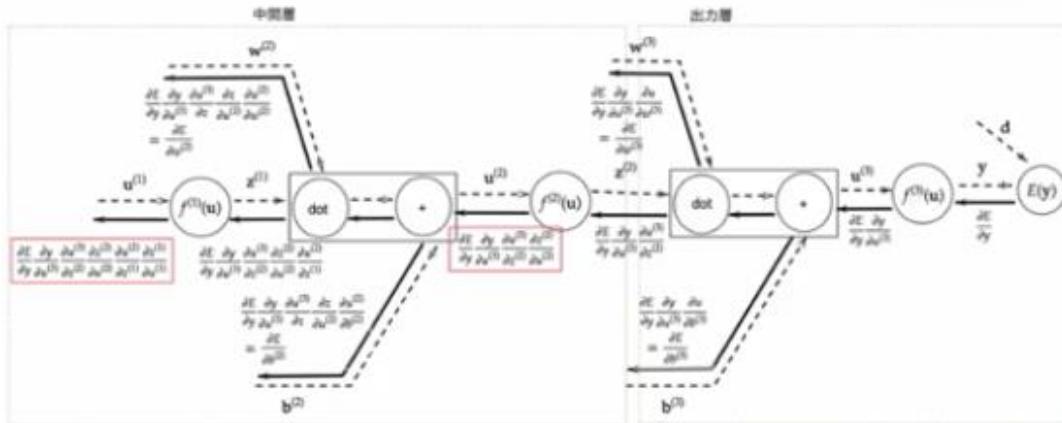
展開している。

2.2. LSTM

2.2.1 全体像（前回の流れと課題全体像のビジョン）

RNN の課題：時系列を遡るほど、勾配が消失していく。長い時系列の学習は困難。
この課題を、RNN の構造を変えて解決したものが LSTM となる。

勾配消失問題：誤差逆伝播法が下位層に進んでいくにつれて、勾配がゆるやかになる。
勾配降下法による更新では、下位層のパラメータが
ほとんど変わらずに訓練が最適値に収束しなくなる。



$$\begin{aligned} \frac{\partial E}{\partial y} \frac{\partial y}{\partial u^{(3)}} \frac{\partial u^{(3)}}{\partial z^{(2)}} \frac{\partial z^{(2)}}{\partial u^{(2)}} & \xrightarrow{(1 - \text{sigmoid}(x)) \cdot \text{sigmoid}(x)} \\ \frac{\partial E}{\partial y} \frac{\partial y}{\partial u^{(3)}} \frac{\partial u^{(3)}}{\partial z^{(2)}} \frac{\partial z^{(2)}}{\partial u^{(2)}} \frac{\partial u^{(2)}}{\partial z^{(1)}} \frac{\partial z^{(1)}}{\partial u^{(1)}} & \xrightarrow{(1 - \text{sigmoid}(x)) \cdot \text{sigmoid}(x)} \\ & \xrightarrow{(1 - \text{sigmoid}(x)) \cdot \text{sigmoid}(x)} \end{aligned}$$

勾配爆発問題：勾配が、層を逆伝播するごとに指数関数的に大きくなる。

RNN や層の深いモデルでは、勾配消失や勾配爆発が起こる傾向がある。

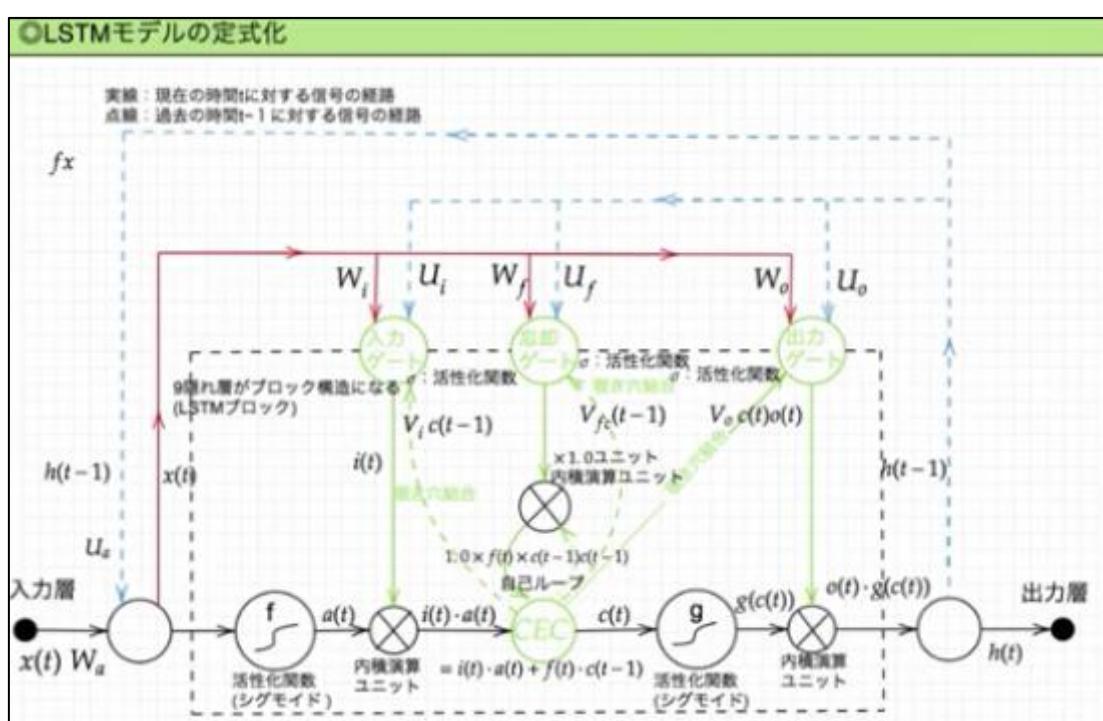
勾配爆発を防ぐためには勾配のクリッピングを行う手法がある。

- ・勾配のノルムがしきい値を超えたら、

勾配のノルムをしきい値に正規化することで防ぐ。

- ・勾配のノルムがしきい値より大きい時、勾配のノルムをしきい値で正規化するので、クリッピングした勾配は、 $\text{勾配} \times (\text{しきい値} \div \text{勾配のノルム})$ となる。

$$\text{gradient} \times \text{threshold} / \text{norm}$$



2.2.2 CEC

勾配消失および勾配爆発の解決のため、勾配が 1 であれば解決。

$$\delta^{t-z-1} = \delta^{t-z} \left\{ Wf'(u^{t-z-1}) \right\} = 1$$

$$\boxed{\frac{\partial E}{\partial c^{t-1}}} = \frac{\partial E}{\partial c^t} \frac{\partial c^t}{\partial c^{t-1}} = \frac{\partial E}{\partial c^t} \frac{\partial}{\partial c^{t-1}} [a^t - c^{t-1}] = \boxed{\frac{\partial E}{\partial c^t}}$$

課題：入力データの時間依存度に関係なく、重みが一律になってしまふ。

ニューラルネットワークの学習特性が無い。

入力層→隠れ層への重み : 入力重みの衝突。

隠れ層→出力層への重み : 出力重みの衝突。

2.2.3 入力ゲートと出力ゲート

入力ゲートと出力ゲートを追加することにより、
それぞれのゲートへの入力値の重みを重み行列 W, U で可変可能とすることで、
CEC の課題を解決。

2.2.4 忘却ゲート

CEC には過去の情報が全て保管されてしまいます。キャッシュのように。
過去のデータに影響を受け続けてしまう。
過去の情報は要らなくなった時点で、情報を忘れる必要がある。

LSTM の順伝播において、新しいセルの状態は、
計算されたセルへの入力と 1 ステップ前のセルの状態に、
入力ゲート、忘却ゲートを掛けて足したものとなる。

$$c = \text{input_gate} * a + \text{forget_gate} * c$$

2.2.5 覗き穴結合

CEC に保存されている過去の情報を、任意のタイミングで他のノードに伝播させたり、
あるいは任意のタイミングで忘却させたい。
CEC 自身の値は、ゲート制御に影響を与えていない。
覗き穴結合では、CEC 自身の値に、重み行列を介して伝播可能にした構造。

演習チャレンジ 1

演習チャレンジ

RNNや深いモデルでは勾配の消失または爆発が起こる傾向がある。勾配爆発を防ぐために勾配のクリッピングを行うという手法がある。具体的には勾配のノルムがしきい値を超えたら、勾配のノルムをしきい値に正規化するというものである。以下は勾配のクリッピングを行う関数である。

(さ) にあてはまるのはどれか。

```
def gradient_clipping(grad, threshold):
    .....
    grad: gradient
    .....
    norm = np.linalg.norm(grad)
    rate = threshold / norm
    if rate < 1:
        return (さ)
    return grad
```

- (1) gradient * rate
- (2) gradient / norm
- (3) gradient / threshold
- (4) np.maximum(gradient, threshold)

(さ)には、(1)が入る。

RNN や層の深いモデルでは、勾配消失や勾配爆発が起こる傾向がある。

勾配爆発を防ぐためには勾配のクリッピングを行う手法がある。

- ・勾配のノルムがしきい値を超えたら、
勾配のノルムをしきい値に正規化することで防ぐ。
- ・勾配のノルムがしきい値より大きい時、勾配のノルムをしきい値で正規化するので、
クリッピングした勾配は、勾配 \times (しきい値 ÷ 勾配のノルム)となる。

$$\text{gradient} \times \text{threshold} / \text{norm}$$

演習チャレンジ 2

演習チャレンジ

以下のプログラムはLSTMの順伝播を行うプログラムである。ただし_sigmoid関数は要素ごとにシグモイド関数を作用させる関数である。
(け)にあてはまるのはどれか。

```
def lstm(x, prev_h, prev_c, W, U, b):
    """
    x: inputs, (batch_size, input_size)
    prev_h: outputs at the previous time step, (batch_size, state_size)
    prev_c: cell states at the previous time step, (batch_size, state_size)
    W: upward weights, (4*state_size, input_size)
    U: lateral weights, (4*state_size, state_size)
    b: bias, (4*state_size,)
    """

    # セルへの入力ゲートをまとめて計算し、分離
    lstm_in = activation(x.dot(W.T) + prev_h.dot(U.T) + b)
    i, f, o, g = np.hsplit(lstm_in, 4)

    # 狙を更新。セルへの入力(i, f), グート(c, o)を用いて
    a = np.tanh(g)
    input_gate = _sigmoid(i)
    forget_gate = _sigmoid(f)
    output_gate = _sigmoid(o)

    # セルの状態を更新し、中間層の出力を計算
    c = [g] # (け)
    h = output_gate * np.tanh(c)
    return c, h
```

- (1) $output_gate * a + forget_gate * c$
- (2) $forget_gate * a + output_gate * c$
- (3) $input_gate * a + forget_gate * c$
- (4) $forget_gate * a + input_gate * c$

(け)には(3)が入れる。

LSTM の順伝播において、新しいセルの状態は、
計算されたセルへの入力と 1 ステップ前のセルの状態に、
入力ゲート、忘却ゲートを掛けて足したものとなる。

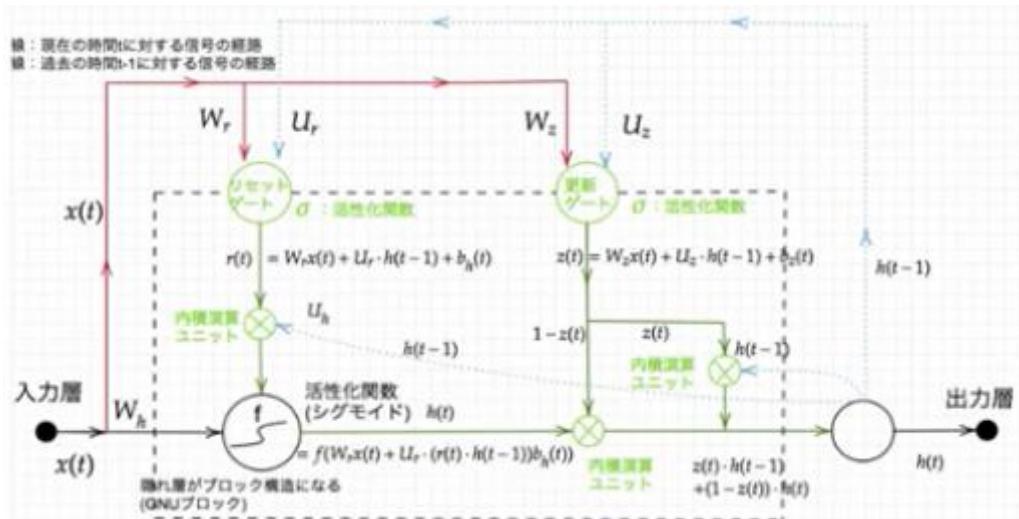
$$c = input_gate * a + forget_gate * c$$

2.3. GRU (Gated Recurrent Unit)

LSTM の課題：パラメータ数が多く、計算負荷が高くなる。時間がかかる。

GRU では、パラメータを大幅に削減し、精度は同等以上となった構造。

ただし、必ずしも上位互換ではない。計算負荷は低い。



GRU も RNN の一種であり、勾配消失問題は解決し、長期的な依存関係を学習可能。

GRU の順伝播において、新しい中間状態は、1 ステップ前の中間表現と計算された中間表現の線形和で表現される。更新ゲート z を用いて次のようにになる。

$$h_{\text{bar}} = \tanh(x \cdot W.T + (r * h) \cdot U.T)$$

$$h_{\text{new}} = (1 - z) * h + z * h_{\text{bar}}$$

x : 入力

h : 1 ステップ前の出力。

r : リセットゲート、 z : 更新ゲート

LSTM と GRU の違い：

パラメータの数が、LSTM は多い、GRU は少ない。

← → C arxiv.org/pdf/1406.1078.pdf

1406.1078.pdf 1 / 15

Learning Phrase Representations using RNN Encoder–Decoder
for Statistical Machine Translation

Kyunghyun Cho
Bart van Merriënboer Caglar Gulcehre
Université de Montréal
firstname.lastname@umontreal.ca

Dzmitry Bahdanau
Jacobs University, Germany
d.bahdanau@jacobs-university.de

Fethi Bougares Holger Schwenk
Université du Maine, France Université de Montréal, CIFAR Senior Fellow
firstname.lastname@lum.univ-lemans.fr find.me@on.the.web

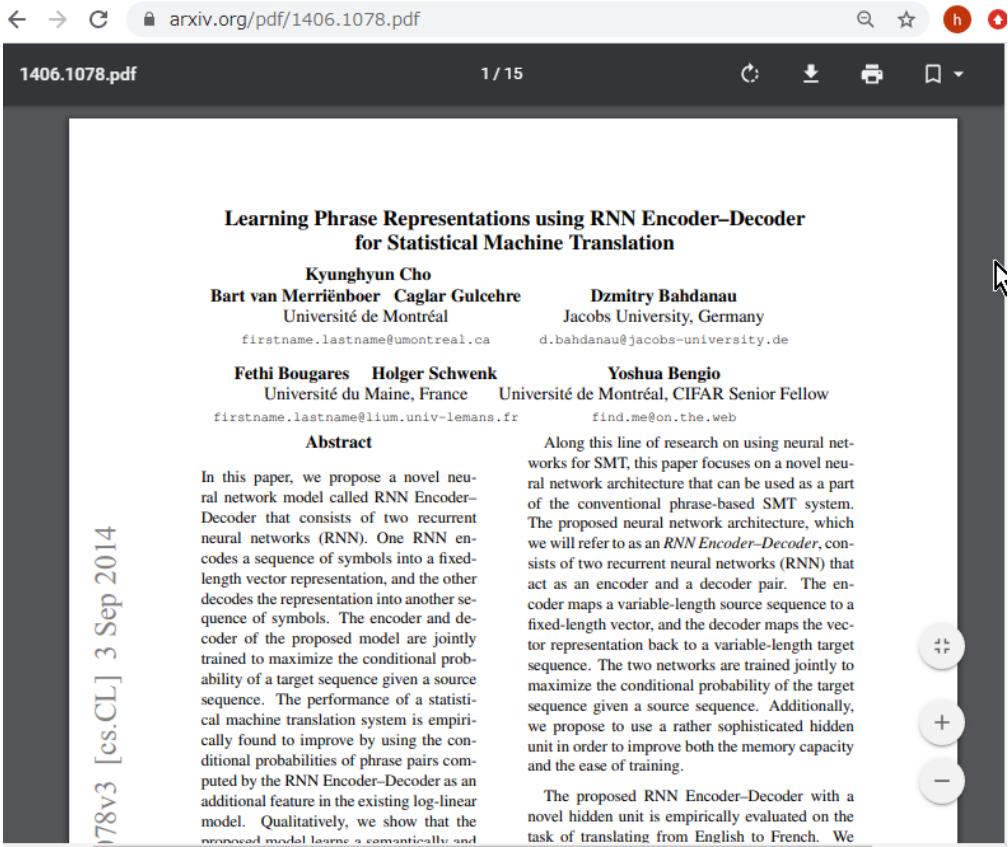
Abstract

In this paper, we propose a novel neural network model called RNN Encoder–Decoder that consists of two recurrent neural networks (RNN). One RNN encodes a sequence of symbols into a fixed-length vector representation, and the other decodes the representation into another sequence of symbols. The encoder and decoder of the proposed model are jointly trained to maximize the conditional probability of a target sequence given a source sequence. The performance of a statistical machine translation system is empirically found to improve by using the conditional probabilities of phrase pairs computed by the RNN Encoder–Decoder as an additional feature in the existing log-linear model. Qualitatively, we show that the proposed model learns a semantically and

Along this line of research on using neural networks for SMT, this paper focuses on a novel neural network architecture that can be used as a part of the conventional phrase-based SMT system. The proposed neural network architecture, which we will refer to as an *RNN Encoder–Decoder*, consists of two recurrent neural networks (RNN) that act as an encoder and a decoder pair. The encoder maps a variable-length source sequence to a fixed-length vector, and the decoder maps the vector representation back to a variable-length target sequence. The two networks are trained jointly to maximize the conditional probability of the target sequence given a source sequence. Additionally, we propose to use a rather sophisticated hidden unit in order to improve both the memory capacity and the ease of training.

The proposed RNN Encoder–Decoder with a novel hidden unit is empirically evaluated on the task of translating from English to French. We

1406.1078v3 [cs.CL] 3 Sep 2014



GRU の元の論文を確認した。[6]

演習チャレンジ

演習チャレンジ

GRU(Gated Recurrent Unit)もLSTMと同様にRNNの一種であり、単純なRNNにおいて問題となる勾配消失問題を解決し、長期的な依存関係を学習することができる。LSTMに比べ変数の数やゲートの数が少なく、より単純なモデルであるが、タスクによってはLSTMより良い性能を発揮する。以下のプログラムはGRUの順伝播を行うプログラムである。ただし sigmoid 関数は要素ごとにシグモイド関数を作用させる関数である。

(c) にあてはまるのはどれか。

```
def gru(x, h, W_r, U_r, W_z, U_z, W, U):
    """
    x: inputs, (batch_size, input_size)
    h: outputs at the previous time step, (batch_size, state_size)
    W_r, U_r: weights for reset gate
    W_z, U_z: weights for update gate
    U, W: weights for new state
    """
    # ゲートを計算
    r = _sigmoid(x.dot(W_r.T) + h.dot(U_r.T))
    z = _sigmoid(x.dot(W_z.T) + h.dot(U_z.T))

    # 次状態を計算
    h_bar = np.tanh(x.dot(W.T) + (r * h).dot(U.T))
    h_new = (c)
    return h_new
```

(1) $z * h_{\bar{h}}$
(2) $(1-z) * h_{\bar{h}}$
(3) $z * h * h_{\bar{h}}$
(4) $(1-z) * h + z * h_{\bar{h}}$

(c)には(4)を入れる。

GRU の順伝播において、新しい中間状態は、1ステップ前の中間表現と計算された中間表現の線形和で表現される。更新ゲート z を用いて次のようになる。

$$h_{\bar{h}} = \tanh(x \cdot W.T + (r * h) \cdot U.T)$$

$$h_{\text{new}} = (1-z) * h + z * h_{\bar{h}}$$

x:入力

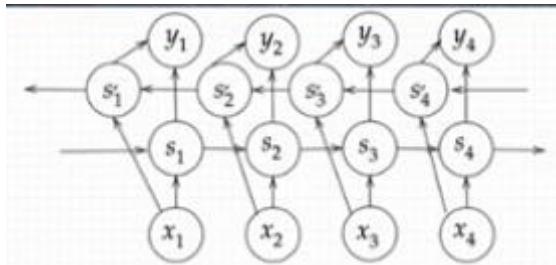
h:1ステップ前の出力。

r:リセットゲート、z:更新ゲート

2.4. 双方向 RNN

過去の情報だけでなく、未来の情報を加味して、精度を向上させるモデル

例) 文章推敲、機械翻訳



双方向 RNNにおいては次のようにする。

W_f : 入力から中間層への重み

U_f : 1 ステップ前の中間層出力から中間層への重み

W_b : 逆方向の重み

U_b : 逆方向の重み

V : 両者の中間層表現を合わせた特徴から出力層への重み

順方向と逆方向に伝播したとき、中間層表現を合わせたものが特徴量となる。

$hs = [np.concatenate([h_f, h_b[::-1]], axis=1) \text{ for } h_f, h_b \text{ in } zip(hs_f, hs_b)]$

`np.concatenate` を使った配列同士の結合。

演習チャレンジ

演習チャレンジ

以下は双方向RNNの順伝播を行うプログラムである。順方向については、入力から中間層への重み W_f 、一ステップ前の中間層出力から中間層への重みを U_f 、逆方向に関する場合は同様にパラメータ W_b , U_b を持ち、両者の中間層表現を合わせた特徴から出力層への重みは V である。 $_rnn$ 関数はRNNの順伝播を表し中間層の系列を返す関数であるとする。(か)にあてはまるのはどれか

```
def bidirectional_rnn_net(xs, W_f, U_f, W_b, U_b, V):
    """
    W_f, U_f: forward rnn weights, (hidden_size, input_size)
    W_b, U_b: backward rnn weights, (hidden_size, input_size)
    V: output weights, (output_size, 2*hidden_size)
    """
    xs_f = np.zeros_like(xs)
    xs_b = np.zeros_like(xs)
    for i, x in enumerate(xs):
        xs_f[i] = x
        xs_b[i] = x[::-1]
    hs_f = _rnn(xs_f, W_f, U_f)
    hs_b = _rnn(xs_b, W_b, U_b)
    hs = [hs_f[i] + hs_b[i] for h_f, h_b in zip(hs_f, hs_b)]
    ys = hs.dot(V.T)
    return ys
```

- (1) $h_f + h_b[:: -1]$
- (2) $h_f * h_b[:: -1]$
- (3) $\text{np.concatenate}([h_f, h_b[:: -1]], \text{axis}=0)$
- (4) $\text{np.concatenate}([h_f, h_b[:: -1]], \text{axis}=1)$

双方向 RNN においては次のようになる。(か)には(4)を入れる。

W_f : 入力から中間層への重み

U_f : 1 ステップ前の中間層出力から中間層への重み

W_b : 逆方向の重み

U_b : 逆方向の重み

V : 両者の中間層表現を合わせた特徴から出力層への重み

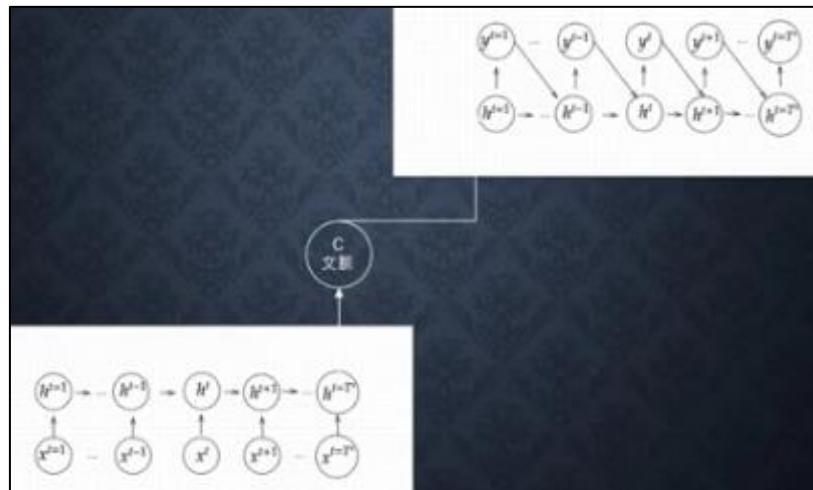
順方向と逆方向に伝播したとき、中間層表現を合わせたものが特徴量となる。

$hs = [\text{np.concatenate}([h_f, h_b[:: -1]], \text{axis}=1) \text{ for } h_f, h_b \text{ in } \text{zip}(hs_f, hs_b)]$

np.concatenate を使った配列同士の結合。

2.5. Seq2Seq RNN での自然言語処理

2.6.1 Seq2Seq 全体像



Seq2Seq とは、Encoder-Decoder モデルの一種。

例) 機械対話、機械翻訳

Sequence to Sequence Learning with Neural Networks

Ilya Sutskever
Google
ilyasu@google.com

Oriol Vinyals
Google
vinyals@google.com

Quoc V. Le
Google
qvl@google.com

Abstract

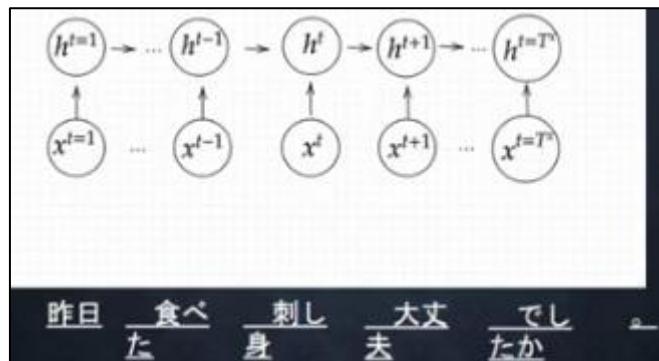
Deep Neural Networks (DNNs) are powerful models that have achieved excellent performance on difficult learning tasks. Although DNNs work well whenever large labeled training sets are available, they cannot be used to map sequences to sequences. In this paper, we present a general end-to-end approach to sequence learning that makes minimal assumptions on the sequence structure. Our method uses a multilayered Long Short-Term Memory (LSTM) to map the input sequence to a vector of a fixed dimensionality, and then another deep LSTM to decode the target sequence from the vector. Our main result is that on an English to French translation task from the WMT-14 dataset, the translations produced by the LSTM achieve a BLEU score of 34.8 on the entire test set, where the LSTM's BLEU score was penalized on out-of-vocabulary words. Additionally, the LSTM did not have difficulty on long sentences. For comparison, a phrase-based SMT system achieves a BLEU score of 33.3 on the same dataset. When we used the LSTM to rerank the 1000 hypotheses produced by the aforementioned SMT system, its BLEU score increases to 36.5, which is close to the previous state of the art. The LSTM also learned sensible phrase and sentence representations that are sensitive to word order and are relatively invariant to the active and the passive voice. Finally, we found that reversing the order of the words in all source sentences (but not target sentences) improved the LSTM's performance markedly, because doing so introduced many short term dependencies between the source and the target sentence which made the optimization problem easier.

