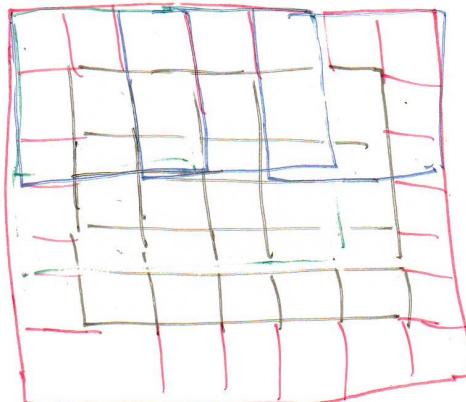


▼ 深層学習Day3

深層学習 Day3.

石窓認テスト 1.

サイズ 5×5 の入力画像を 3×3 のフィルターで畳み入力時の出力画像のサイズは? (Stride:2, パディング:1)



左: 3回

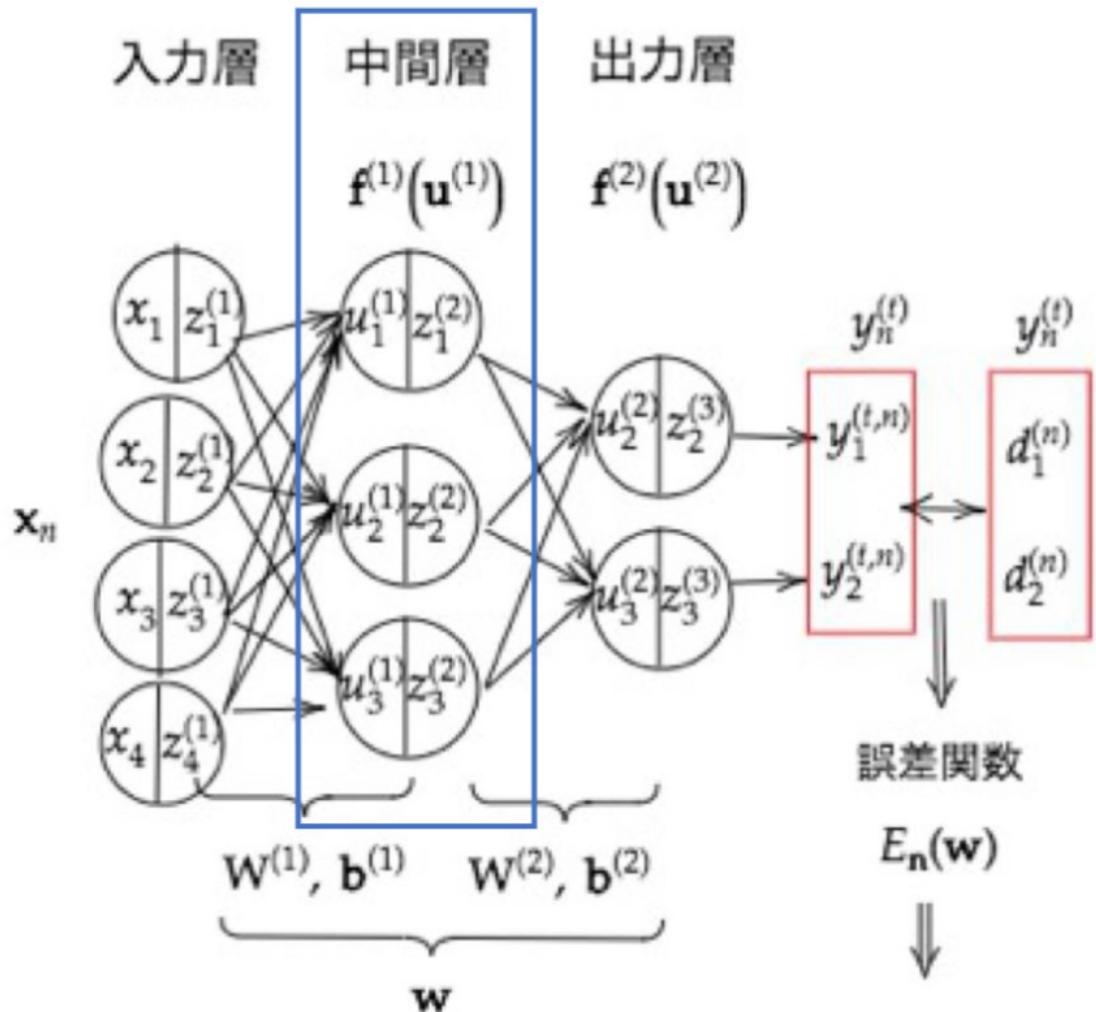
右: 3回

3×3
11

▼ 再帰型ニューラルネットワークについて

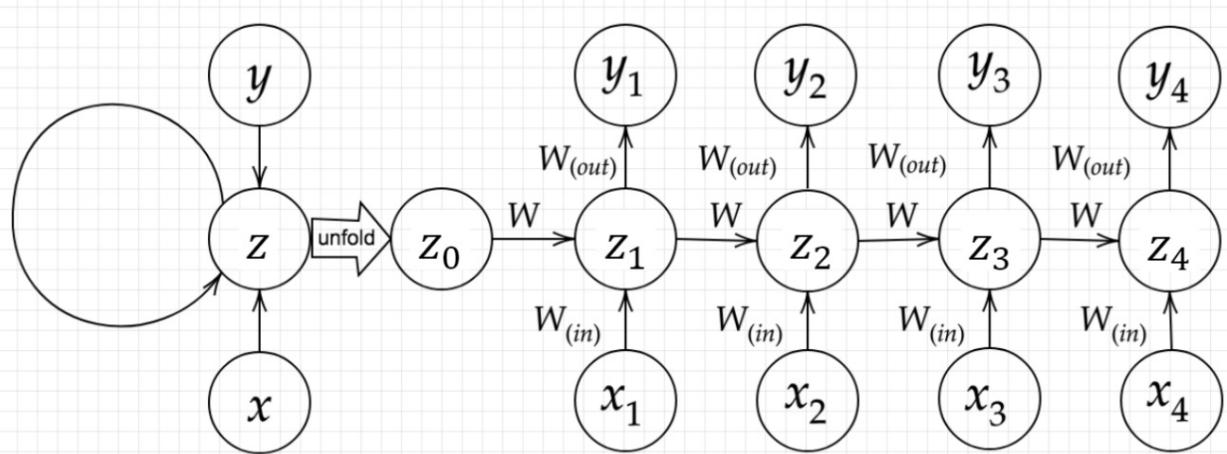
Section1: 再帰型ニューラルネットワークの概念

- 1-1 RNN全体像
 - 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
 - 1-1-2 時系列データ
 - 時間的つながりがあるデータ
例) 株価、音声データ
 - 1-1-3 RNNについて



$$\mathbf{w}^{(t+1)} = \mathbf{w}^{(t)} - \varepsilon \nabla E_n(\mathbf{w}) \Leftarrow \quad \nabla E_n(\mathbf{w}) = \frac{\partial E}{\partial \mathbf{w}}$$