Seq2Seq の論文を確認した。[7]

3つの特徴がある。

- (1) 勾配消失を防ぐため、入力層と出力層で異なる構造。
- (2) 4層の LSTM を使用。深い方が良い。
- (3) 入力語順を反転したものも学習。

2.6.2 Encoder RNN



テキストデータの入力を、単語等のトークンに区切って渡す構造。

Taking : 文章を単語ごとのトークン毎に分割、トークン毎の ID に分割。

Embedding : ID から、そのトークンを表す分散表現ベクトルに変換。

Encoder RNN : ベクトルを順番に RNN に入力していく。

ベクトル 1 を RNN に入力、hidden state を出力。

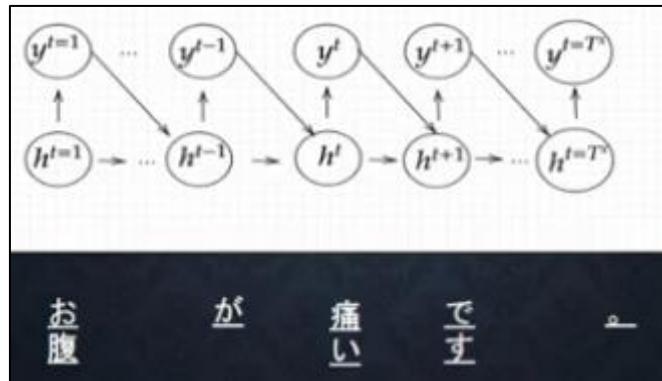
この hidden state と次のベクトル 2 を RNN に入力、hidden state を出力。繰り返す。

最後のベクトルを入力、hidden state を final state として取っておく。

この final state が thought ベクターと呼ばれ、

入力した文の意味を表すベクトルとなる。

2.6.3 Decoder RNN



Decoder RNN : システムがアウトプットデータを、
単語などのトークン毎に生成する構造。

Encoder RNN の final state(thought ベクター)から、
各トークンの生成確率を出力、final state を Decoder RNN の initial state として、
Embedding を入力。

Sampling : 生成確率にもとづきトークンをランダムに選ぶ。

Embedding: 選ばれたトークンを Embedding して Decoder RNN への次の入力とする。

Detokenize : 上記を繰り返し、得られたトークンを文字列に直していく。

演習チャレンジ

演習チャレンジ

機械翻訳タスクにおいて、入力は複数の単語から成る文（文章）であり、それぞれの単語はone-hotベクトルで表現されている。Encoderにおいて、それらの単語は単語埋め込みにより特徴量に変換され、そこからRNNによって（一般にはLSTMを使うことが多い）時系列の情報をもつ特徴へとエンコードされる。以下は、入力である文（文章）を時系列の情報をもつ特徴へとエンコードする関数である。ただし_activation関数はなんらかの活性化関数を表すとする。

(き) にあてはまるのはどれか。

```
def encode(words, E, W, U, b):
    """
    words: sequence words (sentence), one-hot vector, (n_words, vocab_size)
    E: word embedding matrix, (embed_size, vocab_size)
    W: upward weights, (hidden_size, hidden_size)
    U: lateral weights, (hidden_size, embed_size)
    b: bias, (hidden_size,)
    """

    hidden_size = W.shape[0]
    h = np.zeros(hidden_size)
    for w in words:
        e = (w)
        h = activation(W.dot(e) + U.dot(h) + b)
    return h
```

- (1) E.dot(w)
- (2) E.T.dot(w)
- (3) w.dot(E.T)
- (4) E * w

(き) には、(1)を入れる。

words から取り出した単語 w は、one-hot ベクトルで、単語埋め込みにより、特徴量に変換する。これは埋め込み行列 E を用いて $E \cdot w$ となる。

2.6.4 HRED (エイチレッド)

Seq2seq の課題：一問一答しかできない。一問一答に有意義、FAQ 等。

間に対して文脈も何もなく、応答が行われる。

HRED とは？：過去 $n-1$ 個の発話から次の発話を生成する。

前の単語の流れに即して応答がされるため、
より人間らしい文章が生成される。

Seq2Seq + Context RNN

Context RNN : Encoder のまとめた各文章の系列をまとめて、これまでの会話コンテキスト全体を表すベクトルに変換する構造。
過去の発話履歴を加味した返答ができる。

HRED の課題：確率的な多様性が字面にしかなく、
会話の「流れ」のような多様性は無い。
同じコンテキスト（発話リスト）を与えられても、
答えの内容が毎回会話の流れとしては同じものしか出ない。
短く情報量に乏しい答えを返しがち。

短いよくある答えを学ぶ傾向がある。

例) 「うん」「そうだね」「…」

2.6.5 VHRED (ブイエイチレッド)

VHREAD とは? : HRED に、VAE (バエ) の潜在変数の概念を追加。

HRED の課題の解決。(同じコンテキストが改善)

2.6.6 VAE

2.6.6.1 オートエンコーダ

オートエンコーダーとは? : 教師なし学習の1つ。

学習時の入力データは訓練データのみで、
教師データは利用しない。

例) MNIST の場合、入力データ(28x28)を入れると、
同じ画像(28x28)を出力する。

オートエンコーダー構造:

入力データから潜在変数 z に変換する Encoder。

逆に潜在変数 z をインプットとして、元のデータに復元する Decoder。

オートエンコーダーのメリット: 次元削減が行える。

潜在変数 z の次元が、入力データより小さい場合、
次元削減とみなすことができる。

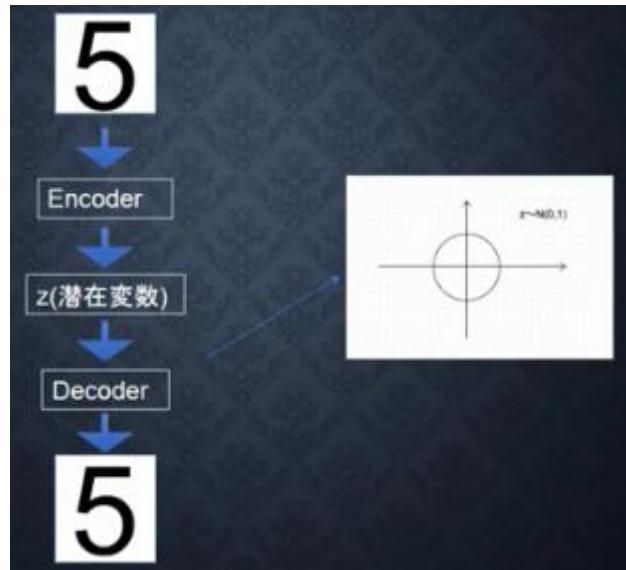


2.6.6.2 VAE (バエ)

オートエンコーダーでは何かしらの潜在変数 z にデータを押し込むものの、その構造がどのような状態かわからない。

VAE はこの潜在変数 z に確率分布 $z \sim N(0,1)$ を仮定。

データを潜在変数 z の確率分布という構造に押し込む。



2.6. word2vec

単語のような可変長の文字列を RNN に与えることはできない。

固定長形式で単語を表す

学習データからボキャブラリを作成、辞書の単語数できる。

入力層には、辞書の単語数だけ one-hot ベクトルができる。

word2vec のメリット：大規模データの分散表現の学習が、現実的な計算速度とメモリ量で実現可能にした。

ボキャブラリ × 任意の単語ベクトル次元で重み行列を生成。

(以前はボキャブラリ × ボキャブラリの重み行列だった)

arxiv.org/pdf/1301.3781.pdf

1 / 12

Efficient Estimation of Word Representations in Vector Space

Tomas Mikolov
Google Inc., Mountain View, CA
tmikolov@google.com

Kai Chen
Google Inc., Mountain View, CA
kaichen@google.com

Greg Corrado
Google Inc., Mountain View, CA
gcorrado@google.com

Jeffrey Dean
Google Inc., Mountain View, CA
jeff@google.com

Abstract

We propose two novel model architectures for computing continuous vector representations of words from very large data sets. The quality of these representations is measured in a word similarity task, and the results are compared to the previously best performing techniques based on different types of neural networks. We observe large improvements in accuracy at much lower computational cost, i.e. it takes less than a day to learn high quality word vectors from a 1.6 billion words data set. Furthermore, we show that these vectors provide state-of-the-art performance on our test set for measuring syntactic and semantic word similarities.

Xiv:1301.3781v3 [cs.CL] 7 Sep 2013

word2vec の論文を確認した。[8]

word2vec では 2 つのモデルを使っている。

- CBOW(continuous bag-of-words)
- skip-gram

2.7. Attention Mechanism

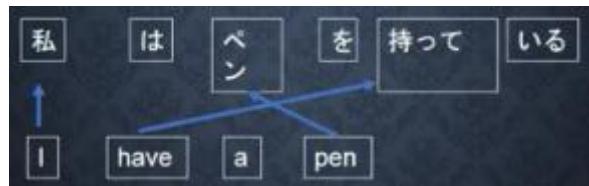
seq2seq の課題：長い文章への対応が困難。2 語でも 100 語でも、

固定次元ベクトルの中で入力しなくてはならない。

文章が長くなるほどそのシーケンスの内部表現の次元も大きくなっていく仕組みが必要。

Attention Machanism (アテンションメカニズム) :

「入力と出力のどの単語が関連しているのか」の関連度を学習する。



長い文章を入れても、翻訳がなりたつ。

演習チャレンジ

演習チャレンジ

以下は再帰型ニューラルネットワークにおいて構文木を入力として再帰的に文全体の表現ベクトルを得るプログラムである。ただしニューラルネットワークの重みパラメータはグローバル変数として定義してあるものとし、`_activation`関数はなんらかの活性化関数であるとする。木構造は再帰的な辞書で定義しており、`root`が最も外側の辞書であると仮定する。
（く）にあてはまるのはどれか。

```
def traverse(node):
    ...
    node: tree node, recursive dict, {left: node', right: node''}
    if leaf, word embedded vector, (embed_size,)
        ...
        W: weights, global variable, (embed_size, 2*embed_size)
        b: bias, global variable, (embed_size,)

    if not isinstance(node, dict):
        v = node
    else:
        left = traverse(node['left'])
        right = traverse(node['right'])
        v = _activation(left + right)  # <-- (く)
    return v
```

- (1) `W.dot(left + right)`
- (2) `W.dot(np.concatenate([left, right]))`
- (3) `W.dot(left * right)`
- (4) `W.dot(np.maximum(left, right))`

（く）には、（2）を入れる。

隣接単語（表現ベクトル）から表現ベクトルを作る処理は、

隣接表現 `left` と `right` を合わせたものを特徴量として、

そこに重みを掛けることで実現する。

`W.dot(np.concatenate([left, right]))`

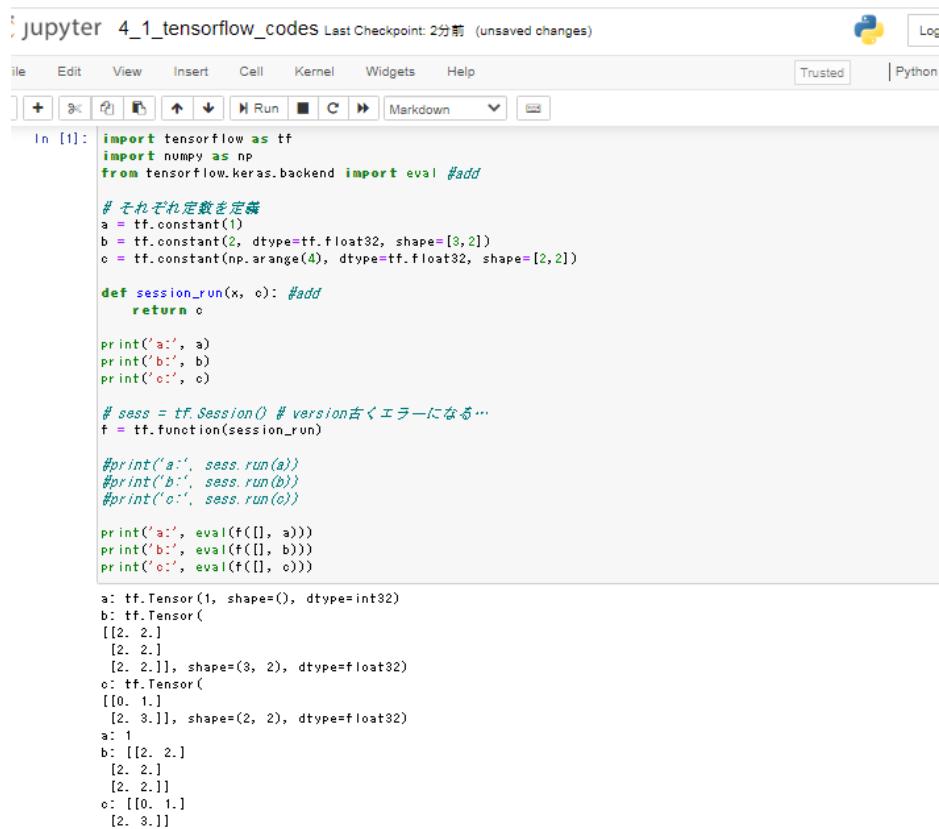
3. 深層学習（後編2）

3.1 Tensorflow の実装演習

TensorFlow : google が作った deeplearning のフレームワーク、
世界的にもユーザー数が一番多いライブラリ。

ハンズオン（実装演習）

3.1.1 Tensorflow



The screenshot shows a Jupyter Notebook interface with the following details:

- Title Bar:** jupyter 4_1_tensorflow_codes Last Checkpoint: 2分前 (unsaved changes)
- Toolbar:** File Edit View Insert Cell Kernel Widgets Help
- Status Bar:** Trusted | Python
- Cell Content:** In [1]:

```
import tensorflow as tf
import numpy as np
from tensorflow.keras.backend import eval #add

# それぞれ定数を定義
a = tf.constant(1)
b = tf.constant(2, dtype=tf.float32, shape=[3,2])
c = tf.constant(np.arange(4), dtype=tf.float32, shape=[2,2])

def session_run(x, c): #add
    return c

print('a:', a)
print('b:', b)
print('c:', c)

# sess = tf.Session() # version古くエラーになる…
f = tf.function(session_run)

#print('a:', sess.run(a))
#print('b:', sess.run(b))
#print('c:', sess.run(c))

print('a:', eval(f([], a)))
print('b:', eval(f([], b)))
print('c:', eval(f([], c)))
```
- Output:**

```
a: tf.Tensor(1, shape=(), dtype=int32)
b: tf.Tensor(
[[2. 2.]
 [2. 2.]
 [2. 2.]], shape=(3, 2), dtype=float32)
c: tf.Tensor(
[[0. 1.]
 [2. 3.]], shape=(2, 2), dtype=float32)
a: 1
b: [[2. 2.]
 [2. 2.]
 [2. 2.]]
c: [[0. 1.]
 [2. 3.]]
```

最近のバージョンでは tensor の中身も見られるようになっている。

jupyter 4_1_tensorflow_codes Last Checkpoint: 数秒前 (unsaved changes)

File Edit View Insert Cell Kernel Widgets Help Trusted Python 3

constant

```
In [4]: import tensorflow as tf
import numpy as np
#from tensorflow.keras.backend import eval #add
import tensorflow.compat.v1 as tf
tf.disable_v2_behavior()

# それぞれ定数を定義
a = tf.constant(1)
b = tf.constant(2, dtype=tf.float32, shape=[3, 2])
c = tf.constant(np.arange(4), dtype=tf.float32, shape=[2, 2])

def session_run(x, c): #add
    return c

print('a:', a)
print('b:', b)
print('c:', c)

sess = tf.Session() # version古くエラーになる...
print('a:', sess.run(a))
print('b:', sess.run(b))
print('c:', sess.run(c))

#sess = tf.function(session_run)
#print('a:', eval(sess[], a))
#print('b:', eval(sess[], b))
#print('c:', eval(sess[], c))
```

a: Tensor("Const_3:0", shape=(), dtype=int32)
b: Tensor("Const_4:0", shape=(3, 2), dtype=float32)
c: Tensor("Const_5:0", shape=(2, 2), dtype=float32)
a: 1
b: [[2., 2.]
 [2., 2.]
 [2., 2.]]
c: [[0., 1.]
 [2., 3.]]

古いバージョンで動作させることもできた。

jupyter 4_1_tensorflow_codes Last Checkpoint: 2分前 (autosaved)

File Edit View Insert Cell Kernel Widgets Help Trusted Python 3

placeholder

```
In [6]: import tensorflow as tf
import numpy as np
import tensorflow.compat.v1 as tf
tf.disable_v2_behavior()

# プレイスホルダーを定義
x = tf.placeholder(dtype=tf.float32, shape=[None, 3])

print('x:', x)

sess = tf.Session()

X = np.random.rand(2, 3)
print('X:', X)

# プレイスホルダにX[0]を入力
# shapeを(3,)から(1, 3)にするためreshape
print('x:', sess.run(x, feed_dict={x:X[0].reshape(1, -1)}))
# プレイスホルダにX[1]を入力
print('x:', sess.run(x, feed_dict={x:X[1].reshape(1, -1)}))

x: Tensor("Placeholder:0", shape=(?, 3), dtype=float32)
X: [[0.61315183 0.90332592 0.47993899]
     [0.57258604 0.67320552 0.9587367 ]]
x: [[0.61315183 0.9033259 0.47993898]]
x: [[0.57258606 0.6732055 0.9587367 ]]
```

placeholder も現在のバージョンでは無い。現在は関数の形で定義する。

jupyter 4_1_tensorflow_codes Last Checkpoint: 7分前 (unsaved changes) Logout

File Edit View Insert Cell Kernel Widgets Help Trusted Python 3

variables

```
In [7]: # 定数を定義
a = tf.constant(10)
print('a:', a)
# 変数を定義
x = tf.Variable(1)
print('x:', x)

calc_op = x * a

# xの値を更新
update_x = tf.assign(x, calc_op)

sess = tf.Session()

# 変数の初期化
init = tf.global_variables_initializer()
sess.run(init)

print(sess.run(x))

sess.run(update_x)
print(sess.run(x))

sess.run(update_x)
print(sess.run(x))

a: Tensor("Const_6:0", shape=(), dtype=int32)
x: <tf.Variable 'Variable_0' shape=() dtype=int32_ref>
1
10
100
```

3.1.2 Tensorflow、線形回帰

jupyter 4_1_tensorflow_codes Last Checkpoint: 数秒前 (autosaved) Logout

File Edit View Insert Cell Kernel Widgets Help Trusted Python 3

```
In [9]: import numpy as np
import tensorflow as tf
import matplotlib.pyplot as plt
import tensorflow.compat.v1 as tf #add
tf.disable_v2_behavior() #add

iters_num = 300
plot_interval = 10

# データを生成
n = 100
x = np.random.rand(n)
d = 3 * x + 2

# ノイズを加える
noise = 0.3
d = d + noise * np.random.randn(n)

# 入力値
xt = tf.placeholder(tf.float32)
dt = tf.placeholder(tf.float32)

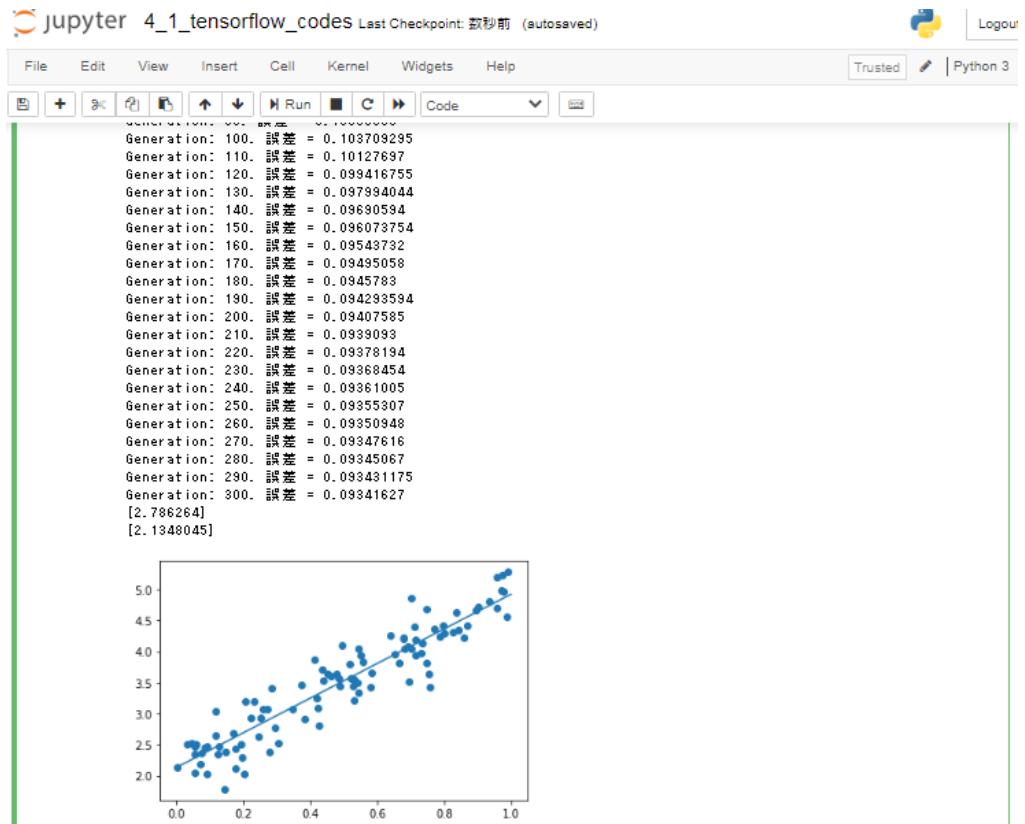
# 最適化の対象の変数を初期化
W = tf.Variable(tf.zeros([1]))
b = tf.Variable(tf.zeros([1]))

y = W * xt + b

# 誤差関数 平均2乗誤差
loss = tf.reduce_mean(tf.square(y - dt))
optimizer = tf.train.GradientDescentOptimizer(0.1)
train = optimizer.minimize(loss)

# 初期化
init = tf.global_variables_initializer()
sess = tf.Session()
sess.run(init)

# 作成したデータをトレーニングデータとして準備
x_train = x.reshape(-1, 1)
d_train = d.reshape(-1, 1)
```



noise の値を変更する。0.3 \Rightarrow 3 10 倍

The screenshot shows a Jupyter Notebook interface with the title "jupyter 4_1_tensorflow_codes". The notebook has a "Trusted" status and is running in Python 3. The code cell contains the following TensorFlow code:

```

In [13]:
import numpy as np
import tensorflow as tf
import matplotlib.pyplot as plt
import tensorflow.compat.v1 as tf #add
tf.disable_v2_behavior() #add

iters_num = 300
plot_interval = 10

#####
# データを生成
n = 100
x = np.random.rand(n)
d = 3 * x + 2

# ノイズを加える
noise = 3
d = d + noise * np.random.randn(n)
#####

# 入力値
xt = tf.placeholder(tf.float32)
dt = tf.placeholder(tf.float32)

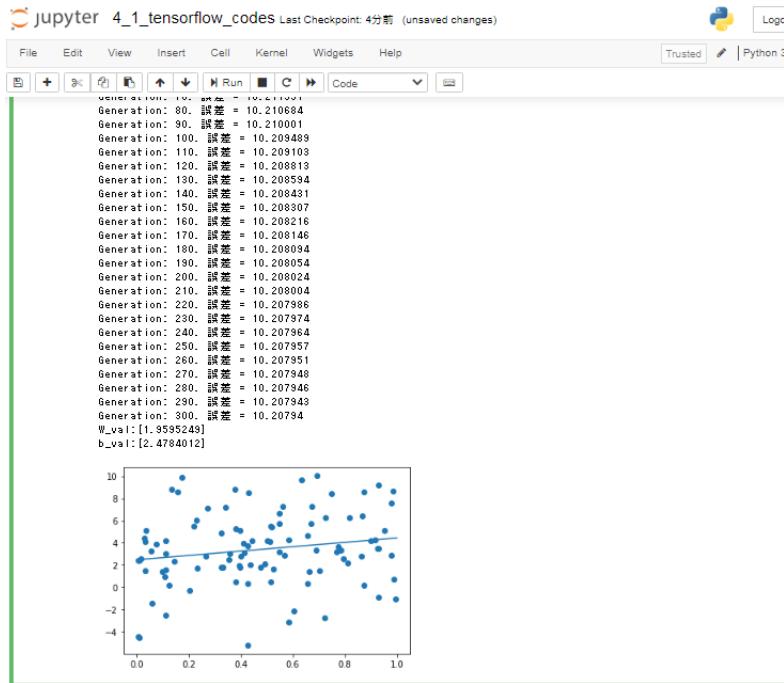
#####
# 最適化の対象の変数を初期化
w = tf.Variable(tf.zeros([1]))
b = tf.Variable(tf.zeros([1]))

y = w * xt + b

# 総差平方根誤差
loss = tf.reduce_mean(tf.square(y - dt))
optimizer = tf.train.GradientDescentOptimizer(0.1) #学習率
train = optimizer.minimize(loss)
#####

# 初期化
init = tf.global_variables_initializer()
sess = tf.Session()
sess.run(init)