- 基本的にRNNも、入力層、中間層、出力層という構造は同じ
- 一定の時間的つながりがあるデータを学習するのがRNNの目的



- RNNでは中間層に工夫がある
- 右側と左側は同じものを模試帰化したもの

- 左側

$x \Rightarrow z \Rightarrow y$ に値が行く
 $\Rightarrow z \Rightarrow y$ に値が行く (ちょっとわかりにくいので)

- 右側

ループ部分を分けた図

- 式

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

- コード

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

数式S1)1-3Python日本語

- 確認テスト 2

RNNのネットワークには大きくわけて3つの重みがある。

1つは入力から現在の中間層を定義する際にかけられる重み、

1つは中間層から出力を定義する際にかけられる重みである。

残り1つの重みについて説明せよ。

A. 中間層から中間層の重み

- 実装演習

- ▼ 準備

- ▼ Google ドライブのマウント

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

Mounted at /content/drive
```

- ▼ sys.pathの設定

以下では、Google ドライブのマイドライブ直下にDNN_code フォルダを置くことを仮定しています。必要に応じて、パスを変更してください。

```
import sys
sys.path.append('/content/drive/My Drive/')
```

- ▼ simple RNN

- バイナリ加算

```
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 = 2000 # 出力が多いので調整

    # 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

    # He

    # 勾配
    W_in_grad = np.zeros_like(W_in)
    W_out_grad = np.zeros_like(W_out)
    W_grad = np.zeros_like(W)

    u = np.zeros((hidden_layer_size, binary_dim + 1))
    z = np.zeros((hidden_layer_size, binary_dim + 1))
    y = np.zeros((output_layer_size, binary_dim))

    delta_out = np.zeros((output_layer_size, binary_dim))
    delta = np.zeros((hidden_layer_size, binary_dim + 1))

    all_losses = []
```

```

for i in range(iters_num):

    # A, B初期化 (a + b = d)
    a_int = np.random.randint(largest_number/2)
    a_bin = binary[a_int] # binary encoding
    b_int = np.random.randint(largest_number/2)
    b_bin = binary[b_int] # binary encoding

    # 正解データ
    d_int = a_int + b_int
    d_bin = binary[d_int]

    # 出力バイナリ
    out_bin = np.zeros_like(d_bin)

    # 時系列全体の誤差
    all_loss = 0

    # 時系列ループ
    for t in range(binary_dim):
        # 入力値
        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])

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

        # 勾配更新
        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))

        # 勾配適用
        W_in -= learning_rate * W_in_grad
        W_out -= learning_rate * W_out_grad
        W -= learning_rate * W_grad

        W_in_grad *= 0
        W_out_grad *= 0
        W_grad *= 0

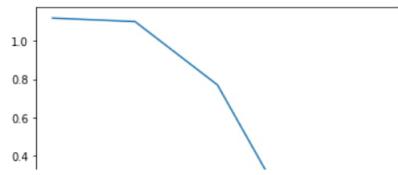
    if(i % plot_interval == 0):
        all_losses.append(all_loss)
        print("iters:" + str(i))
        print("Loss:" + str(all_loss))
        print("Pred:" + str(out_bin))
        print("True:" + str(d_bin))
        out_int = 0
        for index, x in enumerate(reversed(out_bin)):
            out_int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

```

```

iters:0
Loss:1.1176885529277911
Pred:[1 1 1 1 1 1 1]
True:[0 1 0 1 0 1 1 1]
61 + 26 = 255
-----
iters:2000
Loss:1.0994668722760093
Pred:[0 1 0 0 1 1 1 1]
True:[1 0 0 0 1 0 1 0]
39 + 99 = 79
-----
iters:4000
Loss:0.7695061993257646
Pred:[1 1 0 1 0 1 1]
True:[1 0 0 0 1 0 1 1]
109 + 30 = 235
-----
iters:6000
Loss:0.007937800763854476
Pred:[0 1 0 1 1 1 1 1]
True:[0 1 0 1 1 1 1 1]
0 + 95 = 95
-----
iters:8000
Loss:0.0020261104184226522
Pred:[0 1 1 0 1 1 0 0]
True:[0 1 1 0 1 1 0 0]
9 + 99 = 108
-----
```



▼ [try] weight_init_stdやlearning_rate, hidden_layer_sizeを変更してみよう

```

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 = 18
output_layer_size = 1

weight_init_std = 2
learning_rate = 0.3

iters_num = 10000
plot_interval = 2000 # 出力が多いので調整

# 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

# ハードミクシング

# 勾配
W_in_grad = np.zeros_like(W_in)
W_out_grad = np.zeros_like(W_out)
W_grad = np.zeros_like(W)

u = np.zeros((hidden_layer_size, binary_dim + 1))
```

```

z = np.zeros((hidden_layer_size, binary_dim + 1))
y = np.zeros((output_layer_size, binary_dim))

delta_out = np.zeros((output_layer_size, binary_dim))
delta = np.zeros((hidden_layer_size, binary_dim + 1))

all_losses = []

for i in range(iters_num):

    # A, B初期化 (a + b = d)
    a_int = np.random.randint(largest_number/2)
    a_bin = binary[a_int] # binary encoding
    b_int = np.random.randint(largest_number/2)
    b_bin = binary[b_int] # binary encoding

    # 正解データ
    d_int = a_int + b_int
    d_bin = binary[d_int]

    # 出力バイナリ
    out_bin = np.zeros_like(d_bin)

    # 時系列全体の誤差
    all_loss = 0

    # 時系列ループ
    for t in range(binary_dim):
        # 入力値
        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])

        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], b_bin[-t]]).reshape(1, -1)

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

        # 勾配更新
        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))

    # 勾配適用
    W_in -= learning_rate * W_in_grad
    W_out -= learning_rate * W_out_grad
    W -= learning_rate * W_grad

    W_in_grad *= 0
    W_out_grad *= 0
    W_grad *= 0

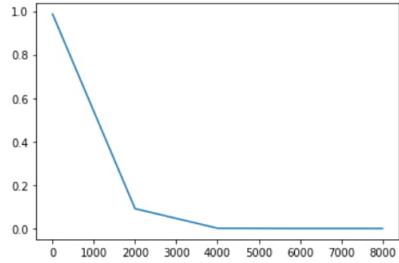
if(i % plot_interval == 0):
    all_losses.append(all_loss)
    print("iters:" + str(i))
    print("Loss:" + str(all_loss))
    print("Pred:" + str(out_bin))
    print("True:" + str(d_bin))
    out_int = 0
    for index,x in enumerate(reversed(out_bin)):
        out_int += x * pow(2, index)
    print(str(a_int)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

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

```

```

iters:0
Loss:0.9866100633001044
Pred:[1 1 1 1 1 1 1]
True:[1 0 0 1 1 1 1]
97 + 62 = 255
-----
iters:2000
Loss:0.09123474119982379
Pred:[1 1 0 0 0 0 0]
True:[1 1 1 0 0 0 0]
115 + 109 = 224
-----
iters:4000
Loss:0.0011424632125659293
Pred:[1 0 1 1 0 0 0 1]
True:[1 0 1 1 0 0 0 1]
72 + 105 = 177
-----
iters:6000
Loss:0.0001971645364993515
Pred:[1 1 0 1 0 0 1 1]
True:[1 1 0 1 0 0 1 1]
125 + 86 = 211
-----
iters:8000
Loss:0.00013196598927794664
Pred:[0 1 1 0 1 0 0 1]
True:[0 1 1 0 1 0 0 1]
9 + 96 = 105
-----
```



▼ [try] 重みの初期化方法を変更してみよう

Xavier, He

```

# Xavier
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 = 2
learning_rate = 0.3

iters_num = 10000
plot_interval = 2000 # 出力が多いので調整

# 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
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))

# He

# 勾配
W_in_grad = np.zeros_like(W_in)
W_out_grad = np.zeros_like(W_out)
W_grad = np.zeros_like(W)

u = np.zeros((hidden_layer_size, binary_dim + 1))
z = np.zeros((hidden_layer_size, binary_dim + 1))
y = np.zeros((output_layer_size, binary_dim))

delta_out = np.zeros((output_layer_size, binary_dim))
delta = np.zeros((hidden_layer_size, binary_dim + 1))

all_losses = []

for i in range(iters_num):

    # A, B初期化 (a + b = d)
    a_int = np.random.randint(largest_number/2)
    a_bin = binary[a_int] # binary encoding
    b_int = np.random.randint(largest_number/2)
    b_bin = binary[b_int] # binary encoding

    # 正解データ
    d_int = a_int + b_int
    d_bin = binary[d_int]

    # 出力バイナリ
    out_bin = np.zeros_like(d_bin)

    # 時系列全体の誤差
    all_loss = 0

    # 時系列ループ
    for t in range(binary_dim):
        # 入力値
        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])

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

        #誤差
        loss = functions.mean_squared_error(dd, 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] = (np.dot(delta[:, t+1].T, W) + np.dot(delta_out[:, t].T, W_out.T)) * functions.d_sigmoid(u[:, t+1])

        # 勾配更新
        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))

    # 勾配適用
    W_in -= learning_rate * W_in_grad
    W_out -= learning_rate * W_out_grad
    W -= learning_rate * W_grad

    W_in_grad *= 0
    W_out_grad *= 0
    W_grad *= 0

    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()

```

iters:0
Loss:0.933385745102287
Pred:[0 0 0 0 0 0 0]
True:[0 1 1 0 0 0 0]
82 + 14 = 0
-----
```

```

iters:2000
Loss:0.15585993227363298
Pred:[0 1 1 1 1 0 0 0]
True:[0 1 1 1 1 0 0 0]
106 + 14 = 120
-----
```

```

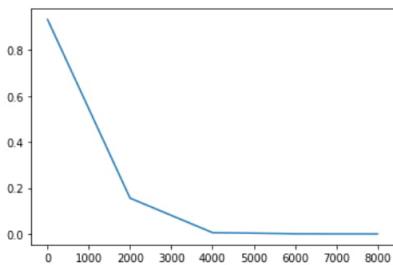
iters:4000
Loss:0.005696474822099283
Pred:[1 0 0 0 1 0 1 0]
True:[1 0 0 0 1 0 1 0]
124 + 14 = 138
-----
```

```

iters:6000
Loss:0.0006623198000603943
Pred:[1 0 1 1 1 0 0 1]
True:[1 0 1 1 1 0 0 1]
107 + 78 = 185
-----
```

```

iters:8000
Loss:0.00014733245101880983
Pred:[0 1 0 0 1 0 1 1]
True:[0 1 0 0 1 0 1 1]
32 + 43 = 75
-----
```



```

# He
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)]).astype(np.uint8)).T, axis=1

input_layer_size = 2
hidden_layer_size = 18
output_layer_size = 1

weight_init_std = 2
learning_rate = 0.3

iters_num = 10000
plot_interval = 2000 # 出力が多いので調整

# 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

# 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)

# 勾配
W_in_grad = np.zeros_like(W_in)
W_out_grad = np.zeros_like(W_out)
W_grad = np.zeros_like(W)

u = np.zeros((hidden_layer_size, binary_dim + 1))
z = np.zeros((hidden_layer_size, binary_dim + 1))
y = np.zeros((output_layer_size, binary_dim))

delta_out = np.zeros((output_layer_size, binary_dim))
delta = np.zeros((hidden_layer_size, binary_dim + 1))

all_losses = []

for i in range(iters_num):

    # A, B初期化 (a + b = d)
    a_int = np.random.randint(largest_number/2)
    a_bin = binary[a_int] # binary encoding
    b_int = np.random.randint(largest_number/2)
    b_bin = binary[b_int] # binary encoding

    # 正解データ
    d_int = a_int + b_int
    d_bin = binary[d_int]

    # 出力バイナリ
    out_bin = np.zeros_like(d_bin)

    # 時系列全体の誤差
    all_loss = 0

    # 時系列ループ
    for t in range(binary_dim):
        # 入力値
        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])

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

        # 勾配更新
        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))

    # 勾配適用
    W_in -= learning_rate * W_in_grad
    W_out -= learning_rate * W_out_grad
    W -= learning_rate * W_grad

    W_in_grad *= 0
    W_out_grad *= 0
    W_grad *= 0

    if(i % plot_interval == 0):
        all_losses.append(all_loss)

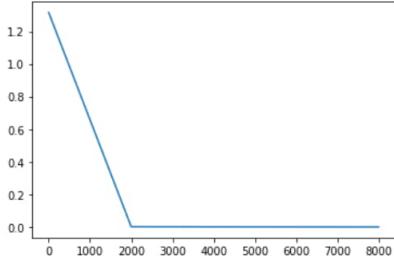
```

```

    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()

iters:0
Loss:1.314785450287343
Pred:[1 1 1 1 1 1 1]
True:[1 0 1 0 0 0 0]
120 + 40 = 255
-----
iters:2000
Loss:0.0015315820796761948
Pred:[0 0 1 0 1 1 0 0]
True:[0 0 1 0 1 1 0 0]
13 + 31 = 44
-----
iters:4000
Loss:0.0008881129278999514
Pred:[1 0 1 0 1 0 1 1]
True:[1 0 1 0 1 0 1 1]
64 + 107 = 171
-----
iters:6000
Loss:0.00035918579582061225
Pred:[1 0 0 0 0 0 1 1]
True:[1 0 0 0 0 0 1 1]
19 + 112 = 131
-----
iters:8000
Loss:1.802037012870606e-05
Pred:[1 0 1 1 1 1 0 0]
True:[1 0 1 1 1 1 0 0]
61 + 127 = 188
-----
```