```



ノイズが増えて、誤差が大きくなつたが、予測しようとはしている。

d の数値を変更する。 $d = 3 * x + 2 \Rightarrow d = -3 * x + 20$

The screenshot shows a Jupyter Notebook interface with the title "jupyter 4_1_tensorflow_codes". The notebook displays the following Python code in a cell:

```

In [14]: import numpy as np
import tensorflow as tf
import matplotlib.pyplot as plt
import tensorflow.compat.v1 as tf #add
tf.disable_v2_behavior() #add

iters_num = 300
plot_interval = 10

#####
# データ生成
n = 100
x = np.random.rand(n)
d = -3 * x + 20

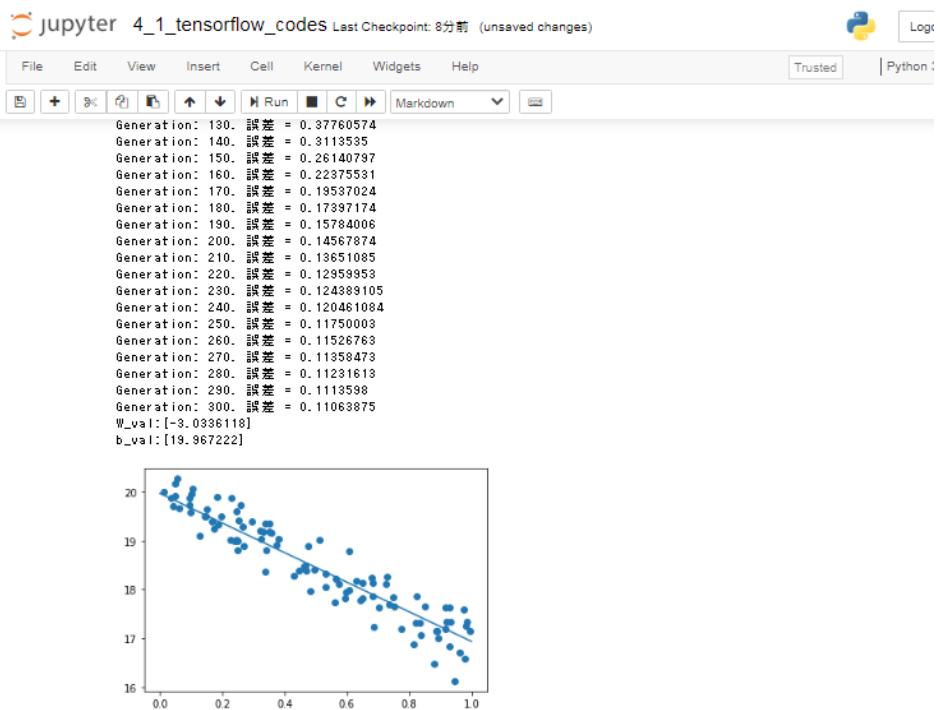
# ノイズを加える
noise = 0.3
d = d + noise * np.random.randn(n)
#####

# 入力値
xt = tf.placeholder(tf.float32)
dt = tf.placeholder(tf.float32)

#####
# 最適化のための変数を初期化
W = tf.Variable(tf.zeros([1])) #add
b = tf.Variable(tf.zeros([1])) #add

y = W * xt + b

# 誤差関数 平均2乗誤差
loss = tf.reduce_mean(tf.square(y - dt))
optimizer = tf.train.GradientDescentOptimizer(0.1) #学習率
train = optimizer.minimize(loss)
#####
```



線形予測できている。

3.1.3 Tensorflow、非線形回帰

```
In [16]: import numpy as np
import tensorflow as tf
import matplotlib.pyplot as plt
import tensorflow.compat.v1 as tf #add
tf.disable_v2_behavior() #add

iters_num = 10000
plot_interval = 100

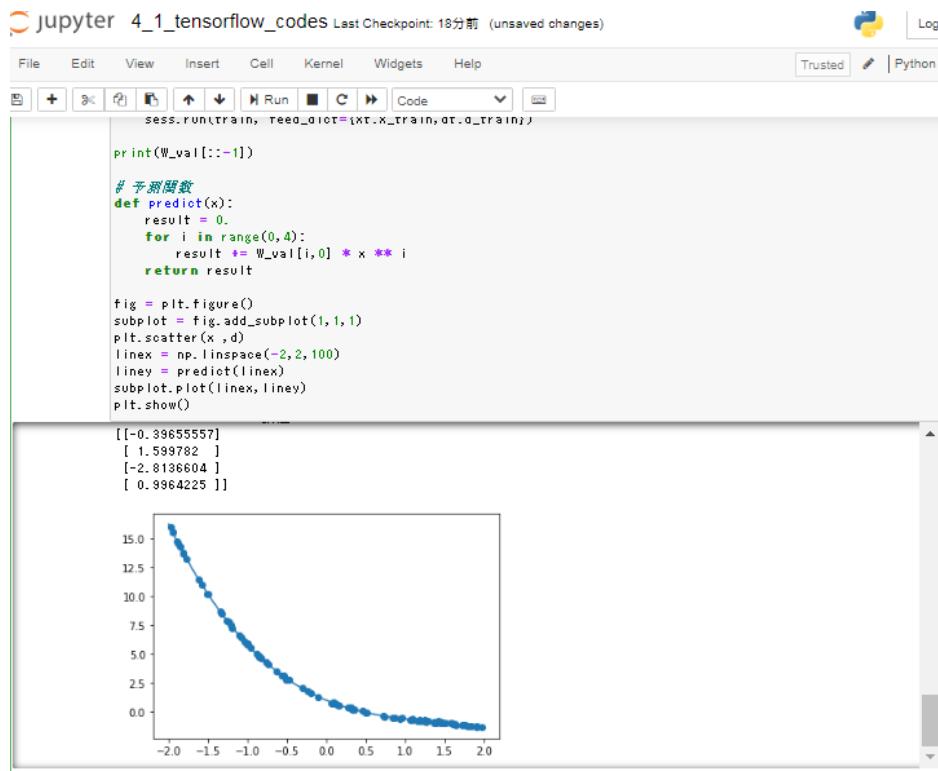
#####
# データを生成
n=100
x = np.random.rand(n).astype(np.float32) * 4 - 2
d = -0.4 * x ** 3 + 1.6 * x ** 2 - 2.8 * x + 1

# ノイズを加える
noise = 0.05
d = d + noise * np.random.randn(n)
#####

# モデル
# ここで使っていないことに注意。
xt = tf.placeholder(tf.float32, [None, 4])
dt = tf.placeholder(tf.float32, [None, 1])
W = tf.Variable(tf.random_normal([4, 1], stddev=0.01)) # ランダムな初期値
y = tf.matmul(xt, W)

# 誤差関数 平均の二乗誤差
loss = tf.reduce_mean(tf.square(y - dt))
optimizer = tf.train.AdamOptimizer(0.001)
train = optimizer.minimize(loss)
#####

# 初期化
init = tf.global_variables_initializer()
sess = tf.Session()
sess.run(init)
```



noise の値を変更 0.05 \Rightarrow 5 100 倍

```

In [17]: import numpy as np
import tensorflow as tf
import matplotlib.pyplot as plt
import tensorflow.compat.v1 as tf #add
tf.disable_v2_behavior() #add

iters_num = 10000
plot_interval = 100

#####
# データを生成
n=100
x = np.random.rand(n).astype(np.float32) * 4 - 2
d = - 0.4 * x ** 3 + 1.6 * x ** 2 - 2.8 * x + 1

# ノイズを加える
noise = 5
d = d + noise * np.random.randn(n)
#####

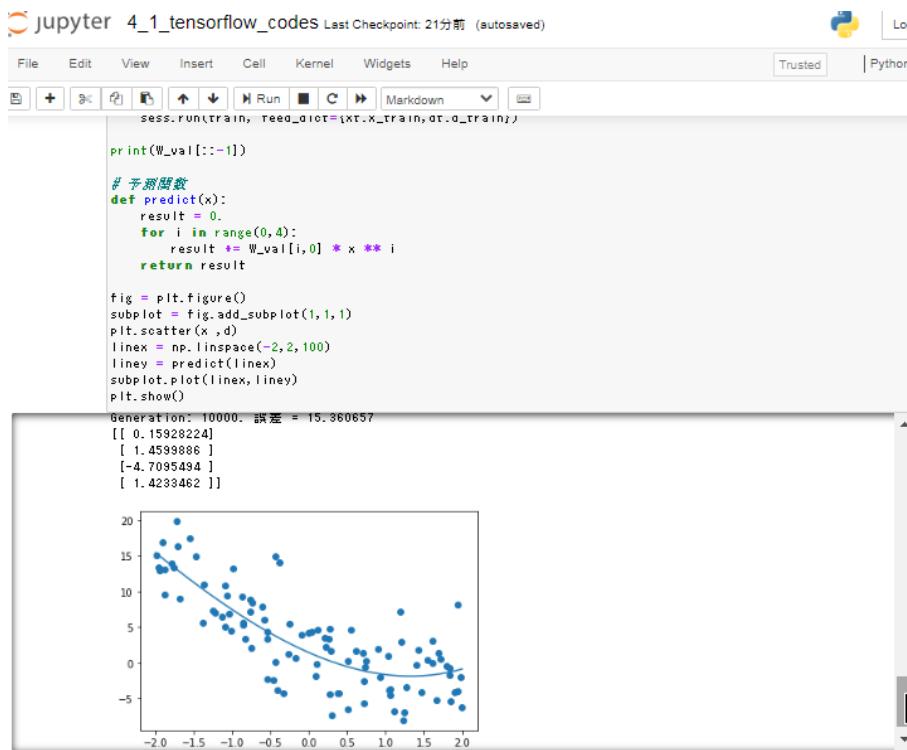
# モデル
# bを使っていないことに注意。
xt = tf.placeholder(tf.float32, [None, 4])
dt = tf.placeholder(tf.float32, [None, 1])
W = tf.Variable(tf.random_normal([4, 1], stddev=0.01)) # ランダムな初期値
y = tf.matmul(xt,W)

# 誤差関数 平均2乗誤差
loss = tf.reduce_mean(tf.square(y - dt))
optimizer = tf.train.AdamOptimizer(0.001)
train = optimizer.minimize(loss)
#####

# 初期化
init = tf.global_variables_initializer()
sess = tf.Session()
sess.run(init)

# 作成したデータをトレーニングデータとして準備
d_train = d.reshape(-1,1)

```



ノイズは大きいが、予測しようとしている。

d の数値を変更する。 $d = 3 * x + 2 \Rightarrow d = -3 * x + 20$

```

jupyter 4_1_tensorflow_codes Last Checkpoint: 24分前 (unsaved changes)
File Edit View Insert Cell Kernel Widgets Help
Trusted Python 3
Run Cell Code

```

```

In [18]: import numpy as np
import tensorflow as tf
import matplotlib.pyplot as plt
import tensorflow.compat.v1 as tf #add
tf.disable_v2_behavior() #add

iters_num = 10000
plot_interval = 100

#####
# データ生成
n=100
x = np.random.rand(n).astype(np.float32) * 4 - 2
d = -0.4 * x ** 3 + 1.6 * x ** 2 - 2.8 * x + 1
d = 4 * x ** 3 - 2 * x ** 2 + 7 * x + 100

# ノイズを加える
noise = 0.05
d = d + noise * np.random.randn(n)
#####

# モデル
# aを使っていないことに注意
xt = tf.placeholder(tf.float32, [None, 4])
dt = tf.placeholder(tf.float32, [None, 1])
W = tf.Variable(tf.random_normal([4, 1], stddev=0.01)) # ランダムな初期値
y = tf.matmul(xt,W)

# 誤差関数 平均2乗誤差
loss = tf.reduce_mean(tf.square(y - dt))
optimizer = tf.train.AdamOptimizer(0.001)
train = optimizer.minimize(loss)
#####

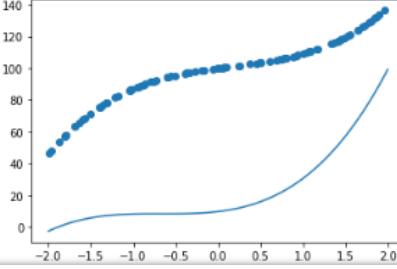
# 初期化
init = tf.global_variables_initializer()
sess = tf.Session()
sess.run(init)

```

jupyter 4_1_tensorflow_codes Last Checkpoint: 25分前 (unsaved changes)

File Edit View Insert Cell Kernel Widgets Help Trusted Python

```
print(W_val[::1])  
  
# 予測関数  
def predict(x):  
    result = 0.  
    for i in range(0,4):  
        result += W_val[i,0] * x ** i  
    return result  
  
fig = plt.figure()  
subplot = fig.add_subplot(1,1,1)  
plt.scatter(x,d)  
linex = np.linspace(-2,2,100)  
liney = predict(linex)  
subplot.plot(linex,liney)  
plt.show()  
Generation: 10000. 誤差 = 5753.1  
[[4.753125]  
 [9.631657]  
 [6.4105406]  
 [9.759061]]
```



誤差が大きく、10000回でも学習を終えていない…。初期値から離れすぎたか。

実装演習 データ生成のモデルの式の変更。

jupyter 4_1_tensorflow_codes Last Checkpoint: 9分前 (unsaved changes)

File Edit View Insert Cell Kernel Widgets Help Trusted Python 3

次の式をモデルとして回帰を行おう
 $y = 30x^2 + 0.5x + 0.2$

誤差が収束するようiters_numやlearning_rateを調整しよう

```
In [24]: import numpy as np
import tensorflow as tf
import matplotlib.pyplot as plt
import tensorflow.compat.v1 as tf #add
tf.disable_v2_behavior() #add

iters_num = 10000
plot_interval = 100

#####
# データを生成
n=100
x = np.random.rand(n).astype(np.float32) * 4 - 2
d = 30 * x ** 2 + 0.5 * x + 0.2
## モデル: y = 30x^2 + 0.5x + 0.2

# ノイズを加える
noise = 0.05
d = d + noise * np.random.randn(n)
#####

# モデル
# もぎ使っていないことに注意。
xt = tf.placeholder(tf.float32, [None, 3])
dt = tf.placeholder(tf.float32, [None, 1])
W = tf.Variable(tf.random_normal([3, 1], stddev=0.01)) # ランダムな初期値
y = tf.matmul(xt,W)

# 誤差関数 平均2乗誤差
loss = tf.reduce_mean(tf.square(y - dt))
optimizer = tf.train.AdamOptimizer(0.001) # learning_rate
train = optimizer.minimize(loss)
```

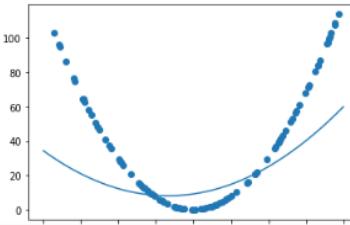
jupyter 4_1_tensorflow_codes Last Checkpoint: 9分前 (unsaved changes)

File Edit View Insert Cell Kernel Widgets Help Trusted Python 3

```
# プリント関数
def predict(x):
    result = 0.
    for i in range(0,3):
        result += W_val[i,0] * x ** i
    return result

fig = plt.figure()
subplot = fig.add_subplot(1,1,1)
plt.scatter(x ,d)
linex = np.linspace(-2,2,100)
liney = predict(linex)
subplot.plot(linex,liney)
plt.show()
```

Generation: 9900. 誤差 = 786.3793
Generation: 10000. 誤差 = 755.5774
[[9.455617]
[6.3872647]
[9.283901]]



誤差が大きいので、学習率を 0.001 \Rightarrow 0.01 に変更。

jupyter 4_1_tensorflow_codes Last Checkpoint: 10分前 (unsaved changes)

File Edit View Insert Cell Kernel Widgets Help Trusted Python 3

```
In [24]: import numpy as np
import tensorflow as tf
import matplotlib.pyplot as plt
import tensorflow.compat.v1 as tf #add
tf.disable_v2_behavior() #add

iters_num = 10000
plot_interval = 100

#####
# データを生成
n=100
x = np.random.rand(n).astype(np.float32) * 4 - 2
d = 30 * x ** 2 + 0.5 * x + 0.2
## モデル: y = 30 x^2 + 0.5 x + 0.2

# ノイズを加える
noise = 0.05
d = d + noise * np.random.randn(n)
#####

# モデル
# カラムでないことに注意。
xt = tf.placeholder(tf.float32, [None, 3])
dt = tf.placeholder(tf.float32, [None, 1])
W = tf.Variable(tf.random_normal([3, 1], stddev=0.01)) # ランダムな初期値
y = tf.matmul(xt,W)

# 誤差関数 平均2乗誤差
loss = tf.reduce_mean(tf.square(y - dt))
optimizer = tf.train.AdamOptimizer(0.01) # learning_rate
train = optimizer.minimize(loss)
#####

# 初期化
init = tf.global_variables_initializer()
sess = tf.Session()
sess.run(init)
```

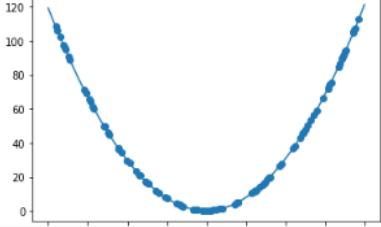
jupyter 4_1_tensorflow_codes Last Checkpoint: 10分前 (unsaved changes)

File Edit View Insert Cell Kernel Widgets Help Trusted Python 3

```
# 予測関数
def predict(x):
    result = 0.
    for i in range(0,3):
        result += W_val[i,0] * x ** i
    return result

fig = plt.figure()
subplot = fig.add_subplot(1,1,1)
plt.scatter(x,d)
linex = np.linspace(-2,2,100)
liney = predict(linex)
subplot.plot(linex,liney)
plt.show()
```

Generation: 8900. 誤差 = 0.0025890225
Generation: 10000. 誤差 = 0.0025890537
[[29.999922]
[0.50231844]
[0.20458564]]



誤差の少ない予測が行えた。

3.1.4 Tensorflow、MNIST (分類 1 層)

jupyter 4_1_tensorflow_codes Last Checkpoint: 10分前 (autosaved) Trusted Python 3

```
In [1]: import tensorflow as tf
import matplotlib.pyplot as plt
#tf.disable_v2_behavior() #add
from tensorflow_core.examples.tutorials.mnist import input_data
mnist = input_data.read_data_sets("MNIST_data/", one_hot=True)
#(x_train, y_train), (x_test, y_test) = tf.keras.datasets.mnist.load_data()

iters_num = 100
batch_size = 100
plot_interval = 1

# _____ ここを補填 _____
x = tf.placeholder(tf.float32, [None, 784])
d = tf.placeholder(tf.float32, [None, 10])
W = tf.Variable(tf.random_normal([784, 10], stddev=0.01))
b = tf.Variable(tf.zeros([10]))
#
y = tf.nn.softmax(tf.matmul(x, W) + b)

# 文字エンコード
cross_entropy = -tf.reduce_sum(d * tf.log(y), reduction_indices=[1])
loss = tf.reduce_mean(cross_entropy)
train = tf.train.GradientDescentOptimizer(0.1).minimize(loss)

# 正解を保存
correct = tf.equal(tf.argmax(y, 1), tf.argmax(d, 1))
# 正解率
accuracy = tf.reduce_mean(tf.cast(correct, tf.float32))

init = tf.global_variables_initializer()
sess = tf.Session()
sess.run(init)

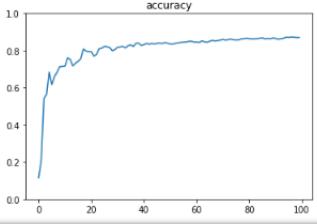
accuracies = []
for i in range(iters_num):
```

jupyter 4_1_tensorflow_codes Last Checkpoint: 10分前 (autosaved) Trusted Python 3

```
    sess.run(train, feed_dict={x: x_batch, d: d_batch})
    if (i+1) % plot_interval == 0:
        print(sess.run(correct, feed_dict={x: mnist.test.images, d: mnist.test.labels}))
        accuracy_val = sess.run(accuracy, feed_dict={x: mnist.test.images, d: mnist.test.labels})
        accuracies.append(accuracy_val)
        print('Generation: ' + str(i+1) + ', 正解率 = ' + str(accuracy_val))

    lists = range(0, iters_num, plot_interval)
    plt.plot(lists, accuracies)
    plt.title("accuracy")
    plt.ylim(0, 1.0)
    plt.show()
```

Generation: 99, 正解率 = 0.8703
[True True True... True False True]
Generation: 100, 正解率 = 0.8705



1 層で 87%

3.1.5 Tensorflow、MNIST (分類 3 層)

jupyter 4_1_tensorflow_codes Last Checkpoint: 20分前 (autosaved)

File Edit View Insert Cell Kernel Widgets Help Trusted Python 3

```
In [2]: import tensorflow as tf
import numpy as np
from tensorflow.examples.tutorials.mnist import input_data
import matplotlib.pyplot as plt

mnist = input_data.read_data_sets("MNIST_data/", one_hot=True)

iters_num = 3000
batch_size = 100
plot_interval = 100

hidden_layer_size_1 = 600
hidden_layer_size_2 = 300

dropout_rate = 0.5

x = tf.placeholder(tf.float32, [None, 784])
d = tf.placeholder(tf.float32, [None, 10])
W1 = tf.Variable(tf.random_normal([784, hidden_layer_size_1], stddev=0.01))
W2 = tf.Variable(tf.random_normal([hidden_layer_size_1, hidden_layer_size_2], stddev=0.01))
W3 = tf.Variable(tf.random_normal([hidden_layer_size_2, 10], stddev=0.01))

b1 = tf.Variable(tf.zeros([hidden_layer_size_1]))
b2 = tf.Variable(tf.zeros([hidden_layer_size_2]))
b3 = tf.Variable(tf.zeros([10]))

z1 = tf.sigmoid(tf.matmul(x, W1) + b1)
z2 = tf.sigmoid(tf.matmul(z1, W2) + b2)

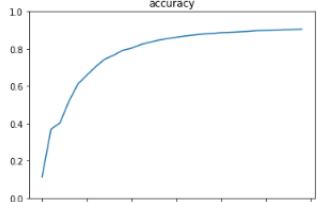
keep_prob = tf.placeholder(tf.float32)
drop = tf.nn.dropout(z2, keep_prob)

y = tf.nn.softmax(tf.matmul(drop, W3) + b3)
loss = tf.reduce_mean(tf.reduce_sum(d * tf.log(y), reduction_indices=[1]))
```

jupyter 4_1_tensorflow_codes Last Checkpoint: 20分前 (autosaved)