▼ [try] 中間層の活性化関数を変更してみよう

ReLU(勾配爆発を確認しよう)
tanh(numpyにtanhが用意されている。導関数をd_tanhとして作成しよう)

```

# Xavier
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 = 18
output_layer_size = 1
```

```

output_layer_size = 1

weight_init_std = 2
learning_rate = 0.3

iters_num = 10000
plot_interval = 2000

# 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
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))

# He

# 勾配
W_in_grad = np.zeros_like(W_in)
W_out_grad = np.zeros_like(W_out)
W_grad = np.zeros_like(W)

u = np.zeros((hidden_layer_size, binary_dim + 1))
z = np.zeros((hidden_layer_size, binary_dim + 1))
y = np.zeros((output_layer_size, binary_dim))

delta_out = np.zeros((output_layer_size, binary_dim))
delta = np.zeros((hidden_layer_size, binary_dim + 1))

all_losses = []

for i in range(iters_num):

    # A, B初期化 (a + b = d)
    a_int = np.random.randint(largest_number/2)
    a_bin = binary[a_int] # binary encoding
    b_int = np.random.randint(largest_number/2)
    b_bin = binary[b_int] # binary encoding

    # 正解データ
    d_int = a_int + b_int
    d_bin = binary[d_int]

    # 出力バイナリ
    out_bin = np.zeros_like(d_bin)

    # 時系列全体の誤差
    all_loss = 0

    # 時系列ループ
    for t in range(binary_dim):
        # 入力値
        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])
        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])
        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], b_bin[-t]]).reshape(1, -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])

        # 勾配更新
        W_out_grad += np.dot(z[:, t+1].reshape(-1, 1), delta_out[:, t].reshape(-1, 1))
        ...

```

```

W_grad += np.dot(z1[:, t].reshape(-1, 1), delta[:, t].reshape(1, -1))
W_in_grad += np.dot(X.T, delta[:, t].reshape(1, -1))

# 勾配適用
W_in -= learning_rate * W_in_grad
W_out -= learning_rate * W_out_grad
W -= learning_rate * W_grad

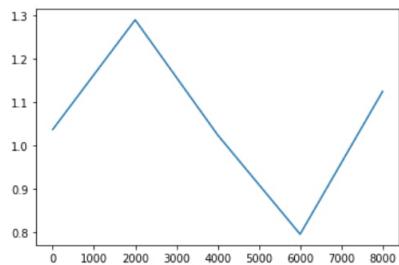
W_in_grad *= 0
W_out_grad *= 0
W_grad *= 0

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()

iters:0
Loss:1.0369688764583775
Pred:[1 1 0 1 1 1 1 1]
True:[0 1 1 1 0 1 1 1]
31 + 88 = 223
-----
iters:2000
Loss:1.289473532846869
Pred:[0 1 1 1 1 1 1 1]
True:[1 0 0 0 0 0 1 0]
108 + 22 = 126
-----
iters:4000
Loss:1.0244744275219426
Pred:[0 0 1 1 1 1 1 1]
True:[0 1 0 1 0 0 1 1]
56 + 27 = 63
-----
iters:6000
Loss:0.7957800641265407
Pred:[0 1 1 1 1 1 1 1]
True:[0 0 1 1 1 1 1 1]
22 + 41 = 63
-----
iters:8000
Loss:1.124515883331014
Pred:[1 1 0 1 1 1 1 1]
True:[0 1 1 1 0 0 0 1]
43 + 70 = 239

```



```
# He
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
# 例 [0, 1, ..., 2^8 - 1]
# 例 [0, 1, ..., 2^8 - 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 = 18
output_layer_size = 1

weight_init_std = 2
learning_rate = 0.3

iters_num = 10000
plot_interval = 2000

# 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
# 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)

# 勾配
W_in_grad = np.zeros_like(W_in)
W_out_grad = np.zeros_like(W_out)
W_grad = np.zeros_like(W)

u = np.zeros((hidden_layer_size, binary_dim + 1))
z = np.zeros((hidden_layer_size, binary_dim + 1))
y = np.zeros((output_layer_size, binary_dim))

delta_out = np.zeros((output_layer_size, binary_dim))
delta = np.zeros((hidden_layer_size, binary_dim + 1))

all_losses = []

for i in range(iters_num):

    # A, B初期化 (a + b = d)
    a_int = np.random.randint(largest_number/2)
    a_bin = binary[a_int] # binary encoding
    b_int = np.random.randint(largest_number/2)
    b_bin = binary[b_int] # binary encoding

    # 正解データ
    d_int = a_int + b_int
    d_bin = binary[d_int]

    # 出力バイナリ
    out_bin = np.zeros_like(d_bin)

    # 時系列全体の誤差
    all_loss = 0

    # 時系列ループ
    for t in range(binary_dim):
        # 入力値
        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] = np.tanh(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])
        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])
        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] = (np.dot(delta[:, t+1].T, W.T) + np.dot(delta_out[:, t].T, W_out.T)) * functions.d_sigmoid(u[:, t+1])

# 勾配更新
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))

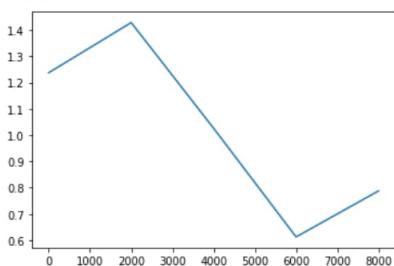
# 勾配適用
W_in -= learning_rate * W_in_grad
W_out -= learning_rate * W_out_grad
W -= learning_rate * W_grad

W_in_grad *= 0
W_out_grad *= 0
W_grad *= 0

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()

iters:0
Loss:1.2375792966460064
Pred:[0 1 1 0 0 1 1 0]
True:[1 0 1 0 1 1 0 0]
70 + 102 = 102
-----
iters:2000
Loss:1.4284801027994764
Pred:[0 1 1 0 0 0 1 0]
True:[1 1 0 0 0 1 0 0]
98 + 98 = 98
-----
iters:4000
Loss:1.0259570948993795
Pred:[0 1 0 1 0 1 1 1]
True:[0 0 0 1 1 0 1 1]
22 + 5 = 27
-----
iters:6000
Loss:0.6123534855462193
Pred:[0 1 0 1 1 0 0 1]
True:[0 1 0 0 1 0 0 1]
1 + 72 = 73
-----
iters:8000
Loss:0.7877624595878249
Pred:[0 1 1 1 0 1 1 0]
True:[0 0 1 1 1 1 1 0]
28 + 34 = 62
-----
```



- 演習チャレンジ
右と左の特徴量を保ったままになるもの
解答 - concatenate がある B
- 1-2 BPTT

- 1-2-1 BPTTとは
 - RNNにおいてパラメータ調整方法の一種
 - 誤差逆伝播の一種

- 確認テスト3

- 連鎖率を使い次をもとめる

$$\begin{aligned} z &= t \cdot 2 \\ t &= x + y \\ \frac{\partial z}{\partial x} &= \frac{\partial z}{\partial t} \frac{\partial t}{\partial x} = 2t \cdot 1 = 2t = 2(x + y) \end{aligned}$$

- 1-2-2 BPTTの数学的記述

- 式とコード

$$\frac{\partial E}{\partial W_{(in)}} = \frac{\partial E}{\partial u^t} \left[\frac{\partial u^t}{\partial W_{(in)}} \right]^T = \delta^{out,t} [z^t]^T$$

```
np.dot(X.T,delta[:,t].reshape(1,-1))
```

$$\frac{\partial E}{\partial W_{(out)}} = \frac{\partial E}{\partial v^t} \left[\frac{\partial v^t}{\partial W_{(out)}} \right]^T = \delta^{out,t} [z^t]^T$$

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

$$\frac{\partial E}{\partial W} = \frac{\partial E}{\partial u^t} \left[\frac{\partial u^t}{\partial W} \right]^T = \delta^t [z^{t-1}]^T$$

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

$$\begin{aligned} \frac{\partial E}{\partial b} &= \frac{\partial E}{\partial u^t} \frac{\partial u^t}{\partial b} = \delta^t \frac{\partial u^t}{\partial b} = 1 \\ \frac{\partial E}{\partial c} &= \frac{\partial E}{\partial v^t} \frac{\partial v^t}{\partial c} = \delta^{out,t} \frac{\partial v^t}{\partial c} = 1 \end{aligned}$$

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

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

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

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

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

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

- パラメータの更新式

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

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

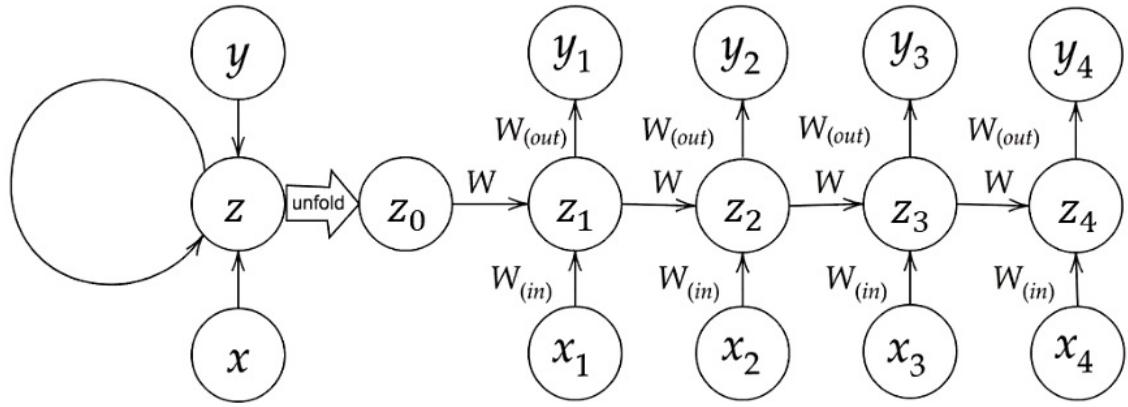
$$W^{t+1} = W^t - \varepsilon \frac{\partial E}{\partial W} = W_{(in)}^t - \varepsilon \sum_{z=0}^{T_i} \delta^{t-z} [z^{t-z-1}]^T$$

$$b^{t+1} = b^t - \varepsilon \frac{\partial E}{\partial b} = b_t^t - \varepsilon \sum_{z=0}^{T_i} \delta^{t-z}$$