File Edit View Insert Cell Kernel Widgets Help Trusted Python 3

```
Generation: 1200. 正解率 = 0.8223
Generation: 1300. 正解率 = 0.835
Generation: 1400. 正解率 = 0.8464
Generation: 1500. 正解率 = 0.8545
Generation: 1600. 正解率 = 0.8622
Generation: 1700. 正解率 = 0.8688
Generation: 1800. 正解率 = 0.8742
Generation: 1900. 正解率 = 0.8796
Generation: 2000. 正解率 = 0.882
Generation: 2100. 正解率 = 0.8867
Generation: 2200. 正解率 = 0.8883
Generation: 2300. 正解率 = 0.8909
Generation: 2400. 正解率 = 0.8933
Generation: 2500. 正解率 = 0.8975
Generation: 2600. 正解率 = 0.8983
Generation: 2700. 正解率 = 0.8992
Generation: 2800. 正解率 = 0.9023
Generation: 2900. 正解率 = 0.9029
Generation: 3000. 正解率 = 0.9053
```

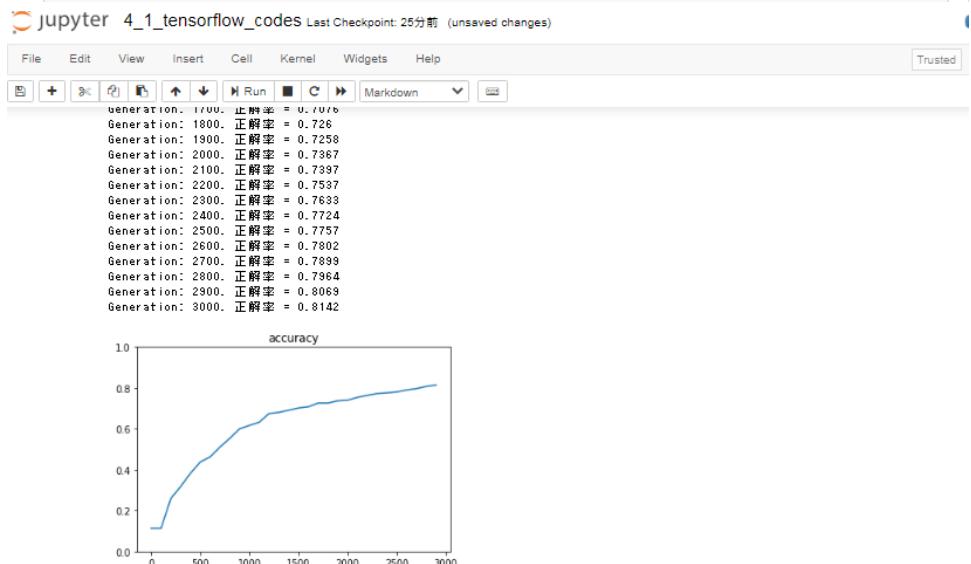


3 層で 90%

隠れ層のサイズを変更 600,300 ⇒ 300,100

jupyter 4_1_tensorflow_codes Last Checkpoint: 25分前 (unsaved changes)

```
In [3]:  
import tensorflow as tf  
import numpy as np  
from tensorflow.examples.tutorials.mnist import input_data  
import matplotlib.pyplot as plt  
  
mnist = input_data.read_data_sets("MNIST_data/", one_hot=True)  
  
iters_num = 3000  
batch_size = 100  
plot_interval = 100  
  
hidden_layer_size_1 = 300  
hidden_layer_size_2 = 100  
  
dropout_rate = 0.5  
  
x = tf.placeholder(tf.float32, [None, 784])  
d = tf.placeholder(tf.float32, [None, 10])  
W1 = tf.Variable(tf.random_normal([784, hidden_layer_size_1], stddev=0.01))  
W2 = tf.Variable(tf.random_normal([hidden_layer_size_1, hidden_layer_size_2], stddev=0.01))  
W3 = tf.Variable(tf.random_normal([hidden_layer_size_2, 10], stddev=0.01))  
  
b1 = tf.Variable(tf.zeros([hidden_layer_size_1]))  
b2 = tf.Variable(tf.zeros([hidden_layer_size_2]))  
b3 = tf.Variable(tf.zeros([10]))  
  
z1 = tf.sigmoid(tf.matmul(x, W1) + b1)  
z2 = tf.sigmoid(tf.matmul(z1, W2) + b2)  
  
keep_prob = tf.placeholder(tf.float32)  
drop = tf.nn.dropout(z2, keep_prob)  
  
y = tf.nn.softmax(tf.matmul(drop, W3) + b3)  
loss = tf.reduce_mean(-tf.reduce_sum(d * tf.log(y), reduction_indices=[1]))  
  
optimizer = tf.train.AdamOptimizer(1e-4)
```



81%まで落ちる

オプティマイザの変更

jupyter 4_1_tensorflow_codes Last Checkpoint: 29分前 (unsaved changes)

File Edit View Insert Cell Kernel Widgets Help Trusted Python 3

```
import numpy as np
from tensorflow.examples.tutorials.mnist import input_data
import matplotlib.pyplot as plt

mnist = input_data.read_data_sets("MNIST_data/", one_hot=True)

iters_num = 3000
batch_size = 100
plot_interval = 100

hidden_layer_size_1 = 600
hidden_layer_size_2 = 300

dropout_rate = 0.5

x = tf.placeholder(tf.float32, [None, 784])
d = tf.placeholder(tf.float32, [None, 10])
W1 = tf.Variable(tf.random_normal([784, hidden_layer_size_1], stddev=0.01))
W2 = tf.Variable(tf.random_normal([hidden_layer_size_1, hidden_layer_size_2], stddev=0.01))
W3 = tf.Variable(tf.random_normal([hidden_layer_size_2, 10], stddev=0.01))

b1 = tf.Variable(tf.zeros([hidden_layer_size_1]))
b2 = tf.Variable(tf.zeros([hidden_layer_size_2]))
b3 = tf.Variable(tf.zeros([10]))

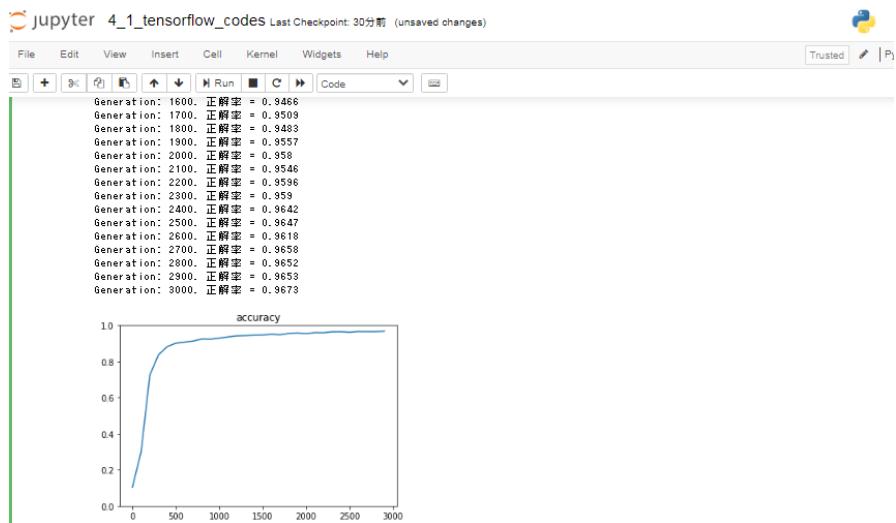
z1 = tf.sigmoid(tf.matmul(x, W1) + b1)
z2 = tf.sigmoid(tf.matmul(z1, W2) + b2)

keep_prob = tf.placeholder(tf.float32)
drop = tf.nn.dropout(z2, keep_prob)

y = tf.nn.softmax(tf.matmul(drop, W3) + b3)
loss = tf.reduce_mean(-tf.reduce_sum(d * tf.log(y), reduction_indices=[1]))

#optimizer = tf.train.AdamOptimizer(1e-4)
optimizer = tf.train.RMSPropOptimizer(0.001)

train = optimizer.minimize(loss)
correct = tf.equal(tf.argmax(y, 1), tf.argmax(d, 1)) ..
```



96.7%になる。

3.1.6 Tensorflow、MNIST(分類 CNN)

jupyter 4_1_tensorflow_codes Last Checkpoint: 1時間前 (autosaved)

File Edit View Insert Cell Kernel Widgets Help Trusted Python 3

```
In [5]: import tensorflow as tf
from tensorflow.examples.tutorials.mnist import input_data
mnist = input_data.read_data_sets("MNIST_data/", one_hot=True)
import matplotlib.pyplot as plt

iters_num = 300
batch_size = 100
plot_interval = 10

dropout_rate = 0.5

# placeholder
x = tf.placeholder(tf.float32, shape=[None, 784])
d = tf.placeholder(tf.float32, shape=[None, 10])

# 画像を784の一次元から28x28の二次元に変換する
# 画像を28x28にreshape
x_image = tf.reshape(x, [-1, 28, 28, 1])

# 第一層のweightsとbiasのvariable
w_conv1 = tf.Variable(tf.truncated_normal([5, 5, 1, 32], stddev=0.1))
b_conv1 = tf.Variable(tf.constant(0.1, shape=[32]))

# 第一層のconvolutionとpool
# strides[0] = strides[3] = 1固定
h_conv1 = tf.nn.relu(tf.nn.conv2d(x_image, w_conv1, strides=[1, 1, 1, 1], padding='SAME') + b_conv1)
# プーリングサイズ n*n にしたい場合 ksizes=[1, n, n, 1]
h_pool1 = tf.nn.max_pool(h_conv1, ksize=[1, 2, 2, 1], strides=[1, 2, 2, 1], padding='SAME')

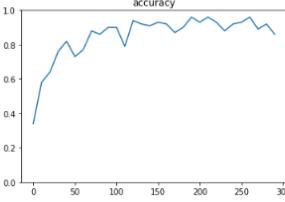
# 第二層
w_conv2 = tf.Variable(tf.truncated_normal([5, 5, 32, 64], stddev=0.1))
b_conv2 = tf.Variable(tf.constant(0.1, shape=[64]))
h_conv2 = tf.nn.relu(tf.nn.conv2d(h_pool1, w_conv2, strides=[1, 1, 1, 1], padding='SAME') + b_conv2)
h_pool2 = tf.nn.max_pool(h_conv2, ksize=[1, 2, 2, 1], strides=[1, 2, 2, 1], padding='SAME')

# 第一層と第二層でreduceされてできた特徴に対してrelu
w_fc1 = tf.Variable(tf.truncated_normal([7 * 7 * 64, 1024], stddev=0.1))
b_fc1 = tf.Variable(tf.constant(0.1, shape=[1024]))
```

jupyter 4_1_tensorflow_codes Last Checkpoint: 1時間前 (autosaved)

File Edit View Insert Cell Kernel Widgets Help Trusted Python 3

```
Generation: 160. 正解率: 0.93
Generation: 170. 正解率: 0.92
Generation: 180. 正解率: 0.87
Generation: 190. 正解率: 0.9
Generation: 200. 正解率: 0.96
Generation: 210. 正解率: 0.98
Generation: 220. 正解率: 0.98
Generation: 230. 正解率: 0.98
Generation: 240. 正解率: 0.88
Generation: 250. 正解率: 0.92
Generation: 260. 正解率: 0.93
Generation: 270. 正解率: 0.96
Generation: 280. 正解率: 0.89
Generation: 290. 正解率: 0.92
Generation: 300. 正解率: 0.86
```



A line graph titled 'accuracy' showing the accuracy of the model over 300 generations. The x-axis represents generations from 0 to 300, and the y-axis represents accuracy from 0.0 to 1.0. The accuracy starts at approximately 0.35 and rises steadily, fluctuating slightly, to reach about 0.86 by generation 300.

最高で 96% ぐらい。

ドロップアウト率をゼロに変更

jupyter 4_1_tensorflow_codes Last Checkpoint: 1時間前 (unsaved changes)

File Edit View Insert Cell Kernel Widgets Help Python 3

分類CNN (mnist)

```
conv - relu - pool - conv - relu - pool -  
affin - relu - dropout - affin - softmax
```

[try]

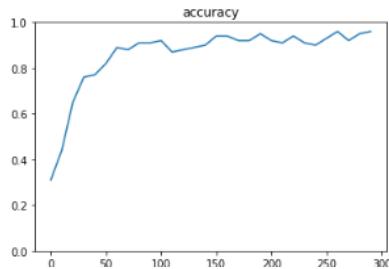
- ドロップアウト率を0に変更しよう

```
In [8]:  
import tensorflow as tf  
from tensorflow.examples.tutorials.mnist import input_data  
mnist = input_data.read_data_sets("MNIST_data/", one_hot=True)  
import matplotlib.pyplot as plt  
  
iters_num = 300  
batch_size = 100  
plot_interval = 10  
  
#dropout_rate = 0.5  
dropout_rate = 0  
  
# placeholder  
x = tf.placeholder(tf.float32, shape=[None, 784])  
d = tf.placeholder(tf.float32, shape=[None, 10])  
  
# 画像を784の一次元から28x28の二次元に変換する  
# 画像を28x28にreshape  
x_image = tf.reshape(x, [-1, 28, 28, 1])  
  
# 第一層のweightsとbiasのvariable  
W_conv1 = tf.Variable(tf.truncated_normal([5, 5, 1, 32], stddev=0.1))  
b_conv1 = tf.Variable(tf.constant(0.1, shape=[32]))
```

jupyter 4_1_tensorflow_codes Last Checkpoint: 1時間前 (unsaved changes)

File Edit View Insert Cell Kernel Widgets Help Python 3

```
Generation: 180. 正解率 = 0.92  
Generation: 180. 正解率 = 0.92  
Generation: 200. 正解率 = 0.95  
Generation: 210. 正解率 = 0.92  
Generation: 220. 正解率 = 0.91  
Generation: 230. 正解率 = 0.94  
Generation: 240. 正解率 = 0.91  
Generation: 250. 正解率 = 0.9  
Generation: 260. 正解率 = 0.93  
Generation: 270. 正解率 = 0.96  
Generation: 280. 正解率 = 0.92  
Generation: 290. 正解率 = 0.95  
Generation: 300. 正解率 = 0.96
```



The graph shows the accuracy of the model during training. The x-axis represents the generation number from 0 to 300, and the y-axis represents the accuracy from 0.00 to 1.00. The accuracy starts at approximately 0.35 and increases rapidly initially, reaching about 0.85 by generation 50. It then continues to rise more slowly, fluctuating slightly between 0.85 and 0.95, eventually reaching a final accuracy of about 0.96 at generation 300.

ドロップアウト率をゼロに変更してもあまり変わらない、

トレーニングデータでの正解率であるため。

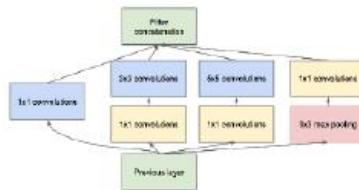
本来はドロップアウトがある方が汎化性能があがり精度はあがる。

3.1.7 例題解説 1

問：以下の問いに記号で答えよ。

画像認識に用いられるCNNのアーキテクチャにはいくつか有名なものがある。その中の一つとしてGoogleNetと呼ばれるアーキテクチャがある。GoogLeNetの特徴としてInception moduleと呼ばれる複数のフィルタ群により構成されたブロックがあげられる。以下の中でinception moduleに関する記述として誤っているものを一つ選べ。

- (a) Inception moduleは通常のK*Kの畳み込みフィルタと比較すると非常にパラメータが増えているとみなせるため、相対的にdenseな演算であると言える。
- (b) Inception moduleは大きな畳み込みフィルタを小さな畳み込みフィルタのグループで近似することで、モデルの表現力とパラメータ数のトレードオフを改善していると言える。
- (c) Inception moduleには1*1の畳み込みフィルタが使われているが、このフィルターは次元削減と等価な効果がある。
- (d) Inception moduleは小さなネットワークを1つのモジュールとして定義し、モジュールの積み重ねでネットワークを構築する。



2018年07月24 JDIA提供

1. googLe.net は、Inception モジュールの積み重ね ((d)は正しい)
 2. Inception モジュール
 - ①黄色部分を除くと、同じ出力データサイズで、フィルタ 1x1、フィルタ 3x3
複数のフィルタサイズを利用して表現力があがった ((b)は正しい)
 - ②1x1 畳み込みによる次元削減 ((c)は正しい)
- (a)が答え : dense な演算が必要なパラメータ数は増えるが、その分スペースな演算となる。

3.1.8 例題解説 2

またもう一つのGoogLeNetの特徴としてAuxiliary Lossの存在が挙げられる。以下の中でAuxiliary Lossに関する記述のとして間違っているものを一つ選べ。

- (a) AuxiliaryLossを導入することで多層なネットワークでも計算量を抑える効果が期待できる。
- (b) GoogLeNetの学習ではネットワークの途中から分岐させたサブネットワークにおいてもクラス分類を行っている。
- (c) AuxiliaryLossを導入しない場合でもBatchNormalizationを加えることにより、同様に学習がうまく進むことがある。
- (d) AuxiliaryLossによってアンサンブル学習と同様の効果が得られるため、汎化性能の向上が期待できる。



2018年07月24 JDLa提供

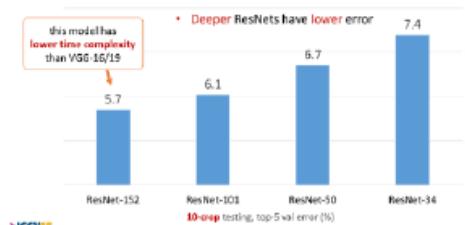
途中から分岐させたサブネットワークにおいてもクラス分類を行っている。((b)は正しい)
BatchNormalization は勾配消失の対策の 1 つ、Auxiliary classifiers を追加しなくても
勾配消失問題を改善できる ((c)は正しい)
アンサンブル学習では複数の学習器を用意し予測結果を統合し、汎化性能を高める ((d)は
正しい)
1x1 の Convolution を使って GoogLeNet では次元削減を行っているので、計算量を抑えて
いる。((a)は誤りであり答えとなる)

3.1.9 例題解説 3

問：以下の空欄に入る選択肢を記号で答えよ。

Residual Network (ResNet)は2015年のImageNetコンペティションとCOCOセグメンテーション内の主要なコンペティション5つ全てにおいて2位以下を大きく引き離した最良モデルであり、深層学習に飛躍的な進展をもたらした。

ImageNet の分類タスクコンペティションにおいて、ResNet が登場する以前からニューラルネットワークは他手法を圧倒する性能を出していた。ResNet がそれ以前のモデルと顕著に異なる点は、層の深さである。2012年に登場した AlexNet は 8 層、2014年に登場した VGG、GoogleNet は各々 19、22 層の深さを持つが、ResNet は 152 層の深さを持つ。



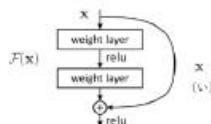
2018年07月24 JDIA提供

一般に、深層学習においてモデルを深く設計することはモデルの表現力を効率的に向上できると知られている。しかし、ResNet の登場以前は（a）問題によって、深いモデルを学習することができなかった。
単純に層を積んだ深いモデルは、より浅いモデルと比べて、訓練誤差が大きくなる傾向がある。
ResNet はこの問題に取り組み、より深いモデルがより高い性能を発揮するよう改善を行った。

- (a) 勾配消失 (b) 非収束 (c) 過剰適合 (d) ノーフリーランチ

ResNet では層をまたがる結合として（い）mapping を用いる。そうすることでスキップコネクションの内側の層は ブロックの入出力の残差を学習するということになる。

- (a) Identity (b) Residual (c) Recurrent (d) Convolution



2018年07月24 JDIA提供

Resnet より前は、勾配消失問題により層を深くすることが困難だった。（a）が答え
Identity mapping、スキップコネクションにより、上手く残差を学習することができる。

Residual Block の導入によって (う) ことが期待される。この手法により、従来の層を深くすると学習できないという問題が解決された。

- (a) ブロックへの入力にこれ以上の変換が必要ない場合は重みが 0 となり、小さな変換が求められる場合は対応する小さな変動をより見つけやすくなる
- (b) ブロックの出力ごとに教師データを用意して誤差逆伝播を行うため、勾配消失問題を低減する
- (c) ブロックへの入出力の差分が小さくなるように学習が行われ、勾配消失問題を低減する
- (d) モデルパラメータを大幅に減らすことができ、効率的な学習ができるようになる

2018年07月24 JDIA提供

ResNet のワイドという Block のスキップコネクションを搭載したのが、ResNet Block

(a) が期待された。このブロックへの入力、これ以上の変換が無い時は、重みをゼロにする。スキップコネクションの weight_layer を 0,0、その前の出力をそのまま渡すだけ。

3.1.10 例題解説 4

問：次の選択肢の中から誤ったものを1つ選べ。

- (a) 転移学習の極端な形式として、ワンショット学習がある。ワンショット学習は、転移前のタスクで表現学習を行い、その表現を利用することで転移先のタスクを実行する。このとき、転移先のタスクにおいてはラベル付き事例が与えられない。
- (b) 転移学習やドメイン適応とは、あるタスクで学習したことを別のタスクにおける汎化性能向上に役立てることである。
- (c) 機械学習システムがうまく動作しない場合、それがアルゴリズム自体の問題か、実装にバグがあるのかを判断することは難しい。デバッグの際には定量的な評価値を見るだけでなく、モデルから出力される画像・音声の可視化・可聴化が大切である。
- (d) 表現学習における理想的な表現の仮説の一つに、その表現の中の特徴量が観測データの潜在的原因に対応していることが挙げられる。人の顔画像の表現学習というタスクに対し、最小二乗誤差基準の自己符号化器を用いると耳の生成が疎かになる。顔を表す耳という潜在的原因を捉えられていないと考えられる。このような問題に対する1つのアプローチは敵対的生成ネットワークを用いることである。

2018年07月24 JDLa提供

転移学習の問題。(a)が誤り。他は正しい。

ワンショット学習は、1枚の画像から特徴ベクトルを抽出、同じクラスならば近い、違うならば離れているという学習方法。転移学習の例としては誤り。

3.1.11 例題解説 5

問：一般物体検出アルゴリズムのその他のアルゴリズムについて以下に述べた4つの記述内から正しいものを一つ選べ。

- (a) YOLOと呼ばれるネットワークアーキテクチャは検出速度は高速だが、画像内のオブジェクト同士が複数重なっている場合に上手く検出できないという欠点がある。
- (b) YOLOでは画像をある固定長で分割したセル毎に最大2つの候補領域が推定される。
- (c) Single Shot MultiBox Detectorと呼ばれるネットワークモデルの誤差はオブジェクト領域の位置特定誤差と各クラスの確信度に対する確信度誤差の2つの誤差の単純和で表現される。
- (d) YOLOでは画像をある固定長で分解したセル毎にそれぞれどのカテゴリの物体なのかもしくはただの背景なのか、のそれぞれの確率が出力される。

2018年07月24 JDLa提供

- (a) は正しい。(b)は最大でなく、2つの候補領域は生成される。
(c) 2つの誤算の単純ではなく、位置特定誤差と各クラスの確信度に対する2つの誤差の重みづけまで表現されている。(d) は背景の確率ではない。

VGG	古く 2014 年のモデル。Convolution, Convolution, max_pool という単純な層の積み重ねなので、ネットワークの構成がシンプル。パラメータ数は他のものより多い。
GoogLeNet	Inception module を使っている。 1×1 の畳み込みを使った次元削減や、さまざまなフィルターサイズを使うスペースなものである。
ResNet	スキップコネクションアイデンティモジュールを使うことで残差接続を行い、深い学習が行える。

3.1.12 Keras、線形回帰

jupyter 4_3_keras_codes Last Checkpoint: 2018/11/24 (autosaved)