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

- 確認テスト4

下図の y_1 を $x \cdot z_0 \cdot z_1 \cdot w_{in} \cdot w \cdot w_{out}$ を用いて数式で表せ



答え

$$y_1 = g(W_{(out)} f(W_{in} z_1 + W z_0 + b) + c)$$

- 1-2-3 BPTTの全体像

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

- コード演習問題

解答 中間層から中間層 \Rightarrow (2)

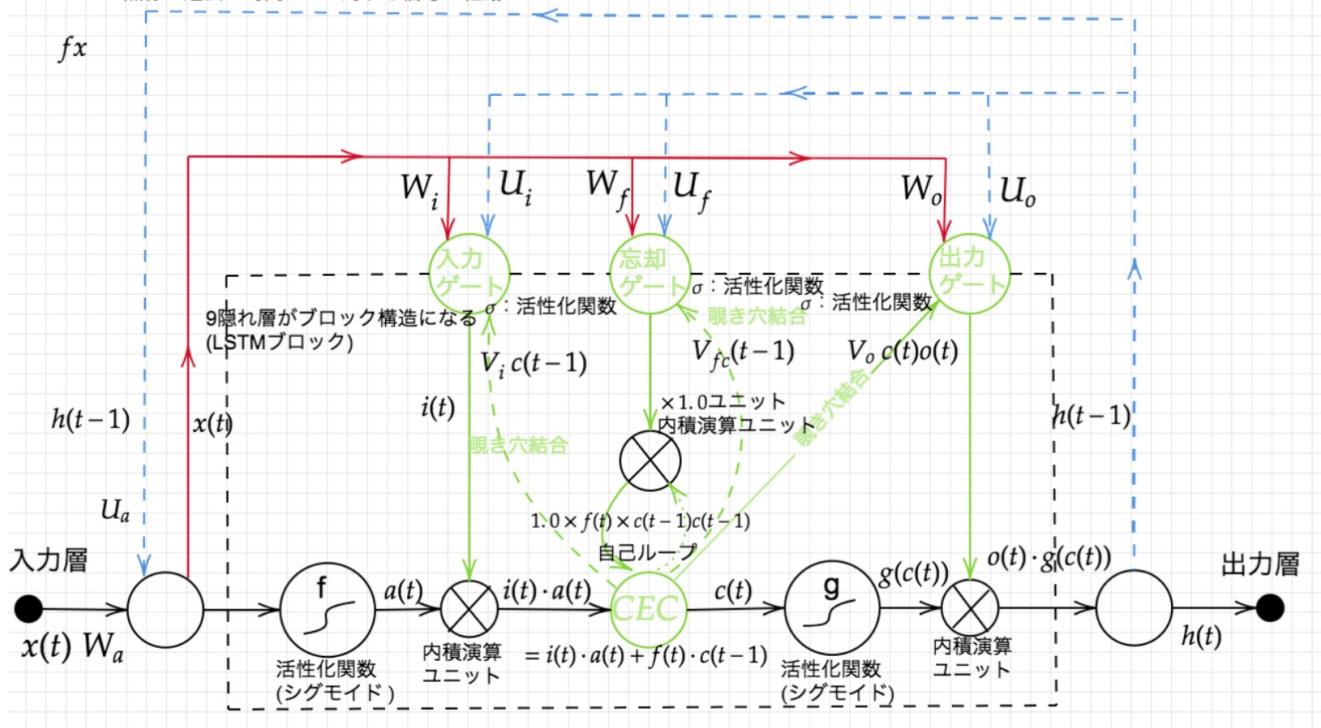
▼ Section2:LSTM

実際にRNNを使ったモデル

- 全体像（前回の流れと課題全体像のビジョン）
 - RNNの課題
 - 時間をさかのぼればさかのぼる \Rightarrow 勾配消失問題発生
 - 解決策
 - 勾配消失問題が起きづらいネットワーク構造にする。
 - 確認テスト5
 - シグモイド関数を微分したとき、入力値が0の時に歳代をとる。その値は
答え (2) 0.25
 - 勾配爆発とは
 - 勾配が、層を逆伝播するごとに指数関数的に大きくなること。
学習率 推奨されている値でない場合割とすぐに起きる。
 - 演習チャレンジ
 - 勾配のクリッピングのソースは
 - 答え 閾値を超したら、勾配のノルムを閾値に正規化する
 \Rightarrow (1) grad * rate
 $(rate = threshold / norm)$ (勾配のノルム) なので
- LSTMの全体像

◎LSTMモデルの定式化

実線：現在の時間tに対する信号の経路
点線：過去の時間t-1に対する信号の経路



• 2-1 CEC

- RNNでいう中間層を抜き出したもの
RNNの問題点 勾配爆発が起こる
⇒ CECは記憶することを担当する
勾配消失および勾配爆発の解決方法は、勾配を1以下にしたらよい

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

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

◦ 課題

- CECは覚えることがないので
⇒ ニューラルネットワークの学習特性がなくなる
⇒ CECの周りに学習機能を配置することで解決（入力ゲートと出力ゲート）

• 2-2 入力ゲートと出力ゲート

- 入力ゲート
 - 入力された値をこんな風に覚えてくださいね というのが役割
今回の入力値と前回の出力値から学習する
- 出力ゲート
 - どんな感じにCECの値を使うか というのが役割
今回の入力値と前回の出力値から学習する

• 2-3 忘却ゲート

- CECの過去の記憶をずっと覚えている
過去の情報がいらなくなった場合、そのタイミングで切り落として忘れてくれる。
今回の入力値と前回の出力値から判断する
 $c(t) = i(t) \cdot a(t) + f(t) \cdot c(t-1)$
- 確認テスト6
以下の文章をLSTMに入力して空欄にあてはまる単語を予測したい
文中の「とても」という言葉は空欄の予測において、無くなってしまって影響がない場合
どのゲートが作用したのか 「映画面白かったね。ところで、とてもお腹が空いていたから何か____。」
答え 忘却ゲート
- 演習チャレンジ
LSTMの順伝播を行なうプログラムで (け) にあてはまるもの
⇒ inputと忘却ゲート

⇒ 答え (3) $\text{input_geta} * \text{a} + \text{forget_geta} * \text{c}$
 (*コメントに書いてある)

- 2-4 覗き穴結合

- CEC自身の値に、重み行列を介して伝播可能にしたもの

- LSTM参考ページ

https://qiita.com/t_Signull/items/21b82be280b46f467d1b#%E5%8F%82%E8%80%83web%E3%83%9A%E3%83%BC%E3%82%B8

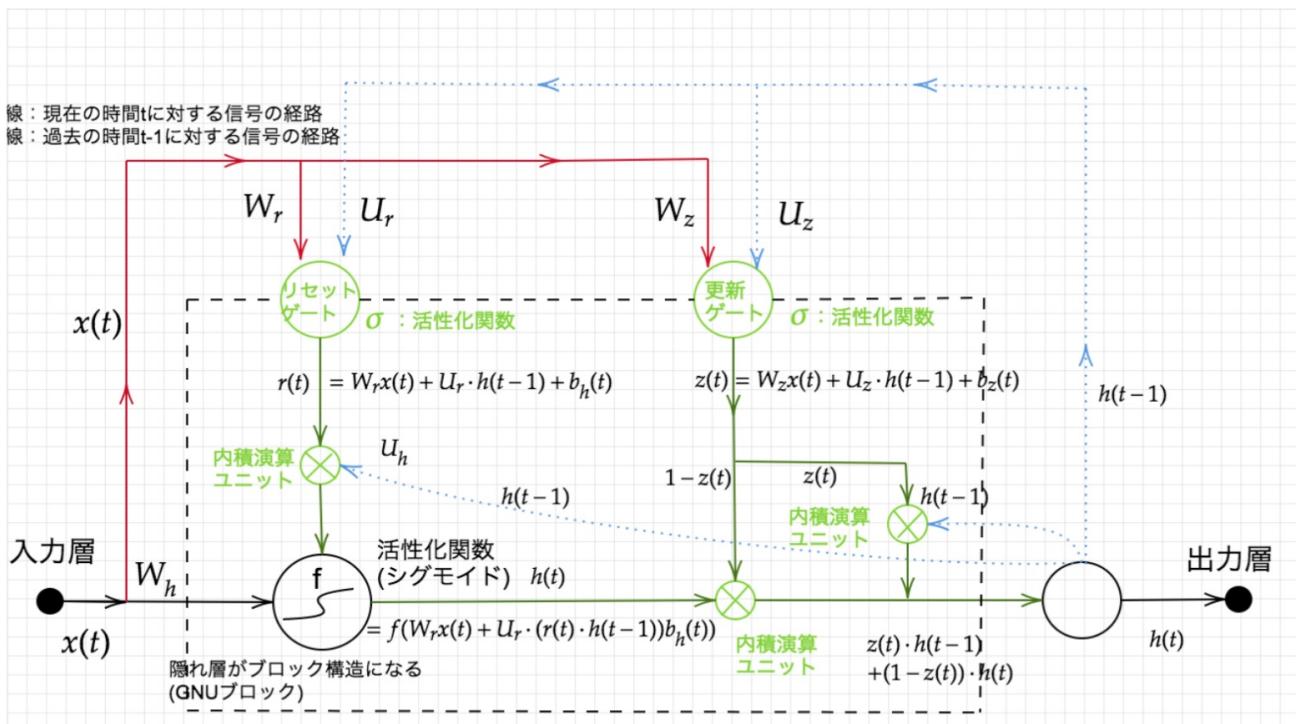
- 意外と歴史のあるモデル（1995年～）
- 初期のころからLSTMは入力ゲート、出力ゲート、忘却ゲート、覗き穴結合があったわけではなく、欠点を補う形で、ゲートが追加されて、進化している。

▼ Section3:GRU

LSTMはRNNを動かそうとしたがやりすぎた

⇒ GRU

- 全体像



- LSTMと違うところ

- 簡素になった
- リセットゲートと更新ゲートのみになった
 - 今回の入力と、前回の出力より求める
 - 隠れ層とした

- 確認テスト 6

- LSTMが抱える課題
 - パラメータが多すぎて計算量が大きい
- CECが抱える課題
 - 学習することができないので、周りに入力ゲート、出力ゲートが必要

- 演習チャレンジ

- GRMの順伝播にあたる（こ）にあてはまるコードは？

答え (4) $(1-z) * h + z * h_{\text{bar}}$
 (全体像の図と見比べる)

- 確認テスト 7

- LSTMとGRUの違い

⇒ LSTM パラメータが多い GRU パラメータが少ない

LSTM 計算量が多い GRU 計算量が少ない

LSTM CRCと入力ゲート、出力ゲートがある GRU リセットゲートと更新ゲートがある

- コード演習(3_3_predict_sin.ipynb)
-

▼ 準備

▼ Google ドライブのマウント

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

Mounted at /content/drive
```

▼ sys.pathの設定

以下では、Google ドライブのマイドライブ直下にDNN_codeフォルダを置くことを仮定しています。必要に応じて、パスを変更してください。

```
import sys
sys.path.append('/content/drive/My Drive/')
```

▼ predict sin

[try]

- iters_numを100にしよう
-

```
import numpy as np
from common import functions
import matplotlib.pyplot as plt
from sklearn.model_selection import train_test_split

np.random.seed(0)

# sin曲線
round_num = 10
div_num = 500
ts = np.linspace(0, round_num * np.pi, div_num)
f = np.sin(ts)

def d_tanh(x):
    return 1/(np.cosh(x)**2 + 1e-4)

# ひとつの時系列データの長さ
maxlen = 2

# sin波予測の入力データ
test_head = [[f[k]] for k in range(0, maxlen)]

data = []
target = []

for i in range(div_num - maxlen):
    data.append(f[i:i + maxlen])
    target.append(f[i + maxlen])

X = np.array(data).reshape(len(data), maxlen, 1)
D = np.array(target).reshape(len(data), 1)

# データ設定
N_train = int(len(data) * 0.8)
N_validation = len(data) - N_train

x_train, x_test, d_train, d_test = train_test_split(X, D, test_size=N_validation)

input_layer_size = 1
hidden_layer_size = 5
output_layer_size = 1

weight_init_std = 0.01
learning_rate = 0.1

iters_num = 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)