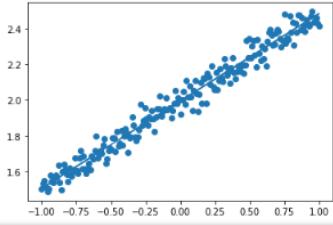
File Edit View Insert Cell Kernel Widgets Help Trusted Python 3

keras

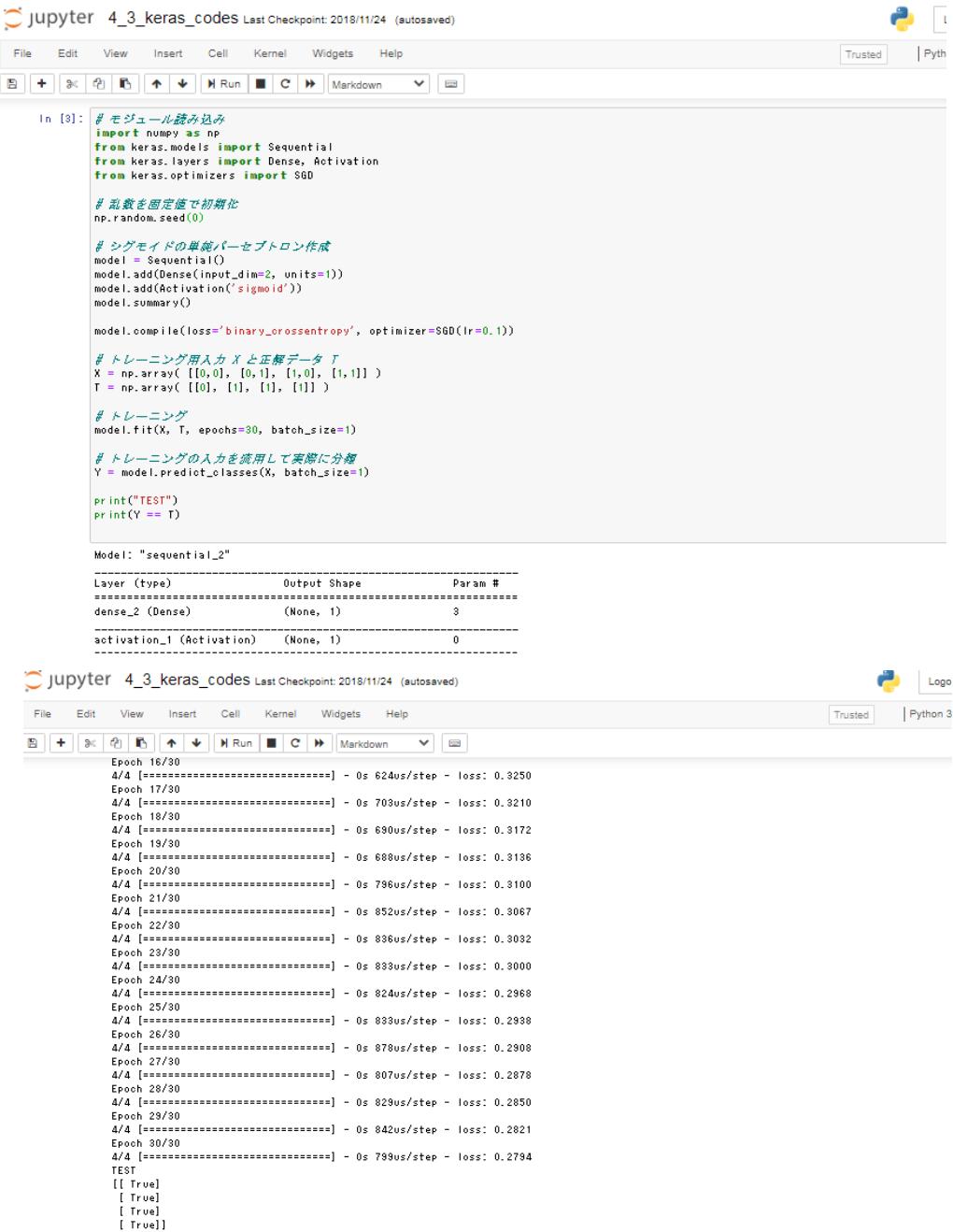
線形回帰

```
In [2]:  
import numpy as np  
import matplotlib.pyplot as plt  
  
iters_num = 1000  
plot_interval = 10  
  
x = np.linspace(-1, 1, 200)  
np.random.shuffle(x)  
d = 0.5 * x + 2 + np.random.normal(0, 0.05, (200,))  
  
from keras.models import Sequential  
from keras.layers import Dense  
  
# モデルを作成  
model = Sequential()  
model.add(Dense(input_dim=1, output_dim=1))  
  
# モデルを表示  
model.summary()  
  
# モデルのコンパイル  
model.compile(loss='mse', optimizer='sgd')  
  
# train  
for i in range(iters_num):  
    loss = model.train_on_batch(x, d)  
    if (i+1) % plot_interval == 0:  
        print('Generation: ' + str(i+1) + ', 誤差 = ' + str(loss))  
  
    W, b = model.layers[0].get_weights()  
    print('W:', W)  
  
loss = model.train_on_batch(x, d)  
if (i+1) % plot_interval == 0:  
    print('Generation: ' + str(i+1) + ', 誤差 = ' + str(loss))  
  
W, b = model.layers[0].get_weights()  
print('W:', W)  
print('b:', b)  
  
y = model.predict(x)  
plt.scatter(x, d)  
plt.plot(x, y)  
plt.show()
```

Generation: 990. 誤差 = 0.002870619
Generation: 1000. 誤差 = 0.0023705917
W: [[0.48915526]]
b: [1.3947736]



3.1.13 Keras、単純パーセプトロン（分類）



The screenshot shows two Jupyter Notebook sessions. The top session (In [3]) contains the following Python code:

```
# モジュール読み込み
import numpy as np
from keras.models import Sequential
from keras.layers import Dense, Activation
from keras.optimizers import SGD

# 亂数を固定値で初期化
np.random.seed(0)

# シグモイドの単純パーセプトロン作成
model = Sequential()
model.add(Dense(input_dim=2, units=1))
model.add(Activation('sigmoid'))
model.summary()

model.compile(loss='binary_crossentropy', optimizer=SGD(lr=0.1))

# トレーニング用入力Xと正解データT
X = np.array([ [0,0], [0,1], [1,0], [1,1] ])
T = np.array([ [0], [1], [1], [1] ])

# トレーニング
model.fit(X, T, epochs=30, batch_size=1)

# トレーニングの入力を適用して実際に分類
Y = model.predict_classes(X, batch_size=1)

print("TEST")
print(Y == T)

Model: "sequential_2"
-----
```

Layer (type)	Output Shape	Param #
dense_2 (Dense)	(None, 1)	3
activation_1 (Activation)	(None, 1)	0

The bottom session (In [4]) shows the training progress:

```
Epoch 16/30
4/4 [=====] - 0s 624us/step - loss: 0.3250
Epoch 17/30
4/4 [=====] - 0s 703us/step - loss: 0.3210
Epoch 18/30
4/4 [=====] - 0s 690us/step - loss: 0.3172
Epoch 19/30
4/4 [=====] - 0s 688us/step - loss: 0.3136
Epoch 20/30
4/4 [=====] - 0s 796us/step - loss: 0.3100
Epoch 21/30
4/4 [=====] - 0s 852us/step - loss: 0.3067
Epoch 22/30
4/4 [=====] - 0s 836us/step - loss: 0.3032
Epoch 23/30
4/4 [=====] - 0s 833us/step - loss: 0.3000
Epoch 24/30
4/4 [=====] - 0s 824us/step - loss: 0.2968
Epoch 25/30
4/4 [=====] - 0s 833us/step - loss: 0.2938
Epoch 26/30
4/4 [=====] - 0s 878us/step - loss: 0.2908
Epoch 27/30
4/4 [=====] - 0s 807us/step - loss: 0.2878
Epoch 28/30
4/4 [=====] - 0s 829us/step - loss: 0.2850
Epoch 29/30
4/4 [=====] - 0s 842us/step - loss: 0.2821
Epoch 30/30
4/4 [=====] - 0s 799us/step - loss: 0.2794
TEST
[[ True]
 [ True]
 [ True]
 [ True]]
```

シード値をゼロから1に変更。

jupyter 4_3_keras_codes (unsaved changes)  Logout

File Edit View Insert Cell Kernel Widgets Help Trusted | Python 3

In [4]:

```
# モジュール読み込み
import numpy as np
from keras.models import Sequential
from keras.layers import Dense, Activation
from keras.optimizers import SGD

# 亂数を固定値で初期化
np.random.seed(1)

# シグモイドの単純パーセプトロン作成
model = Sequential()
model.add(Dense(input_dim=2, units=1))
model.add(Activation('sigmoid'))
model.summary()

model.compile(loss='binary_crossentropy', optimizer=SGD(lr=0.1))

# トレーニング用入力Xと正解データT
X = np.array([[0,0], [0,1], [1,0], [1,1]] )
T = np.array([0, [1], [1], [1]]) 

# トレーニング
model.fit(X, T, epochs=30, batch_size=1)

# トレーニングの入力を適用して実際に分類
Y = model.predict_classes(X, batch_size=1)

print("TEST")
print(Y == T)
```

Model: "sequential_3"

Layer (type)	Output Shape	Param #
dense_3 (Dense)	(None, 1)	3

jupyter 4_3_keras_codes (unsaved changes)

File Edit View Insert Cell Kernel Widgets Help Trusted

Code

```
4/4 [=====] - 0s 629us/step - loss: 0.3555
Epoch 15/30
4/4 [=====] - 0s 638us/step - loss: 0.3506
Epoch 16/30
4/4 [=====] - 0s 640us/step - loss: 0.3460
Epoch 17/30
4/4 [=====] - 0s 567us/step - loss: 0.3417
Epoch 18/30
4/4 [=====] - 0s 575us/step - loss: 0.3375
Epoch 19/30
4/4 [=====] - 0s 608us/step - loss: 0.3334
Epoch 20/30
4/4 [=====] - 0s 595us/step - loss: 0.3294
Epoch 21/30
4/4 [=====] - 0s 656us/step - loss: 0.3257
Epoch 22/30
4/4 [=====] - 0s 589us/step - loss: 0.3220
Epoch 23/30
4/4 [=====] - 0s 618us/step - loss: 0.3186
Epoch 24/30
4/4 [=====] - 0s 627us/step - loss: 0.3151
Epoch 25/30
4/4 [=====] - 0s 625us/step - loss: 0.3117
Epoch 26/30
4/4 [=====] - 0s 570us/step - loss: 0.3085
Epoch 27/30
4/4 [=====] - 0s 600us/step - loss: 0.3053
Epoch 28/30
4/4 [=====] - 0s 621us/step - loss: 0.3022
Epoch 29/30
4/4 [=====] - 0s 645us/step - loss: 0.2991
Epoch 30/30
4/4 [=====] - 0s 633us/step - loss: 0.2961
TEST
[[False]
 [ True]
 [ True]
 [ True]]
```

少し精度が落ちた。

jupyter 4_3_keras_codes (unsaved changes)

File Edit View Insert Cell Kernel Widgets Help Trusted Python 3 Logout

In [5]:

```
# モジュール読み込み
import numpy as np
from keras.models import Sequential
from keras.layers import Dense, Activation
from keras.optimizers import SGD

#乱数を固定値で初期化
np.random.seed(0)

#シグモイドの単純パーセプトロン作成
model = Sequential()
model.add(Dense(input_dim=2, units=1))
model.add(Activation('sigmoid'))
model.summary()

model.compile(loss='binary_crossentropy', optimizer=SGD(lr=0.1))

#トレーニング用入力Xと正解データT
X = np.array( [[0,0], [0,1], [1,0], [1,1] ] )
T = np.array( [[0], [1], [1], [1]] )

#トレーニング
model.fit(X, T, epochs=100, batch_size=1)

#トレーニングの入力を流用して実際に分類
Y = model.predict_classes(X, batch_size=1)

print("TEST")
print(Y == T)
```

Model: "sequential_4"

Layer (type)	Output Shape	Param #
dense_4 (Dense)	(None, 1)	3

jupyter 4_3_keras_codes (unsaved changes)

File Edit View Insert Cell Kernel Widgets Help Trusted Python 3 Logout

In [5]:

```
#トレーニングの入力を流用して実際に分類
Y = model.predict_classes(X, batch_size=1)

print("TEST")
print(Y == T)
```

Epoch 94/100

4/4 [=====] - 0s 646us/step - loss: 0.1702
Epoch 95/100
4/4 [=====] - 0s 640us/step - loss: 0.1692
Epoch 96/100
4/4 [=====] - 0s 655us/step - loss: 0.1681
Epoch 97/100
4/4 [=====] - 0s 635us/step - loss: 0.1671
Epoch 98/100
4/4 [=====] - 0s 686us/step - loss: 0.1660
Epoch 99/100
4/4 [=====] - 0s 740us/step - loss: 0.1650
Epoch 100/100
4/4 [=====] - 0s 674us/step - loss: 0.1640

TEST

[[True]
[True]
[True]
[True]]

エポック数を 30 から 100 に変更。誤差が減った。

AND 回路で学習。

jupyter 4_3_keras_codes (unsaved changes) Logout

File Edit View Insert Cell Kernel Widgets Help Trusted | Python 3

In [6]:

```
# モジュール読み込み
import numpy as np
from keras.models import Sequential
from keras.layers import Dense, Activation
from keras.optimizers import SGD

# 亂数を固定値で初期化
np.random.seed(0)

# シグモイドの単純バーセプトロン作成
model = Sequential()
model.add(Dense(input_dim=2, units=1))
model.add(Activation('sigmoid'))
model.summary()

model.compile(loss='binary_crossentropy', optimizer=SGD(lr=0.1))

# トレーニング用入力 X と正解データ T
X = np.array( [[0,0], [0,1], [1,0], [1,1] ] )
#T = np.array( [[0], [1], [1], [1]] )
T = np.array( [[0], [0], [0], [1]] )

# トレーニング
model.fit(X, T, epochs=30, batch_size=1)

# トレーニングの入力を流用して実際に分類
Y = model.predict_classes(X, batch_size=1)

print("TEST")
print(Y == T)
```

Model: "sequential_5"

jupyter 4_3_keras_codes (unsaved changes) Logout

File Edit View Insert Cell Kernel Widgets Help Trusted | Python 3

```
4/4 [=====] - 0s 624us/step - loss: 0.4565
Epoch 19/30
4/4 [=====] - 0s 576us/step - loss: 0.4512
Epoch 20/30
4/4 [=====] - 0s 740us/step - loss: 0.4459
Epoch 21/30
4/4 [=====] - 0s 705us/step - loss: 0.4403
Epoch 22/30
4/4 [=====] - 0s 667us/step - loss: 0.4354
Epoch 23/30
4/4 [=====] - 0s 654us/step - loss: 0.4309
Epoch 24/30
4/4 [=====] - 0s 607us/step - loss: 0.4264
Epoch 25/30
4/4 [=====] - 0s 621us/step - loss: 0.4219
Epoch 26/30
4/4 [=====] - 0s 653us/step - loss: 0.4176
Epoch 27/30
4/4 [=====] - 0s 620us/step - loss: 0.4132
Epoch 28/30
4/4 [=====] - 0s 624us/step - loss: 0.4090
Epoch 29/30
4/4 [=====] - 0s 649us/step - loss: 0.4046
Epoch 30/30
4/4 [=====] - 0s 661us/step - loss: 0.4011
TEST
[[ True]
 [ True]
 [ True]
 [ True]]
```

AND 回路で学習された。

XOR回路を学習する

jupyter 4_3_keras_codes (autosaved)

File Edit View Insert Cell Kernel Widgets Help Trusted Python 3 Logout

```
In [7]: # モジュール読み込み
import numpy as np
from keras.models import Sequential
from keras.layers import Dense, Activation
from keras.optimizers import SGD

# 亂数を固定値で初期化
np.random.seed(0)

# シグモイドの単純パーセプトロン作成
model = Sequential()
model.add(Dense(input_dim=2, units=1))
model.add(Activation('sigmoid'))
model.summary()

model.compile(loss='binary_crossentropy', optimizer=SGD(lr=0.1))

# トレーニング用入力 X と正解データ T
X = np.array([ [0,0], [0,1], [1,0], [1,1] ])
#T = np.array([ [0], [1], [1], [0] ])
#T = np.array([ [0], [0], [0], [1] ]) # AND回路
T = np.array([ [0], [1], [1], [0] ]) # XOR回路

# トレーニング
model.fit(X, T, epochs=30, batch_size=1)

# トレーニングの入力を流用して実際に分類
Y = model.predict_classes(X, batch_size=1)

print("TEST")
print(Y == T)

Model: "sequential_6"
-----
```

Layer (type)	Output Shape	Param #
Sequential	(None, 1)	3
Dense	(None, 1)	2

jupyter 4_3_keras_codes (unsaved changes)

File Edit View Insert Cell Kernel Widgets Help Trusted Python

```
Epoch 16/30
4/4 [=====] - 0s 618us/step - loss: 0.7425
Epoch 17/30
4/4 [=====] - 0s 621us/step - loss: 0.7422
Epoch 18/30
4/4 [=====] - 0s 638us/step - loss: 0.7404
Epoch 19/30
4/4 [=====] - 0s 587us/step - loss: 0.7394
Epoch 20/30
4/4 [=====] - 0s 588us/step - loss: 0.7393
Epoch 21/30
4/4 [=====] - 0s 573us/step - loss: 0.7380
Epoch 22/30
4/4 [=====] - 0s 595us/step - loss: 0.7368
Epoch 23/30
4/4 [=====] - 0s 586us/step - loss: 0.7371
Epoch 24/30
4/4 [=====] - 0s 600us/step - loss: 0.7363
Epoch 25/30
4/4 [=====] - 0s 570us/step - loss: 0.7356
Epoch 26/30
4/4 [=====] - 0s 644us/step - loss: 0.7348
Epoch 27/30
4/4 [=====] - 0s 594us/step - loss: 0.7344
Epoch 28/30
4/4 [=====] - 0s 585us/step - loss: 0.7338
Epoch 29/30
4/4 [=====] - 0s 643us/step - loss: 0.7328
Epoch 30/30
4/4 [=====] - 0s 633us/step - loss: 0.7326
TEST
[[ True]
 [ True]
 [False]
 [False]]
```

XOR 回路は誤差が大きい。

jupyter 4_3_keras_codes (unsaved changes) Logout

File Edit View Insert Cell Kernel Widgets Help Trusted Python 3

In [8]:

```
# モジュール読み込み
import numpy as np
from keras.models import Sequential
from keras.layers import Dense, Activation
from keras.optimizers import SGD

# 異数を固定値で初期化
np.random.seed(0)

# シグモイドの単純バーセプトロン作成
model = Sequential()
model.add(Dense(input_dim=2, units=1))
model.add(Activation('sigmoid'))
model.summary()

model.compile(loss='binary_crossentropy', optimizer=SGD(lr=0.1))

# トレーニング用入力 X と正解データ T
X = np.array([[0,0], [0,1], [1,0], [1,1]])
#T = np.array([0,1,1,0])
#T = np.array([0,0,1,1]) # AND回路
T = np.array([0,1,1,0]) # XOR回路

# トレーニング
model.fit(X, T, epochs=300, batch_size=1)

# トレーニングの入力を流用して実際に分類
Y = model.predict_classes(X, batch_size=1)

print("TEST")
print(Y == T)
```

4/4 [=====] - 0s 769us/step - loss: 0.7190
Epoch 294/300
4/4 [=====] - 0s 746us/step - loss: 0.7187
Epoch 295/300
4/4 [=====] - 0s 741us/step - loss: 0.7187

jupyter 4_3_keras_codes (unsaved changes) Logout

File Edit View Insert Cell Kernel Widgets Help Trusted Python 3

```
#T = np.array([0,1,1,0]) # AND回路
T = np.array([0,1,1,0]) # XOR回路

# トレーニング
model.fit(X, T, epochs=300, batch_size=1)

# トレーニングの入力を流用して実際に分類
Y = model.predict_classes(X, batch_size=1)

print("TEST")
print(Y == T)
```

4/4 [=====] - 0s 769us/step - loss: 0.7190
Epoch 294/300
4/4 [=====] - 0s 746us/step - loss: 0.7187
Epoch 295/300
4/4 [=====] - 0s 741us/step - loss: 0.7187
Epoch 296/300
4/4 [=====] - 0s 734us/step - loss: 0.7183
Epoch 297/300
4/4 [=====] - 0s 765us/step - loss: 0.7187
Epoch 298/300
4/4 [=====] - 0s 703us/step - loss: 0.7188
Epoch 299/300
4/4 [=====] - 0s 736us/step - loss: 0.7187
Epoch 300/300
4/4 [=====] - 0s 800us/step - loss: 0.7190
TEST
[[False]
 [False]
 [True]
 [True]]

エポック数を 300 に変更してもロスが減ってこない。

1層のパーセプトロンでは線形の表現はできるが、XOR回路は非線形なデータセットになるため、学習ができなかった。

jupyter 4_3_keras_codes (unSaved changes) Logout

File Edit View Insert Cell Kernel Widgets Help Trusted Python 3

In [9]:

```
# モジュール読み込み
import numpy as np
from keras.models import Sequential
from keras.layers import Dense, Activation
from keras.optimizers import SGD

#乱数を固定値で初期化
np.random.seed(0)

#シグモイドの単純パーセプトロン作成
model = Sequential()
model.add(Dense(input_dim=2, units=1))
model.add(Activation('sigmoid'))
model.summary()

model.compile(loss='binary_crossentropy', optimizer=SGD(lr=0.1))

#トレーニング用入力 X と正解データ T
X = np.array( [[0,0], [0,1], [1,0], [1,1]] )
T = np.array( [[0], [1], [1], [0]] )
#T = np.array( [[0], [0], [0], [1]] ) # AND回路
#T = np.array( [[0], [1], [1], [0]] ) # XOR回路

#トレーニング
model.fit(X, T, epochs=30, batch_size=10)

#トレーニングの入力を流用して実際に分類
Y = model.predict_classes(X, batch_size=10)

print("TEST")
print(Y == T)
```

Model: "sequential_8"

Layer (type)	Output Shape	Param #
Sequential	(None, 1)	3
Dense	(None, 1)	3

jupyter 4_3_keras_codes (unsaved changes)

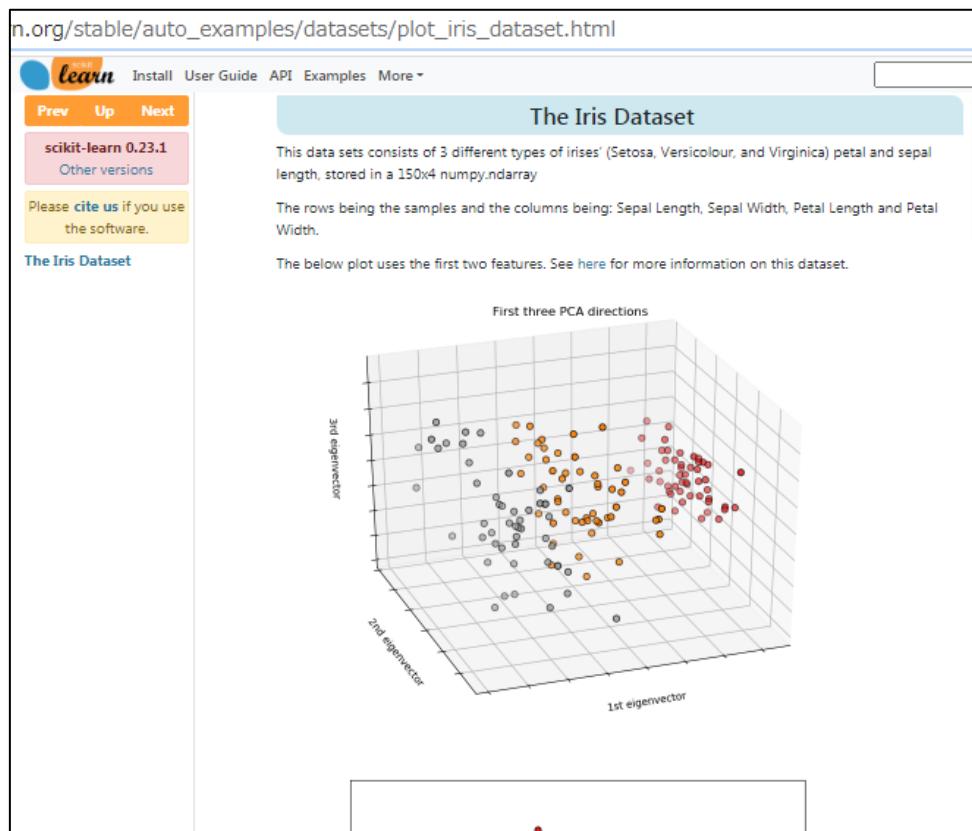
File Edit View Insert Cell Kernel Widgets Help Trusted Python

4/4 [=====] - 0s 139us/step - loss: 0.3837
Epoch 18/30
4/4 [=====] - 0s 142us/step - loss: 0.3815
Epoch 19/30
4/4 [=====] - 0s 138us/step - loss: 0.3794
Epoch 20/30
4/4 [=====] - 0s 128us/step - loss: 0.3773
Epoch 21/30
4/4 [=====] - 0s 150us/step - loss: 0.3753
Epoch 22/30
4/4 [=====] - 0s 153us/step - loss: 0.3733
Epoch 23/30
4/4 [=====] - 0s 151us/step - loss: 0.3714
Epoch 24/30
4/4 [=====] - 0s 135us/step - loss: 0.3696
Epoch 25/30
4/4 [=====] - 0s 151us/step - loss: 0.3678
Epoch 26/30
4/4 [=====] - 0s 308us/step - loss: 0.3660
Epoch 27/30
4/4 [=====] - 0s 305us/step - loss: 0.3643
Epoch 28/30
4/4 [=====] - 0s 148us/step - loss: 0.3626
Epoch 29/30
4/4 [=====] - 0s 140us/step - loss: 0.3610
Epoch 30/30
4/4 [=====] - 0s 139us/step - loss: 0.3594
TEST
[[False]
 [True]
 [True]
 [True]]

OR回路に戻して、バッチサイズを1から10に変更。

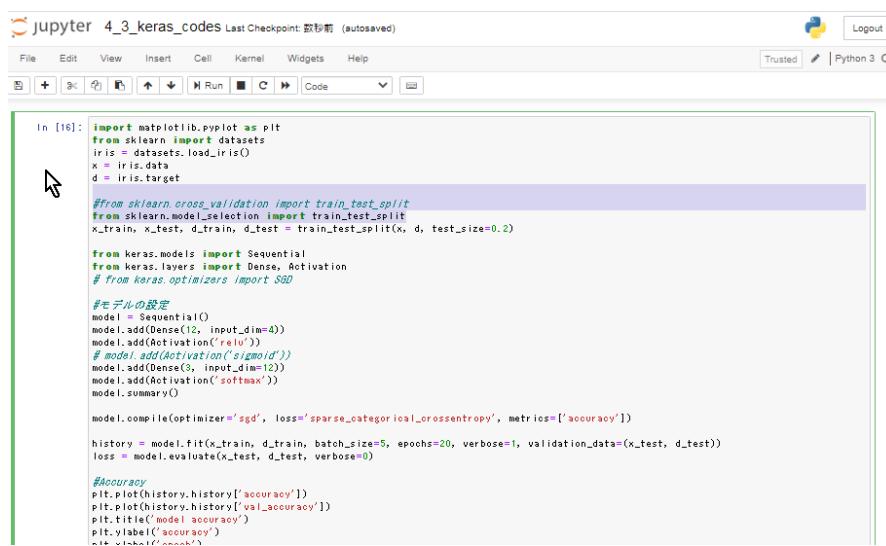
誤差は変わらないが、学習時間が少し減った。

3.1.14 Keras、iris（分類）



scikit-learn のサイトで iris データセットの可視化を確認[1]

花びらの大きさやがく片の長さなどの 4 種類の x が与えられて、アヤメ(花)の種類を分類するもの。



In [16]:

```
import matplotlib.pyplot as plt
from sklearn import datasets
iris = datasets.load_iris()
X = iris.data
y = iris.target

# from sklearn.cross_validation import train_test_split
# from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2)

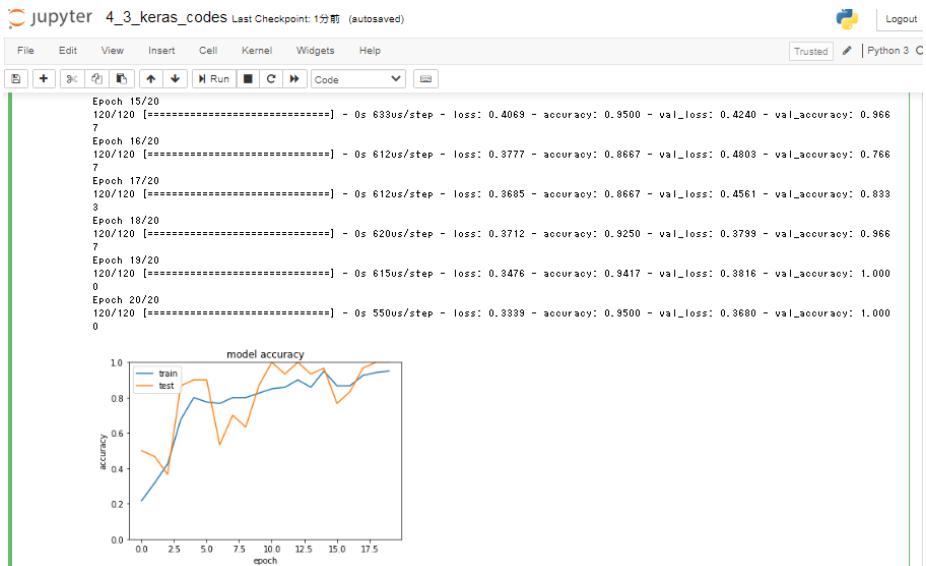
from keras.models import Sequential
from keras.layers import Dense, Activation
# from keras.optimizers import SGD

# モデルの設定
model = Sequential()
model.add(Dense(4, input_dim=4))
model.add(Activation('relu'))
# model.add(Activation('sigmoid'))
model.add(Dense(3, input_dim=4))
model.add(Activation('softmax'))
model.summary()

model.compile(optimizer='sgd', loss='categorical_crossentropy', metrics=['accuracy'])

history = model.fit(X_train, y_train, batch_size=5, epochs=20, verbose=1, validation_data=(X_test, y_test))

# Accuracy
plt.plot(history.history['accuracy'])
plt.plot(history.history['val_accuracy'])
plt.title('Model accuracy')
plt.ylabel('Accuracy')
plt.xlabel('Epoch')
```



train と test が同じように上昇している。

The screenshot shows a Jupyter Notebook interface with the title "jupyter 4_3_keras_codes". The notebook has a "Trusted" status and is using Python 3.

The code cell (In [17]) contains the following Python code:

```

In [17]: import matplotlib.pyplot as plt
from sklearn import datasets
iris = datasets.load_iris()
x = iris.data
d = iris.target

#from sklearn.cross_validation import train_test_split
#from sklearn.model_selection import train_test_split
x_train, x_test, d_train, d_test = train_test_split(x, d, test_size=0.2)

from keras.models import Sequential
from keras.layers import Dense, Activation
#from keras.optimizers import SGD

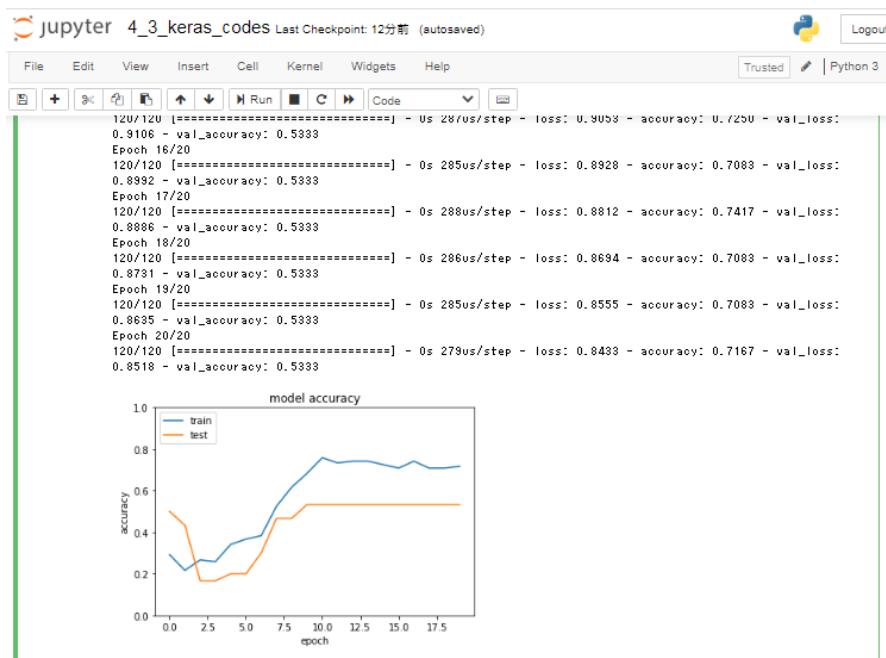
#モデルの設定
model = Sequential()
model.add(Dense(12, input_dim=4))
#model.add(Activation('relu'))
model.add(Activation('sigmoid'))
model.add(Dense(3, input_dim=12))
model.add(Activation('softmax'))
model.summary()

model.compile(optimizer='sgd', loss='sparse_categorical_crossentropy', metrics=['accuracy'])

history = model.fit(x_train, d_train, batch_size=5, epochs=20, verbose=1, validation_data=(x_test, d_test))
loss = model.evaluate(x_test, d_test, verbose=0)

#Accuracy
plt.plot(history.history['accuracy'])
plt.plot(history.history['val_accuracy'])
plt.title('model accuracy')

```



中間層の活性関数を sigmoid に変更すると精度が落ちた。

```

In [18]: import matplotlib.pyplot as plt
from sklearn import datasets
iris = datasets.load_iris()
x = iris.data
d = iris.target

#from sklearn.cross_validation import train_test_split
from sklearn.model_selection import train_test_split
x_train, x_test, d_train, d_test = train_test_split(x, d, test_size=0.2)

from keras.models import Sequential
from keras.layers import Dense, Activation
from keras.optimizers import SGD

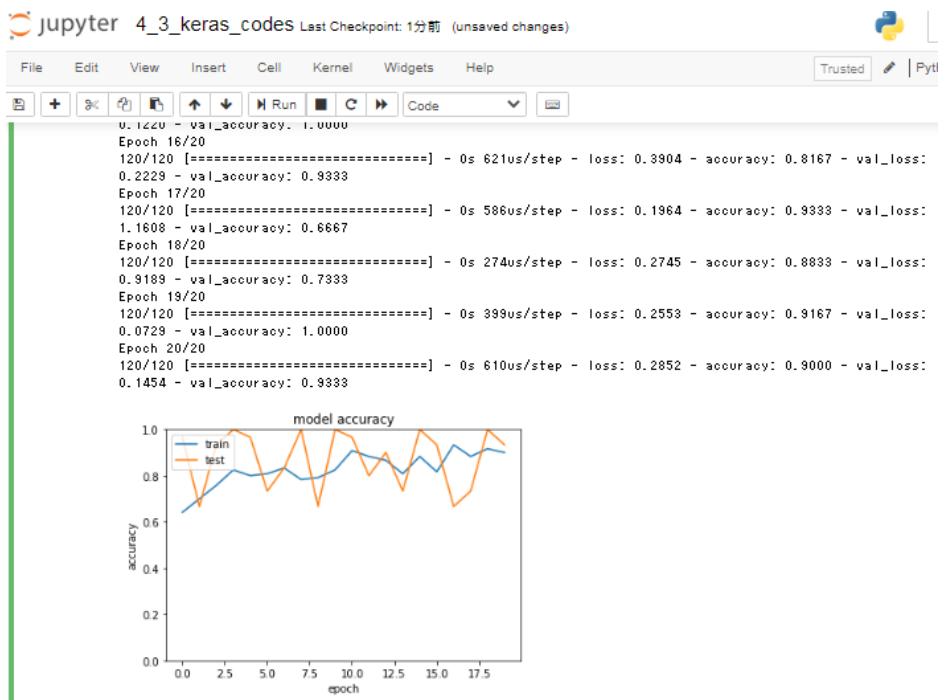
#モデルの設定
model = Sequential()
model.add(Dense(12, input_dim=4))
model.add(Activation('relu'))
#model.add(Activation('sigmoid'))
model.add(Dense(3, input_dim=12))
model.add(Activation('softmax'))
model.summary()

#model.compile(optimizer='sgd', loss='sparse_categorical_crossentropy', metrics=['accuracy'])
model.compile(optimizer=SGD(lr=0.1), loss="sparse_categorical_crossentropy", metrics=['accuracy'])

history = model.fit(x_train, d_train, batch_size=5, epochs=20, verbose=1, validation_data=(x_test, d_test))
loss = model.evaluate(x_test, d_test, verbose=0)

#Accuracy
plt.plot(history.history['accuracy'])
plt.plot(history.history['val_accuracy'])
plt.title('model accuracy')

```



SGD を import し optimizer を SGD(lr=0.1)に変更すると、早い段階で収束する。

3.1.15 Keras、MNIST（分類）

```

In [27]: # 必要なライブラリのインポート
import sys, os
sys.path.append(os.pardir) # 父ディレクトリのファイルをインポートするための設定
import keras
import matplotlib.pyplot as plt
from data.mnist import load_mnist

(x_train, d_train), (x_test, d_test) = load_mnist(normalize=True, one_hot_label=True)

# 必要なライブラリのインポート、最適化手法はAdamを使う
from keras.models import Sequential
from keras.layers import Dense, Dropout
from keras.optimizers import Adam

# モデル作成
model = Sequential()
model.add(Dense(units=512, activation='relu', input_shape=(784,)))
model.add(Dropout(0.2))
model.add(Dense(units=512, activation='relu'))
model.add(Dropout(0.2))
model.add(Dense(units=10, activation='softmax'))
model.summary()

# バッチサイズ、エポック数
batch_size = 128
epochs = 20

model.compile(loss='categorical_crossentropy',
              optimizer='adam', # errorとなるため
              #optimizer='adam(lr=0.001, beta_1=0.9, beta_2=0.999, epsilon=None, decay=0.0, amsgrad=False),
              metrics=['accuracy'])

history = model.fit(x_train, d_train, batch_size=batch_size, epochs=epochs, verbose=1, validation_data=(x_test, d_test))
loss = model.evaluate(x_test, d_test, verbose=0)
print('Test loss:', loss[0])

```



jupyter 4_3_keras_codes Last Checkpoint: 6分前 (unsaved changes)

File Edit View Insert Cell Kernel Widgets Help Trusted Python 3

- 誤差関数をsparse_categorical_crossentropyに変更しよう
- Adamの引数の値を変更しよう

```
In [28]: #必要なライブラリのインポート
import tensorflow as tf
sys.path.append(os.pardir) # 親ディレクトリのファイルをインポートするための設定
import keras
import matplotlib.pyplot as plt
from tensorflow import keras
from tensorflow.keras import Sequential
from tensorflow.keras.layers import Dense, Dropout
from tensorflow.keras.optimizers import Adam

# モデル作成
model = Sequential()
model.add(Dense(units=512, activation='relu', input_shape=(784,)))
model.add(Dropout(0.2))
model.add(Dense(units=512, activation='relu'))
model.add(Dropout(0.2))
model.add(Dense(units=10, activation='softmax'))
model.summary()

# バッチサイズ、エポック数
batch_size = 128
epochs = 20

model.compile(loss='categorical_crossentropy',
              optimizer='adam', # errorとなるため
              metrics=['accuracy'])
```

jupyter 4_3_keras_codes Last Checkpoint: 6分前 (autosaved)

File Edit View Insert Cell Kernel Widgets Help Trusted Python 3

```
--> 32 history = model.fit(x_train, d_train, batch_size=batch_size, epochs=epochs, verbose=1, validation_data=(x_test, d_test))
33 loss = model.evaluate(x_test, d_test, verbose=0)
34 print('Test loss:', loss[0])

/opt/conda/lib/python3.6/site-packages/keras/engine/training.py in fit(self, x, y, batch_size, epochs, verbose, callbacks, validation_data, shuffle, worker, use_multiprocessing, **kwargs)
    1152         sample_weight=sample_weight,
    1153         class_weight=class_weight,
--> 1154         batch_size=batch_size,
    1155         )
    1156         # Prepare validation data.

/opt/conda/lib/python3.6/site-packages/keras/engine/training.py in _standardize_user_data(self, x, y, sample_weight, class_weight,
    check_array_lengths, batch_size)
    619             feed_output_shapes,
    620             check_batch_axis=False, # Don't enforce the batch size.
--> 621             exception_prefix='target')
    622
    623     # Generate sample-wise weight values given the `sample_weight` and
```

ValueError: Error when checking target: expected dense_44 to have shape (10,) but got array with shape (1,)

load_mnist の one_hot_label を False に変更すると fit でエラーとなる。

categorical_crossentropy を使っているため。

次で、これを sparse_categorical_crossentropy に変更する。

Jupyter 4_3_keras_codes Last Checkpoint: 1分前 (unsaved changes)

```
In [29]: # 必要なライブラリのインポート
import sys, os
sys.path.append(os.getcwd()) # 転ディレクトリのファイルをインポートするための設定
import keras
import matplotlib.pyplot as plt
from keras.datasets import load_mnist

(x_train, d_train), (x_test, d_test) = load_mnist(normalize=True, one_hot_label=False)

# 必要なライブラリのインポート。最適化手法はAdamを使う
from keras.models import Sequential
from keras.layers import Dense, Dropout
from keras.optimizers import Adam

# モデル作成
model = Sequential()
model.add(Dense(units=512, activation='relu', input_shape=(784,)))
model.add(Dropout(0.2))
model.add(Dense(units=512, activation='relu'))
model.add(Dropout(0.2))
model.add(Dense(units=10, activation='softmax'))
model.summary()

# バッチサイズ、エポック数
batch_size = 128
epochs = 20

model.compile(loss='sparse_categorical_crossentropy', # categorical_crossentropy
              optimizer='adam', # errorとなるため
              #optimizer='Adam(lr=0.001, beta_1=0.9, beta_2=0.999, epsilon=None, decay=0.0, amsgrad=False),
              metrics=['accuracy'])

history = model.fit(x_train, d_train, batch_size=batch_size, epochs=epochs, verbose=1, validation_data=(x_test, d_test))
loss = model.evaluate(x_test, d_test, verbose=0)
print('Test loss:', loss[0])
```

Jupyter 4_3_keras_CODES Last Checkpoint: 1分前 (unsaved changes)

```
File Edit View Insert Cell Kernel Widgets Help Trusted Python 3 Logout
```

```
Epoch 17/20
60000/60000 [=====] - 2s 41ms/step - loss: 0.0180 - accuracy: 0.9936 - val_loss: 0.0747 - val_accuracy: 0.9831
Epoch 18/20
60000/60000 [=====] - 2s 41ms/step - loss: 0.0166 - accuracy: 0.9948 - val_loss: 0.0755 - val_accuracy: 0.9841
Epoch 19/20
60000/60000 [=====] - 2s 41ms/step - loss: 0.0151 - accuracy: 0.9952 - val_loss: 0.0846 - val_accuracy: 0.9813
Epoch 20/20
60000/60000 [=====] - 2s 41ms/step - loss: 0.0147 - accuracy: 0.9952 - val_loss: 0.0833 - val_accuracy: 0.9829
Test loss: 0.0830384823020809
Test accuracy: 0.9829000234603882
```

Epoch	Training Accuracy	Testing Accuracy
0.0	0.93	0.97
2.5	0.95	0.98
5.0	0.97	0.98
7.5	0.98	0.98
10.0	0.99	0.98
12.5	0.99	0.98
15.0	0.99	0.98
17.5	0.99	0.98

ラベルが One-hot でない時は、sparse_categorical_crossentropy に変更すると最初と同じように学習が行えた。

jupyter 4_3_keras_codes Last Checkpoint: 3分前 (unsaved changes)

In [31]:

```
# 必要なライブラリのインポート
import os
sys.path.append(os.pardir) # 総ディレクトリのファイルをインポートするための設定
import keras
import matplotlib.pyplot as plt
from keras.datasets import mnist
from keras.utils import load_mnist

(x_train, d_train), (x_test, d_test) = load_mnist(normalize=True, one_hot_label=True)

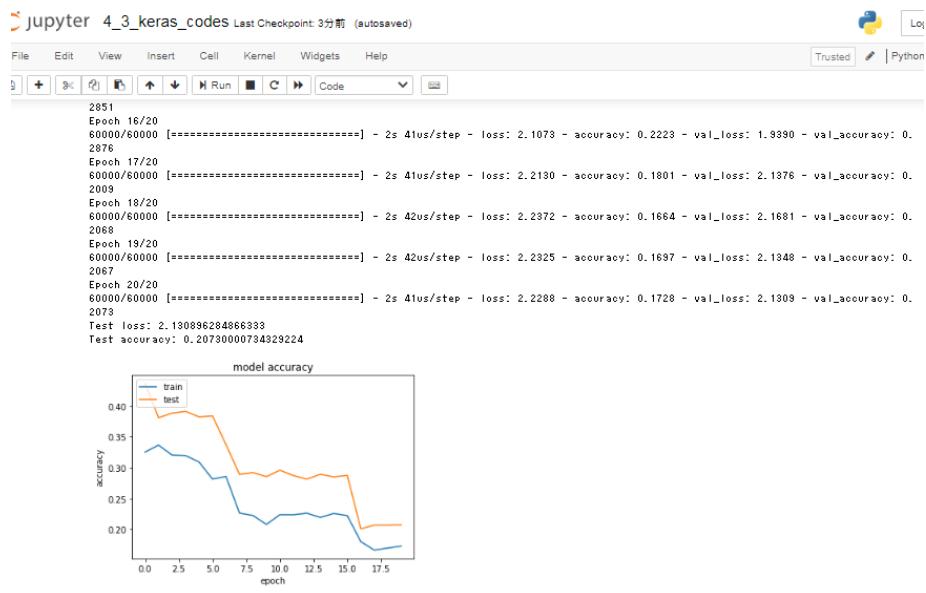
# 必要なライブラリのインポート。最適化手法はAdamを使う
from keras.models import Sequential
from keras.layers import Dense, Dropout
from keras.optimizers import Adam

# モデル作成
model = Sequential()
model.add(Dense(units=512, activation='relu', input_shape=(784,)))
model.add(Dropout(0.2))
model.add(Dense(units=512, activation='relu'))
model.add(Dropout(0.2))
model.add(Dense(units=10, activation='softmax'))
model.summary()

# バッチサイズ、エポック数
batch_size = 128
epochs = 20

model.compile(loss='categorical_crossentropy',
              optimizer='adam', # errorとなるため
              optimizer=Adam(lr=0.1, beta_1=0.9, beta_2=0.999, epsilon=0.0001, decay=0.0, smrgrad=False),
              metrics=['accuracy'])

history = model.fit(x_train, d_train, batch_size=batch_size, epochs=epochs, verbose=1, validation_data=(x_test, d_test))
```



Adam の引数を変更。lr=0.001 から 0.1 にすると、学習されなくなった。

3.1.16 Keras、MNIST (CNNで分類)

jupyter 4_3_keras_codes Last Checkpoint: 7分前 (autosaved)

File Edit View Insert Cell Kernel Widgets Help Trusted Python 3

CNN分類(mnist)

実行に時間がかかるため割愛

```
In [21]: # 必要なライブラリのインポート
import sys, os
sys.path.append(os.pardir) # 父ディレクトリのファイルをインポートするための設定
import keras
import matplotlib.pyplot as plt
from data.mnist import load_mnist

(x_train, d_train), (x_test, d_test) = load_mnist(normalize=True, one_hot_label=True)

# 行列として入力するための加工
batch_size = 128
num_classes = 10
epochs = 20

img_rows, img_cols = 28, 28

x_train = x_train.reshape(x_train.shape[0], img_rows, img_cols, 1)
x_test = x_test.reshape(x_test.shape[0], img_rows, img_cols, 1)
input_shape = (img_rows, img_cols, 1)

# 必要なライブラリのインポート、最適化手法はAdamを使う
from keras.models import Sequential
from keras.layers import Dense, Dropout, Flatten
from keras.layers import Conv2D, MaxPooling2D
from keras.optimizers import Adam

model = Sequential()
model.add(Conv2D(32, kernel_size=(3, 3),
                activation='relu',
                input_shape=input_shape))
```

jupyter 4_3_keras_codes Last Checkpoint: 7分前 (autosaved)

Edit View Insert Cell Kernel Widgets Help Trusted Python

```
Epoch 17/20
60000/60000 [=====] - 26s 430us/step - loss: 0.0163 - accuracy: 0.9944 - val_loss: 0.0273 - val_accuracy: 0.9325
Epoch 18/20
60000/60000 [=====] - 26s 430us/step - loss: 0.0140 - accuracy: 0.9953 - val_loss: 0.0290 - val_accuracy: 0.9325
Epoch 19/20
60000/60000 [=====] - 26s 429us/step - loss: 0.0172 - accuracy: 0.9945 - val_loss: 0.0275 - val_accuracy: 0.9342
Epoch 20/20
60000/60000 [=====] - 26s 429us/step - loss: 0.0145 - accuracy: 0.9951 - val_loss: 0.0315 - val_accuracy: 0.9332
```

model accuracy

Epoch	train accuracy	val accuracy
0	0.93	0.93
2.5	0.98	0.98
5.0	0.99	0.99
7.5	0.99	0.99
10.0	0.99	0.99
12.5	0.99	0.99
15.0	0.99	0.99
17.5	0.99	0.99
20	0.99	0.99

3.1.17 Keras、cifar10

cs.toronto.edu/~kriz/cifar.html

The CIFAR-10 and CIFAR-100 are labeled subsets of the [80 million tiny images](#) dataset. They were collected by Alex Krizhevsky, Vinod Nair, and Geoffrey Hinton.

The CIFAR-10 dataset

The CIFAR-10 dataset consists of 60000 32x32 colour images in 10 classes, with 6000 images per class. There are 50000 training images and 10000 test images.

The dataset is divided into five training batches and one test batch, each with 10000 images. The test batch contains exactly 1000 randomly-selected images from each class. The training batches contain the remaining images in random order, but some training batches may contain more images from one class than another. Between them, the training batches contain exactly 50000 images from each class.

Here are the classes in the dataset, as well as 10 random images from each:

airplane	
automobile	
bird	
cat	
deer	
dog	
frog	
horse	
ship	
truck	

The classes are completely mutually exclusive. There is no overlap between automobiles and trucks. "Automobile" includes sedans, SUVs, things of

トロント大学のサイトにて cifar10 の内容を確認した。[2]

10 種のラベル「飛行機、自動車、鳥、猫、鹿、犬、蛙、馬、船、トラック」をもつ、
32x32 px のカラー画像データ、トレーニングデータ数:50000, テストデータ数:10000

jupyter 4_3_keras_codes Last Checkpoint: 28分前 (autosaved)

In [22]:

```
#CIFAR-10のデータセットのインポート
from keras.datasets import cifar10
(x_train, d_train), (x_test, d_test) = cifar10.load_data()

#CIFAR-10の正規化
from keras.utils import to_categorical

# 例画像の正規化
x_train = x_train/255.
x_test = x_test/255.

# クラスラベルの1-hotベクトル化
d_train = to_categorical(d_train, 10)
d_test = to_categorical(d_test, 10)

# CNNの構築
import keras
from keras.models import Sequential
from keras.layers.convolutional import Conv2D, MaxPooling2D
from keras.layers.core import Dense, Dropout, Activation, Flatten
import numpy as np

model = Sequential()

model.add(Conv2D(32, (3, 3), padding='same', input_shape=x_train.shape[1:]))
model.add(Activation('relu'))
model.add(Conv2D(32, (3, 3)))
model.add(Activation('relu'))
model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(Dropout(0.25))

model.add(Conv2D(64, (3, 3), padding='same'))
model.add(Activation('relu'))
model.add(Conv2D(64, (3, 3)))
model.add(Activation('relu'))
```

jupyter 4_3_keras_codes Last Checkpoint: 29分前 (autosaved)

File Edit View Insert Cell Kernel Widgets Help Trusted Python

```
50000/50000 [=====] - 44s 879us/step - loss: 0.7619 - accuracy: 0.7382
Epoch 7/20
50000/50000 [=====] - 44s 880us/step - loss: 0.7195 - accuracy: 0.7469
Epoch 8/20
50000/50000 [=====] - 44s 880us/step - loss: 0.6859 - accuracy: 0.7590
Epoch 9/20
50000/50000 [=====] - 44s 880us/step - loss: 0.6537 - accuracy: 0.7707
Epoch 10/20
50000/50000 [=====] - 44s 880us/step - loss: 0.6352 - accuracy: 0.7781
Epoch 11/20
50000/50000 [=====] - 44s 880us/step - loss: 0.6163 - accuracy: 0.7854
Epoch 12/20
50000/50000 [=====] - 44s 880us/step - loss: 0.5916 - accuracy: 0.7912
Epoch 13/20
50000/50000 [=====] - 44s 880us/step - loss: 0.5774 - accuracy: 0.7972
Epoch 14/20
50000/50000 [=====] - 44s 880us/step - loss: 0.5600 - accuracy: 0.8027
Epoch 15/20
50000/50000 [=====] - 44s 880us/step - loss: 0.5497 - accuracy: 0.8085
Epoch 16/20
50000/50000 [=====] - 44s 880us/step - loss: 0.5312 - accuracy: 0.8133
Epoch 17/20
50000/50000 [=====] - 44s 880us/step - loss: 0.5236 - accuracy: 0.8167
Epoch 18/20
50000/50000 [=====] - 44s 880us/step - loss: 0.5116 - accuracy: 0.8209
Epoch 19/20
50000/50000 [=====] - 44s 880us/step - loss: 0.4976 - accuracy: 0.8251
Epoch 20/20
50000/50000 [=====] - 44s 880us/step - loss: 0.4859 - accuracy: 0.8292
10000/10000 [=====] - 2s 176us/step
[0.6641298703670502, 0.7792999744415283]
```

3.1.18 Keras、RNN

jupyter 4_3_keras_codes Last Checkpoint: 36分前 (autosaved)

File Edit View Insert Cell Kernel Widgets Help Trusted Python 3 Logout

```
In [23]: import sys, os
sys.path.append(os.pardir) # 親ディレクトリのファイルをインポートするための設定
import numpy as np
import matplotlib.pyplot as plt

import keras
from keras.models import Sequential
from keras.layers.core import Dense, Dropout, Activation
from keras.layers.wrappers import TimeDistributed
from keras.optimizers import SGD
from keras.layers.recurrent import SimpleRNN, LSTM, GRU

# データを用意
# 2進数の桁数
binary_dim = 8
# 最大値 + 1
largest_number = pow(2, binary_dim)

# largest_numberまで2進数を用意
binary = np.unpackbits(np.array([range(largest_number)]), dtype=np.uint8).T, ::-1]

# A, B初期化 (a + b = d)
a_int = np.random.randint(largest_number//2, size=20000)
a_bin = binary[a_int] # binary encoding
b_int = np.random.randint(largest_number//2, size=20000)
b_bin = binary[b_int] # binary encoding

x_int = []
x_bin = []
for i in range(10000):
    x_int.append(np.array([a_int[i], b_int[i]]).T)
    x_bin.append(np.array([a_bin[i], b_bin[i]]).T)
```

```

jupyter 4_3_keras_codes Last Checkpoint: 36分钟前 (autosaved)
File Edit View Insert Cell Kernel Widgets Help
Run C Markdown
history = model.fit(x_bin, d_bin.reshape(-1, 8, 1), epochs=5, batch_size=2)

# テスト結果出力
score = model.evaluate(x_bin_test, d_bin_test.reshape(-1,8,1), verbose=0)
print('Test loss:', score[0])
print('Test accuracy:', score[1])

Model: "sequential_19"
Layer (type)          Output Shape        Param # 
simple_rnn_1 (SimpleRNN) (None, 8, 16)      304      
dense_29 (Dense)       (None, 8, 1)        17        
Total params: 321
Trainable params: 321
Non-trainable params: 0

Epoch 1/5
10000/10000 [=====] - 18s 2ms/step - loss: 0.0967 - accuracy: 0.8804
Epoch 2/5
10000/10000 [=====] - 18s 2ms/step - loss: 0.0120 - accuracy: 0.9906
Epoch 3/5
10000/10000 [=====] - 21s 2ms/step - loss: 0.0027 - accuracy: 0.9999
Epoch 4/5
10000/10000 [=====] - 21s 2ms/step - loss: 0.0011 - accuracy: 1.0000
Epoch 5/5
10000/10000 [=====] - 21s 2ms/step - loss: 6.8247e-04 - accuracy: 1.0000
Test loss: 0.0005409065893527927
Test accuracy: 1.0

```

RNN の出力ノード数を 128 に変更した。

```

jupyter 4_3_keras_codes Last Checkpoint: 2分钟前 (autosaved)
File Edit View Insert Cell Kernel Widgets Help
Run C Markdown
x_int = np.array(x_int)
x_bin = np.array(x_bin)
x_int_test = np.array(x_int_test)
x_bin_test = np.array(x_bin_test)

# 正解データ
d_int = s_int * b_int
d_bin = binary[d_int][0:10000]
d_bin_test = binary[d_int][10001:20000]

model = Sequential()

model.add(SimpleRNN(units=128, # 16
                    return_sequences=True,
                    input_shape=[8, 2],
                    go_backwards=False,
                    activation='relu',
                    # dropout=0.5,
                    # recurrent_dropout=0.3,
                    # unroll = True,
                    ))
# 出力層
model.add(Dense(1, activation='sigmoid', input_shape=(-1,2)))
model.summary()
model.compile(loss='mean_squared_error', optimizer=SGD(lr=0.1), metrics=['accuracy'])
# model.compile(loss='mse', optimizer='adam', metrics=['accuracy'])

history = model.fit(x_bin, d_bin.reshape(-1, 8, 1), epochs=5, batch_size=2)

# テスト結果出力
score = model.evaluate(x_bin_test, d_bin_test.reshape(-1,8,1), verbose=0)
print('Test loss:', score[0])
print('Test accuracy:', score[1])

Model: "sequential_29"

```

```

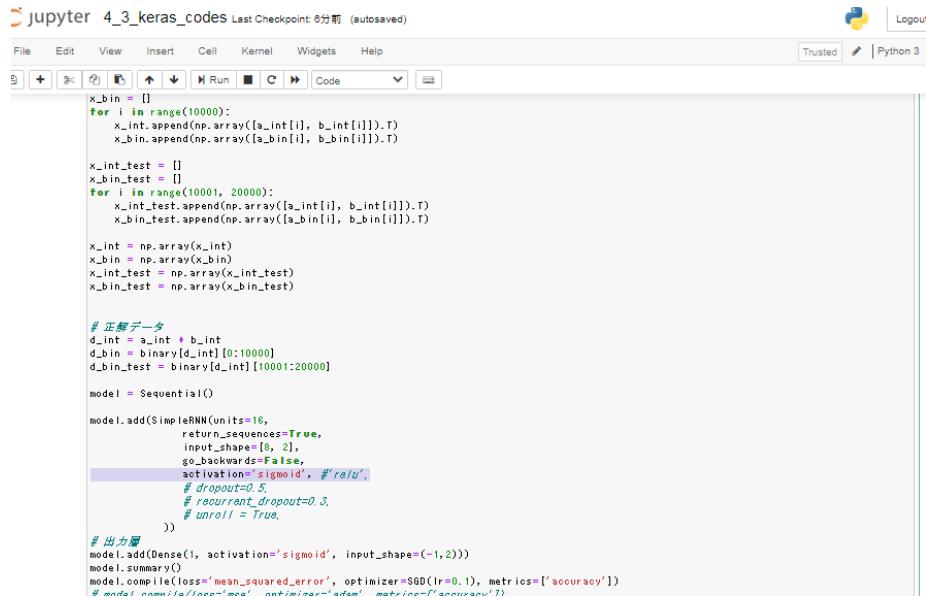
jupyter 4_3_keras_codes Last Checkpoint: 2分钟前 (autosaved)
File Edit View Insert Cell Kernel Widgets Help
Run C Markdown
print('Test accuracy:', score[1])

Model: "sequential_29"
Layer (type)          Output Shape        Param # 
simple_rnn_2 (SimpleRNN) (None, 8, 128)    16768    
dense_57 (Dense)       (None, 8, 1)        129      
Total params: 16,897
Trainable params: 16,897
Non-trainable params: 0

Epoch 1/5
10000/10000 [=====] - 8s 777us/step - loss: 0.0719 - accuracy: 0.9210
Epoch 2/5
10000/10000 [=====] - 8s 19us/step - loss: 0.0018 - accuracy: 1.0000
Epoch 3/5
10000/10000 [=====] - 9s 926us/step - loss: 6.7810e-04 - accuracy: 1.0000
Epoch 4/5
10000/10000 [=====] - 10s 965us/step - loss: 3.9928e-04 - accuracy: 1.0000
Epoch 5/5
10000/10000 [=====] - 11s 1ms/step - loss: 2.7797e-04 - accuracy: 1.0000
Test loss: 0.00023545362648781758
Test accuracy: 1.0

```

同じように学習できた。少し時間がかかったか。



```

jupyter 4_3_keras_codes Last Checkpoint: 5分前 (autosaved)
File Edit View Insert Cell Kernel Widgets Help Trusted Python 3
B + 9: Run C Code ▾
x_int = []
for i in range(10000):
    x_int.append(np.array([a_int[i], b_int[i]]).T)
    x_bin.append(np.array([a_bin[i], b_bin[i]]).T)

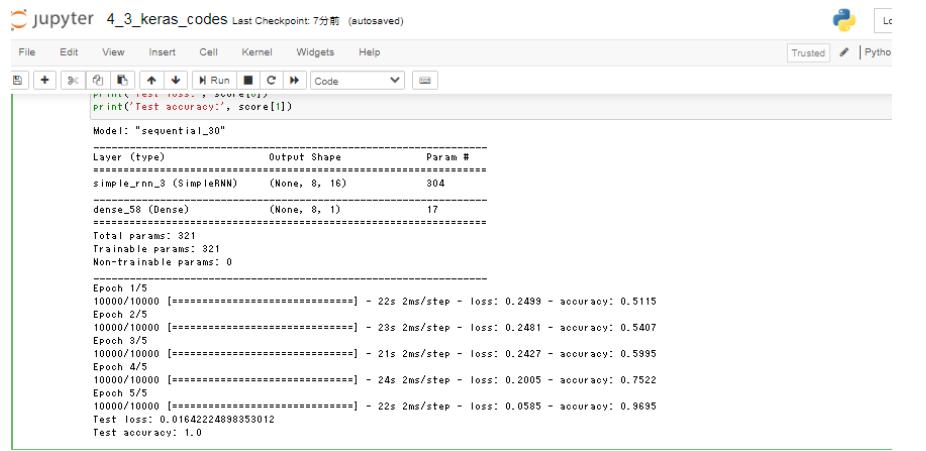
x_int_test = []
x_bin_test = []
for i in range(10001, 20000):
    x_int_test.append(np.array([a_int[i], b_int[i]]).T)
    x_bin_test.append(np.array([a_bin[i], b_bin[i]]).T)

x_int = np.array(x_int)
x_bin = np.array(x_bin)
x_int_test = np.array(x_int_test)
x_bin_test = np.array(x_bin_test)

# 正解データ
d_int = a_int * b_int
d_bin = binary(d_int)[0:10000]
d_bin_test = binary(d_int)[10001:20000]

model = Sequential()
model.add(SimpleRNN(units=16,
                    return_sequences=True,
                    input_shape=[8, 2],
                    go_backwards=False,
                    activation='sigmoid', #relu',
                    #dropout=0.5,
                    # recurrent_dropout=0.3,
                    # unroll=True,
                    ))
# 出力層
model.add(Dense(1, activation='sigmoid', input_shape=(-1, 2)))
model.summary()
model.compile(loss='mean_squared_error', optimizer='SGD(lr=0.1)', metrics=['accuracy'])
# model.compile(loss='mean_squared_error', optimizer='adam', metrics=['accuracy'])

```

```

jupyter 4_3_keras_codes Last Checkpoint: 7分前 (autosaved)
File Edit View Insert Cell Kernel Widgets Help Trusted Python 3
B + 9: Run C Code ▾
print('Test accuracy:', score[1])
Model: "sequential_30"
-----  

Layer (type) Output Shape Param #
-----  

simple_rnn_3 (SimpleRNN) (None, 8, 16) 304  

-----  

dense_58 (Dense) (None, 8, 1) 17  

-----  

Total params: 321  

Trainable params: 321  

Non-trainable params: 0  

-----  

Epoch 1/5  

10000/10000 [=====] - 22s 2ms/step - loss: 0.2499 - accuracy: 0.5115  

Epoch 2/5  

10000/10000 [=====] - 23s 2ms/step - loss: 0.2481 - accuracy: 0.5407  

Epoch 3/5  

10000/10000 [=====] - 21s 2ms/step - loss: 0.2427 - accuracy: 0.5995  

Epoch 4/5  

10000/10000 [=====] - 24s 2ms/step - loss: 0.2005 - accuracy: 0.7522  

Epoch 5/5  

10000/10000 [=====] - 22s 2ms/step - loss: 0.0585 - accuracy: 0.9695  

test loss: 0.0164222489833012  

test accuracy: 1.0

```

RNN の出力活性化関数を sigmoid に変更すると、精度は落ちた。

jupyter 4_3_keras_codes Last Checkpoint: 14分前 (autosaved)

```
# 亂数種別固定
np.random.seed(12345)

# データ読み込み
x_int = np.array([a_int[i], b_int[i]]).T
x_bin = np.array(x_int)
x_int_test = np.array(x_int[10000:])
x_bin_test = np.array(x_int[10001:20000])

# 正解データ
d_int = a_int + b_int
d_bin = binary[d_int][0:10000]
d_bin_test = binary[d_int][10001:20000]

model = Sequential()

model.add(SimpleRNN(units=16,
                     return_sequences=True,
                     input_shape=[8, 2],
                     go_backwards=False,
                     activation='tanh', # 'sigmoid', #'relu',
                     dropout=0.5,
                     recurrent_dropout=0.3,
                     unroll=False,
                     ))

# 出力層
model.add(Dense(1, activation='sigmoid', input_shape=(-1,2)))
model.summary()
model.compile(loss='mean_squared_error', optimizer=SGD(lr=0.1), metrics=['accuracy'])
# model.compile(loss='mse', optimizer='adam', metrics=['accuracy'])

history = model.fit(x_bin, d_bin.reshape(-1, 8, 1), epochs=5, batch_size=2)

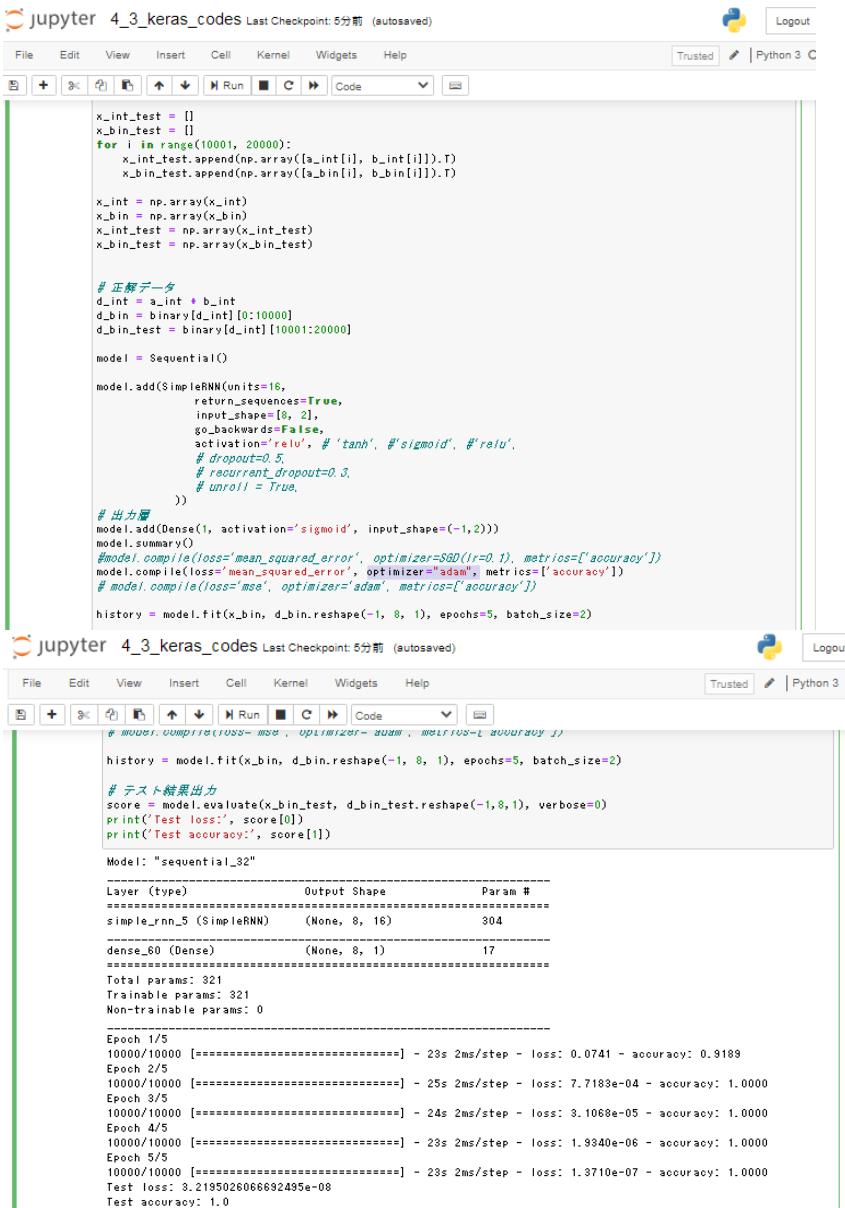
# テスト結果出力
score = model.evaluate(x_bin_test, d_bin_test.reshape(-1,8,1), verbose=0)
```

jupyter 4_3_keras_codes Last Checkpoint: 14分前 (autosaved)

```
# テスト結果出力
score = model.evaluate(x_bin_test, d_bin_test.reshape(-1,8,1), verbose=0)
print('Test loss:', score[0])
print('Test accuracy:', score[1])

Model: "sequential_1"
-----  
Layer (type)          Output Shape         Param #
-----  
simple_rnn_4 (SimpleRNN)    (None, 8, 16)      304  
dense_5_1 (Dense)        (None, 8, 1)       17  
-----  
Total params: 321  
Trainable params: 321  
Non-trainable params: 0  
-----  
Epoch 1/5  
10000/10000 [=====] - 23s 2ms/step - loss: 0.1336 - accuracy: 0.7997  
Epoch 2/5  
10000/10000 [=====] - 25s 2ms/step - loss: 0.0028 - accuracy: 1.0000  
Epoch 3/5  
10000/10000 [=====] - 21s 2ms/step - loss: 7.9449e-04 - accuracy: 1.0000  
Epoch 4/5  
10000/10000 [=====] - 22s 2ms/step - loss: 4.6850e-04 - accuracy: 1.0000  
Epoch 5/5  
10000/10000 [=====] - 23s 2ms/step - loss: 3.2707e-04 - accuracy: 1.0000  
Test loss: 0.0002812037072197074  
Test accuracy: 1.0
```

RNN の出力活性化関数を tanh に変更すると、最初の Relu に近い精度となった。



```

x_int_test = []
x_bin_test = []
for i in range(10001, 20000):
    x_int_test.append(np.array([a_int[i], b_int[i]]).T)
    x_bin_test.append(np.array([a_bin[i], b_bin[i]]).T)

x_int = np.array(x_int)
x_bin = np.array(x_bin)
x_int_test = np.array(x_int_test)
x_bin_test = np.array(x_bin_test)

# 正解データ
d_int = a_int * b_int
d_bin = binary[d_int][0:10000]
d_bin_test = binary[d_int][10001:20000]

model = Sequential()
model.add(SimpleRNN(units=16,
                     return_sequences=True,
                     input_shape=[8, 2],
                     go_backwards=False,
                     activation='relu', # 'tanh', #'sigmoid', #'relu',
                     #dropout=0.5,
                     #recurrent_dropout=0.3,
                     #unroll = True,
                     ))
# 出力層
model.add(Dense(1, activation='sigmoid', input_shape=(-1, 2)))
model.summary()
#model.compile(loss='mean_squared_error', optimizer=SGD(lr=0.1), metrics=['accuracy'])
#model.compile(loss='mean_squared_error', optimizer='adam', metrics=['accuracy'])
#model.compile(loss='mse', optimizer='adam', metrics=['accuracy'])

history = model.fit(x_bin, d_bin.reshape(-1, 8, 1), epochs=5, batch_size=2)

```



```

# モデルを確認する - 損失と精度が表示される
history = model.fit(x_bin, d_bin.reshape(-1, 8, 1), epochs=5, batch_size=2)

# テスト結果出力
score = model.evaluate(x_bin_test, d_bin_test.reshape(-1, 8, 1), verbose=0)
print('Test loss:', score[0])
print('Test accuracy:', score[1])

Model: "sequential_32"
-----  

Layer (type)          Output Shape         Param #
-----  

simple_rnn_5 (SimpleRNN)   (None, 8, 16)      304  

-----  

dense_30 (Dense)        (None, 8, 1)       17  

-----  

Total params: 321
Trainable params: 321
Non-trainable params: 0
-----  

Epoch 1/5
10000/10000 [=====] - 28s 2ms/step - loss: 0.0741 - accuracy: 0.9189
Epoch 2/5
10000/10000 [=====] - 25s 2ms/step - loss: 7.7183e-04 - accuracy: 1.0000
Epoch 3/5
10000/10000 [=====] - 24s 2ms/step - loss: 3.1068e-05 - accuracy: 1.0000
Epoch 4/5
10000/10000 [=====] - 23s 2ms/step - loss: 1.9340e-06 - accuracy: 1.0000
Epoch 5/5
10000/10000 [=====] - 23s 2ms/step - loss: 1.3710e-07 - accuracy: 1.0000
Test loss: 3.2195026066692495e-08
Test accuracy: 1.0

```

最適化方法を adam に変更すると、最初のものよりは精度が上がった。

jupyter 4_3_keras_codes Last Checkpoint: 4分前 (unsaved changes)

File Edit View Insert Cell Kernel Widgets Help Trusted Python 3

```
b_int = np.random.randint(0, size=20000)
b_bin = binary[b_int] # binary encoding

x_int = []
x_bin = []
for i in range(10000):
    x_int.append(np.array([a_int[i], b_int[i]]).T)
    x_bin.append(np.array([a_bin[i], b_bin[i]]).T)

x_int_test = []
x_bin_test = []
for i in range(10001, 20000):
    x_int_test.append(np.array([a_int[i], b_int[i]]).T)
    x_bin_test.append(np.array([a_bin[i], b_bin[i]]).T)

x_int = np.array(x_int)
x_bin = np.array(x_bin)
x_int_test = np.array(x_int_test)
x_bin_test = np.array(x_bin_test)

# 正解データ
d_int = a_int * b_int
d_bin = binary[d_int][0:10000]
d_bin_test = binary[d_int][10001:20000]

model = Sequential()

model.add(SimpleRNN(units=16,
                    return_sequences=True,
                    input_shape=[8, 2],
                    go_backwards=False,
                    activation='relu', # 'tanh', #sigmoid', #relu',
                    dropout=0.5,
                    recurrent_dropout=0.3,
                    # unroll = True,
                    ))
# 出力層
```

jupyter 4_3_keras_codes Last Checkpoint: 4分前 (unsaved changes)

File Edit View Insert Cell Kernel Widgets Help Trusted Python 3

```
print('Test accuracy:', score[1])
```

Model: "sequential_38"

Layer (type)	Output Shape	Param #
simple_rnn_6 (SimpleRNN)	(None, 8, 16)	304
dense_81 (Dense)	(None, 8, 1)	17

Total params: 321
Trainable params: 321
Non-trainable params: 0

Epoch 1/5
10000/10000 [=====] - 8s 787us/step - loss: 0.2449 - accuracy: 0.5527
Epoch 2/5
10000/10000 [=====] - 8s 850us/step - loss: 0.2390 - accuracy: 0.5690
Epoch 3/5
10000/10000 [=====] - 10s 1ms/step - loss: 0.2309 - accuracy: 0.5936
Epoch 4/5
10000/10000 [=====] - 13s 1ms/step - loss: 0.2225 - accuracy: 0.6075
Epoch 5/5
10000/10000 [=====] - 16s 2ms/step - loss: 0.2200 - accuracy: 0.6055
Test loss: 0.2483250266629714
Test accuracy: 0.5751824975013738

RNN の入力 Dropout を 0.5 に設定、
RNN の再帰 Dropout を 0.3 に設定、
収束のスピードは落ち、5 回では精度が低かった。

Keras のドキュメントで unroll の仕様を確認した。[3]

```

x_int_test = []
x_bin_test = []
for i in range(1000, 20000):
    x_int.append(np.array([a_int[i], b_int[i]]).T)
    x_bin.append(np.array([a_bin[i], b_bin[i]]).T)

x_int = np.array(x_int)
x_bin = np.array(x_bin)
x_int_test = np.array(x_int_test)
x_bin_test = np.array(x_bin_test)

# 正解データ
d_int = a_int + b_int
d_bin = binary[d_int][0:10000]
d_bin_test = binary[d_int][10001:20000]

model = Sequential()

model.add(SimpleRNN(units=16,
                    return_sequences=True,
                    input_shape=[8, 2],
                    go_backwards=False,
                    activation='relu', # 'tanh', #'sigmoid', #'relu',
                    #dropout=0.5,
                    #recurrent_dropout=0.3,
                    unroll=True,
                    ))
# 出力層
model.add(Dense(1, activation='sigmoid', input_shape=(-1,2)))
model.summary()
model.compile(loss='mean_squared_error', optimizer=SGD(lr=0.1), metrics=['accuracy'])
#model.compile(loss='mean_squared_error', optimizer='adam', metrics=['accuracy'])
#model.compile(loss='mse', optimizer='adam', metrics=['accuracy'])

history = model.fit(x_bin, d_bin.reshape(-1, 8, 1), epochs=5, batch_size=2)

```

```

Model: "sequential_34"
Layer (Type)          Output Shape         Param #
=====
simple_rnn_7 (SimpleRNN) (None, 8, 16)      304
dense_62 (Dense)      (None, 8, 1)        17
=====
Total params: 321
Trainable params: 321
Non-trainable params: 0

```

```

Epoch 1/5
10000/10000 [=====] - 18s 2ms/step - loss: 0.1205 - accuracy: 0.8880
Epoch 2/5
10000/10000 [=====] - 18s 2ms/step - loss: 0.0133 - accuracy: 0.9984
Epoch 3/5
10000/10000 [=====] - 18s 2ms/step - loss: 0.0015 - accuracy: 1.0000
Epoch 4/5
10000/10000 [=====] - 18s 2ms/step - loss: 7.4531e-04 - accuracy: 1.0000
Epoch 5/5
10000/10000 [=====] - 18s 2ms/step - loss: 4.8124e-04 - accuracy: 1.0000
Test loss: 0.0004051784593491337
Test accuracy: 1.0

```

RNN の unroll を True に設定すると、
学習のスピードが少しあがり、精度は少し落ちた。

3.2.強化学習

3.2.1 強化学習とは

長期的に報酬を最大化できるように、
環境のなかで行動を選択できるエージェントを作ることを目標とする機械学習の一分野。
行動の結果として与えられる利益（報酬）をもとに、
エージェントが行動を決定する原理を改善していく仕組み。

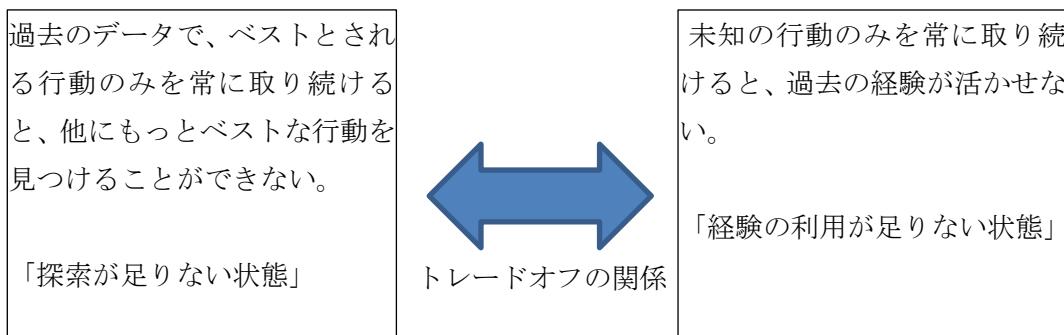
3.2.2 強化学習の応用例

マーケティングの例

環境	会社の販売促進部
エージェント	プロフィールと購入履歴に基づいて、キャンペーンメールを送る顧客を決めるソフトウェア
行動	顧客ごとに送信、非送信の二つの行動を選ぶ
報酬	負の報酬：キャンペーンのコスト 正の報酬：キャンペーンで生み出されると推測される売上

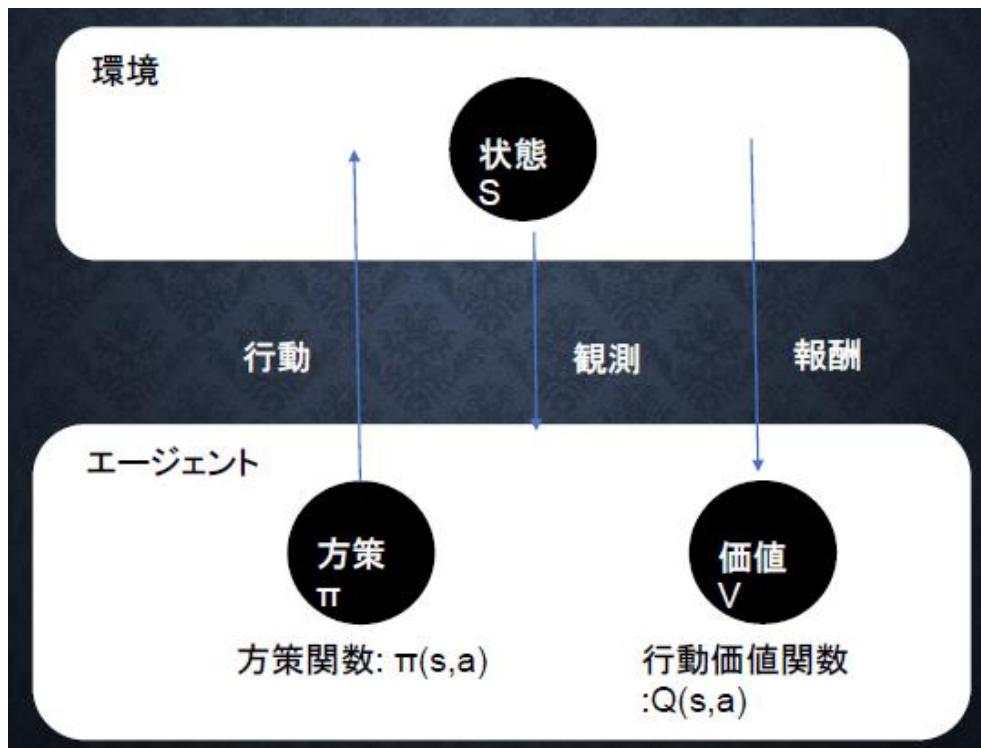
3.2.3 探索と利用のトレードオフ

強化学習では、環境についての不完全な知識を元に、行動しながらデータを収集。
最適な行動を見つけていく。



この2つの探索と利用のトレードオフを調整することが重要。

3.2.4 強化学習のイメージ



3.2.5 強化学習の差分

今までの学習とは目標が異なる。

教師なし、あり学習	データに含まれるパターンを見つけ出す、および、そのデータから予測することが目標
強化学習	優れた方策を見つけることが目標

- ・計算速度の向上
- ・Q 学習：行動価値関数を行動する毎に更新することにより学習を進める方法
- ・関数近似法：行動価値関数や方策関数を関数近似する方法
- ・Q 学習と関数近似法の 2 つを組み合わせる。

3.2.6 行動価値関数

状態価値関数と行動価値関数の 2 種類の価値を表す関数がある。

- ・状態価値関数：ある状態の価値に注目する場合
- ・行動価値関数：状態と価値を組み合わせた価値に注目する場合

3.2.7 方策関数

- ・方策関数：ある状態でどのような行動を探るのかの確率を与える関数

3.2.8 方策勾配法

再帰的に方向を更新していくモデル

$$\theta^{(t+1)} = \theta^{(t)} + \epsilon \nabla J(\theta)$$

t+1回目においてはt回目のもとの学習率 ϵ で定義した方策との加算で次の値を構成していく。

Jは方策の良さを表す。Jの定義が必要。

Jの定義の方法として2つ。

・平均報酬：行動を取った時に生まれる価値の全部の平均。均一に見る。

・割引報酬和：カッコになればなるほどその報酬の加算の割合を減らす。

減衰を使った手法。

一番近いやつの報酬を1番重みづけする。

この2つに対応して、行動価値関数 $Q(s,a)$ を定義。

方策勾配定理が成り立つ。 Q 関数を使って定義できる。

$$\nabla_{\theta} J(\theta) = \mathbb{E}_{\pi_{\theta}}[(\nabla_{\theta} \log \pi_{\theta}(a|s) Q^{\pi}(s,a))]$$

3.2.9 例題解説 1

問：以下に当てはまるものを記号で答えよ。

強化学習の基本的なアルゴリズムのうちの一つとして、ベルマン方程式に従って行動価値関数 $Q(s,a)$ をモデル化する価値反復に基づく方法が挙げられる。しかしこの方法では行動 a が連続な値を取る場合のような場合を自然に扱えないといった欠点がある。

そうした問題に対応する方法として、方策自体をパラメータ θ を用いて $\pi_{\theta}(a|s)$ のようにモデル化した方策勾配に基づくアルゴリズムが挙げられる。方策勾配に基づくアルゴリズムにおいては方策勾配の定理により目的関数 $J(\theta) = V(s_0)$ の θ に関する勾配 $\nabla_{\theta} J(\theta)$ を以下のように計算できる。

$$\nabla_{\theta} J(\theta) = \mathbb{E}_{\pi_{\theta}}[(\text{?})]$$

$$(a) \nabla_{\theta} \log \pi_{\theta}(a|s) Q^{\pi}(s,a) \quad (b) \nabla_{\theta} \pi_{\theta}(a|s) Q^{\pi}(s,a) \quad (c) \frac{\nabla_{\theta} \pi_{\theta}(a|s)}{Q^{\pi}(s,a)} \quad (d) \frac{Q^{\pi}(s,a)}{\nabla_{\theta} \pi_{\theta}(a|s)}$$

こうして計算された勾配を用いて勾配法により、方策 $\pi_{\theta}(a|s)$ をアップデートするのが方策勾配に基づくアルゴリズムである。

2018年07月24 JDIA提供

前項の方策勾配定理がそのまま入るので(a)が正しい。

方策勾配定理は下記の2式から導出される。

- ・状態価値関数 $v(s) = \sum_a (\pi(a|s)Q(s,a))$
- ・ベルマン方程式 $Q(s,a) = \sum_{s'} (P(s'|s,a)[r(s,a,s') + \gamma V(s')])$

3.2.10 DCGAN

1511.06434.pdf 1 / 16 Under review as a conference paper at ICLR 2016

**UNSUPERVISED REPRESENTATION LEARNING
WITH DEEP CONVOLUTIONAL
GENERATIVE ADVERSARIAL NETWORKS**

Alec Radford & Luke Metz
indigo Research
Boston, MA
`{alec,luke}@indigo.io`

Soumith Chintala
Facebook AI Research
New York, NY
`soumith@fb.com`

ABSTRACT

In recent years, supervised learning with convolutional networks (CNNs) has seen huge adoption in computer vision applications. Comparatively, unsupervised learning with CNNs has received less attention. In this work we hope to help bridge the gap between the success of CNNs for supervised learning and unsupervised learning. We introduce a class of CNNs called deep convolutional generative adversarial networks (DCGANs), that have certain architectural constraints, and demonstrate that they are a strong candidate for unsupervised learning. Training on various image datasets, we show convincing evidence that our deep convolutional adversarial pair learns a hierarchy of representations from object parts to scenes in both the generator and discriminator. Additionally, we use the learned features for novel tasks - demonstrating their applicability as general image representations.

.06434v2 [cs.LG] 7 Jan 2016

Arxiv のサイトで、DCGAN の論文を確認した。[4]

DCGAN (DEEP CONVOLUTIONAL GENERATIVE ADVERSARIAL NETWORKS)

CNN のディープなネットワークを使い敵対的学习を行って画像を生成する。

加算減算で、数式的に画像を扱うことができる。

Generator はランダムな 100 個の正規分布値を使い画像を生成。

生成した画像と実際の画像を比較して Discriminator がその画像が作られたものかを判定する。

Discrimnator は正確に判定できるように重みを更新。

Generator は Discrimnator が正確に判定できないような難しい画像を作ることを目標に重みを更新する。

画像のパターンにノイズが多いと、生成がうまくいかない。

3.2.11 例題解説 2

Gan は、生成器(Generator)および判別器(Discriminator)という2つの学習するモデルからなる。Generatorはノイズ x を入力として、新しいデータ $G(x)$ を生成・出力する。Discriminatorは与えられたデータについて、それが訓練データであるか、Generatorが生成したデータ $G(x)$ であるかを識別する。例えば、出力 $D(x)$ は入力データが訓練データである確率(スカラー値)を意味する。

両方のモデルを交互に学習させることで、Generatorは限りなく訓練データに近い分布のデータを生成し、Discriminatorはそのようなデータに対しての識別の確率を高めるようにさせる。

学習が進み、GeneratorおよびDiscriminatorもれぞれが理想的な性能を有するようになった場合、Generatorが生成するデータ分布 p_g と訓練データの分布 p_{data} が一致するため、 $D(x)$ は (a) となる。

(a) の選択肢：

- (a) 0.5
- (b) 0
- (c) 1
- (d) 0.25

2018年7月JDLA問題

学習が進んで Generator および Discriminator のそれぞれが理想的な性能を有するようになった場合、Generator の生成する画像分布と訓練データの分布 P が一致するため、この出力 D(x) は入力データと訓練データの確率が 1:1、つまり 0.5 となる。(a)が正しい。

Gan の学習において、Discriminatorは訓練データ x 、生成データ $G(x)$ それぞれに対して訓練データ x より生成データであると正しく判定する確率を最大化しようとする。
 Generatorは、Discriminatorが生成データを訓練データであると誤判定する確率を最大化しようとする。
 以上の内容は、価値関数 $V(D, G)$ を用いた以下の式で表現される。

(う)

(う) の選択肢：

- (a) $\min_G \max_D V(D, G) = \mathbb{E}_{x \sim p_{data}(x)}[\log D(x)] + \mathbb{E}_{z \sim p_g(z)}[\log(1 - D(G(z)))]$
- (b) $\min_G \max_D V(D, G) = \mathbb{E}_{x \sim p_{data}(x)}[\log D(x)] + \mathbb{E}_{z \sim p_g(z)}[\log(D(G(z)))]$
- (c) $\max_G \min_D V(D, G) = \mathbb{E}_{x \sim p_{data}(x)}[\log D(x)] + \mathbb{E}_{z \sim p_g(z)}[\log(1 - D(G(z)))]$
- (d) $\max_G \min_D V(D, G) = \mathbb{E}_{x \sim p_{data}(x)}[\log D(x)] + \mathbb{E}_{z \sim p_g(z)}[\log(D(G(z)))]$

2018年7月JDLA問題

価値関数 $V(D, G)$ の Discriminator は正しく判定する確率を最大化するので、 $D(x)$ が大きいほど右辺第一項が大きくなるので D は最大化する。一方、Generator は誤判定の確率を最大化するので、右辺第 2 項の G は最小化していくため、 $\min_G \max_D V(D, G)$ の式になる。(a)が正しい。

近年では、Gan の手法をもとに、そこに畳み込みニューラルネットワーク(CNN)を適用し、さらにブーリング層を畳み込みで置き換えていく(Generatorにfractionally-strided convolution、Discriminatorにstrided-convolutionを適用)。また活性化関数として、Generatorの各層にReLU(出力層のみtanh)、Discriminatorの全層に-(ϵ)で定義されるLeaky ReLUを適用する。

$$(a) \quad f = \begin{cases} x & (x > 0) \\ 0.01x & (x \leq 0) \end{cases}$$

$$(b) \quad f = \begin{cases} x & (x > 0) \\ e^x - 1 & (x \leq 0) \end{cases}$$

$$(c) \quad f = \begin{cases} x & (x > 0) \\ ax & (x \leq 0) \end{cases}$$

$$(d) \quad f = \log(1 + e^x)$$

以上のようにして(a)にCNNを適応させ、学習がうまく成立するベストプラクティスについて提案したものをDeep Convolutional Generative Adversarial Networkという。

2016年7月JDLA提供

(a)が Leaky ReLU の定義。

PReLU の論文。ReLU と Leaky ReLU の後に PReLU が出た。

0.01 を用いるのが Leaky ReLU。

arxiv.org/pdf/1502.01852.pdf
1502.01852.pdf 1 / 11

Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification

Kaiming He Xiangyu Zhang Shaoqing Ren Jian Sun
Microsoft Research
`{kahe, v-xiangz, v-shren, jiansun}@microsoft.com`

Abstract

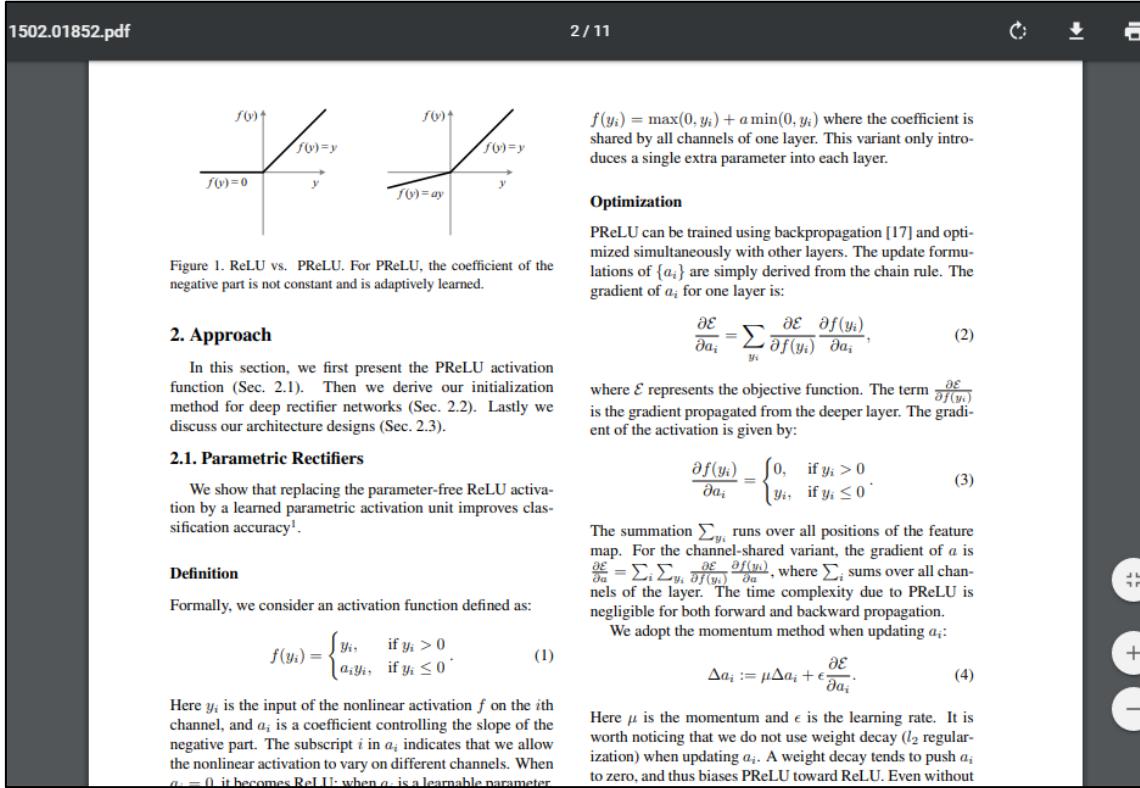
Rectified activation units (rectifiers) are essential for state-of-the-art neural networks. In this work, we study rectifier neural networks for image classification from two aspects. First, we propose a Parametric Rectified Linear Unit (PReLU) that generalizes the traditional rectified unit. PReLU improves model fitting with nearly zero extra computational cost and little overfitting risk. Second, we derive a robust initialization method that particularly considers the rectifier nonlinearities. This method enables us to train extremely deep rectified models directly from scratch and to investigate deeper or wider network architectures. Based on our PReLU networks (PReLU-nets), we achieve **4.94%** top-5 test error on the ImageNet 2012 classification dataset. This is a 26% relative improvement over the ILSVRC 2014 winner (GoogLeNet, 6.66% [29]). To our knowledge, our result is the first to surpass human-level performance (5.1%, [22]) on this visual recognition challenge.

and the use of smaller strides [33, 24, 2, 25]), new nonlinear activations [21, 20, 34, 19, 27, 9], and sophisticated layer designs [29, 11]. On the other hand, better generalization is achieved by effective regularization techniques [12, 26, 9, 31], aggressive data augmentation [16, 13, 25, 29], and large-scale data [4, 22].

Among these advances, the rectifier neuron [21, 8, 20, 34], e.g., Rectified Linear Unit (ReLU), is one of several keys to the recent success of deep networks [16]. It expedites convergence of the training procedure [16] and leads to better solutions [21, 8, 20, 34] than conventional sigmoid-like units. Despite the prevalence of rectifier networks, recent improvements of models [33, 24, 11, 25, 29] and theoretical guidelines for training them [7, 23] have rarely focused on the properties of the rectifiers.

In this paper, we investigate neural networks from two aspects particularly driven by the rectifiers. First, we propose a new generalization of ReLU, which we call *Parametric Rectified Linear Unit* (PReLU). This activation function adaptively learns the parameters of the rectifiers, and improves accuracy at negligible extra computational cost. Second, we study the difficulty of training rectified models that are very deep. By explicitly modeling the non-

Xiv:1502.01852v1 [cs.CV] 6 Feb 2015



Arxiv にて、PReLU の論文を確認した。[5]

3.3. 参考文献

- [1] The Iris Dataset, scikit-learn document, 閲覧日 2020-07-31,
https://scikit-learn.org/stable/auto_examples/datasets/plot_iris_dataset.html
- [2] The CIFAR-10 dataset, Alex Krizhevsky's home page, 閲覧日 2020-08-01,
<https://www.cs.toronto.edu/~kriz/cifar.html>
- [3] Recurrent レイヤー RNN, Keras Documentation, 閲覧日 2020-08-01,
<https://keras.io/ja/layers/recurrent/>
- [4] Alec Radford & Luke Metz, Soumith Chintala, UNSUPERVISED REPRESENTATION LEARNING WITH DEEP CONVOLUTIONAL GENERATIVE ADVERSARIAL NETWORKS, Under review as a conference paper at ICLR 2016, 2016, <https://arxiv.org/pdf/1511.06434.pdf>
- [5] Kaiming He, Xiangyu Zhang, Shaoqing Ren, Jian Sun, Microsoft Research, Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification, 2015, <https://arxiv.org/pdf/1502.01852.pdf>

- [6] Kyunghyun Cho, Bart van Merriënboer, Caglar Gulcehre, Dzmitry Bahdanau, Fethi Bougares, Holger Schwenk, Yoshua Bengio, Learning Phrase Representations using RNN Encoder-Decoder for Statistical Machine Translation, 2014,
<https://arxiv.org/pdf/1406.1078.pdf>
- [7] Ilya Sutskever, Oriol Vinyals, Quoc V. Le, Sequence to Sequence Learning with Neural Networks, NIPS, 2014,
<https://papers.nips.cc/paper/5346-sequence-to-sequence-learning-with-neural-networks.pdf>
- [8] Tomas Mikolov, Kai Chen, Greg Corrado, Jeffrey Dean, Efficient Estimation of Word Representations in Vector Space, 2013, <https://arxiv.org/pdf/1301.3781.pdf>
- [9] IMAGENET Large Scale Visual Recognition Challenge (ILSVRC) 2017 Overview, image-net, http://image-net.org/challenges/talks_2017/ILSVRC2017_overview.pdf
- [10] M. D. Zeiler and R. Fergus. Visualizing and understanding convolutional networks. In Proc. of ECCV, 2014, <https://arxiv.org/pdf/1311.2901.pdf>
- [11] C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, and A. Rabinovich. Going deeper with convolutions. In Proc. of CVPR, 2015, <https://arxiv.org/pdf/1409.4842.pdf>
- [12] K. He, X. Zhang, S. Ren, and J. Sun. Deep residual learning for image recognition. In Proc. of CVPR, 2016, <https://arxiv.org/pdf/1512.03385.pdf>

以上