# 勾配
W_in_grad = np.zeros_like(W_in)
W_out_grad = np.zeros_like(W_out)
W_grad = np.zeros_like(W)

us = []
zs = []

u = np.zeros(hidden_layer_size)
z = np.zeros(hidden_layer_size)
y = np.zeros(output_layer_size)

delta_out = np.zeros(output_layer_size)
delta = np.zeros(hidden_layer_size)

losses = []

# トレーニング
for i in range(iters_num):
    for s in range(x_train.shape[0]):
        us.clear()
        zs.clear()
        z *= 0

        # sにおける正解データ
        d = d_train[s]

        xs = x_train[s]

        # 時系列ループ
        for t in range(maxlen):

            # 入力値
            x = xs[t]
            u = np.dot(x, W_in) + np.dot(z, W)
            us.append(u)
            z = np.tanh(u)
            zs.append(z)

            y = np.dot(z, W_out)

            #誤差
            loss = functions.mean_squared_error(d, y)

            delta_out = functions.d_mean_squared_error(d, y)

            delta *= 0
            for t in range(maxlen)[-1:-1]:
                delta = (np.dot(delta, W.T) + np.dot(delta_out, W_out.T)) * d_tanh(us[t])

            # 勾配更新
            W_grad += np.dot(zs[t].reshape(-1, 1), delta.reshape(1, -1))
            W_in_grad += np.dot(xs[t], delta.reshape(1, -1))
            W_out_grad = np.dot(z.reshape(-1, 1), delta_out)

            # 勾配適用
            W -= learning_rate * W_grad
            W_in -= learning_rate * W_in_grad
            W_out -= learning_rate * W_out_grad.reshape(-1, 1)

            W_in_grad *= 0
            W_out_grad *= 0
            W_grad *= 0

        # テスト
        for s in range(x_test.shape[0]):
            z *= 0

            # sにおける正解データ
            d = d_test[s]

            xs = x_test[s]

            # 時系列ループ
            for t in range(maxlen):

                # 入力値
                x = xs[t]
                u = np.dot(x, W_in) + np.dot(z, W)
                z = np.tanh(u)

```

```

y = np.dot(z, W_out)

#誤差
loss = functions.mean_squared_error(d, y)
# 出力が長いので一部省略
if s < 5 or s > 95:
    print(' loss:', loss, ' d:', d, ' y:', y)

original = np.full(maxlen, None)
pred_num = 200

xs = test_head

# sin波予測
for s in range(0, pred_num):
    z *= 0
    for t in range(maxlen):
        # 入力値
        x = xs[t]
        u = np.dot(x, W_in) + np.dot(z, W)
        z = np.tanh(u)

        y = np.dot(z, W_out)
        original = np.append(original, y)
    xs = np.delete(xs, 0)
    xs = np.append(xs, y)

plt.figure()
plt.ylim([-1.5, 1.5])
plt.plot(np.sin(np.linspace(0, round_num* pred_num / div_num * np.pi, pred_num)), linestyle='dotted', color='aaaaaa')
plt.plot(original, linestyle='dashed', color='black')
plt.show()

loss: 1.001821168835294e-06    d: [-0.47157024]    y: [-0.47298574]
loss: 3.8316285817396264e-05   d: [-0.39789889]   y: [-0.38914489]
loss: 1.195550219244669e-05   d: [-0.78740743]   y: [-0.79229732]
loss: 5.688169957024536e-07   d: [0.25526991]   y: [0.25633651]
loss: 5.042766131764598e-06   d: [0.6529121]    y: [0.65608787]
loss: 4.7044108996155585e-05  d: [-0.48814053]  y: [-0.47844062]
loss: 1.933106793589577e-06  d: [0.54208448]  y: [0.54405075]
loss: 5.148111233523351e-05  d: [-0.69021707] y: [-0.68007004]
loss: 5.3281078139312404e-05  d: [0.60896952] y: [0.59864663]



```

*かなりかけ離れている。

▼ [try]

- maxlenを5, iters_numを500, 3000(※時間がかかる)にしよう

```

# ひとつの時系列データの長さ
maxlen = 5

```

```

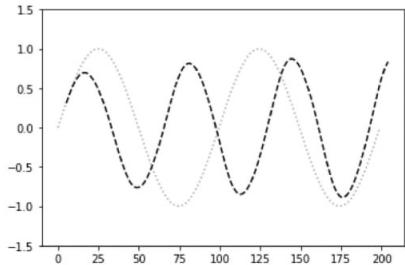
iters_num = 500

```

```

loss: 1.910653011300673e-06    d: [-0.29761864]    y: [-0.29566383]
loss: 4.009145134729275e-06    d: [-0.56307233]    y: [-0.56024067]
loss: 4.117004181533802e-06    d: [-0.65766776]    y: [-0.65479826]
loss: 1.2186275784359606e-06   d: [0.13182648]    y: [0.13026531]
loss: 2.856909186221314e-06   d: [0.49909101]    y: [0.49670065]

```



* 前回より良くなつたが、まだずれがある。

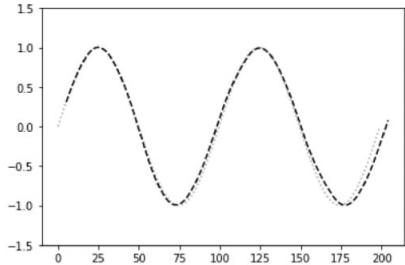
```
# ひとつの時系列データの長さ
maxlen = 5
```

```
iters_num = 3000
```

```

loss: 1.0231756688222737e-07    d: [-0.29761864]    y: [-0.29716628]
loss: 1.2201090156041725e-08    d: [-0.56307233]    y: [-0.56322854]
loss: 4.038245320901024e-11    d: [-0.65766776]    y: [-0.65765877]
loss: 1.012613171470193e-08   d: [0.13182648]    y: [0.13168417]
loss: 6.953992391246068e-08   d: [0.49909101]    y: [0.49871807]

```



* Sinとのずれがかなりなくなった。

- GRU の論文
 - GRUはLSTMのバリエーションの1つとしてとらえることができる。 <https://arxiv.org/pdf/1406.1078v3.pdf>
- 計算量が減っているが、LSTMで扱うことができたものすべてをGRUで扱うことができるわけではない。
 - 計算量と制度はトレードオフの関係にある。

Section4: 双方向RNN

• 特徴

RNNの強力な改良版

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

文章の機械翻訳など

• 演習チャレンジ

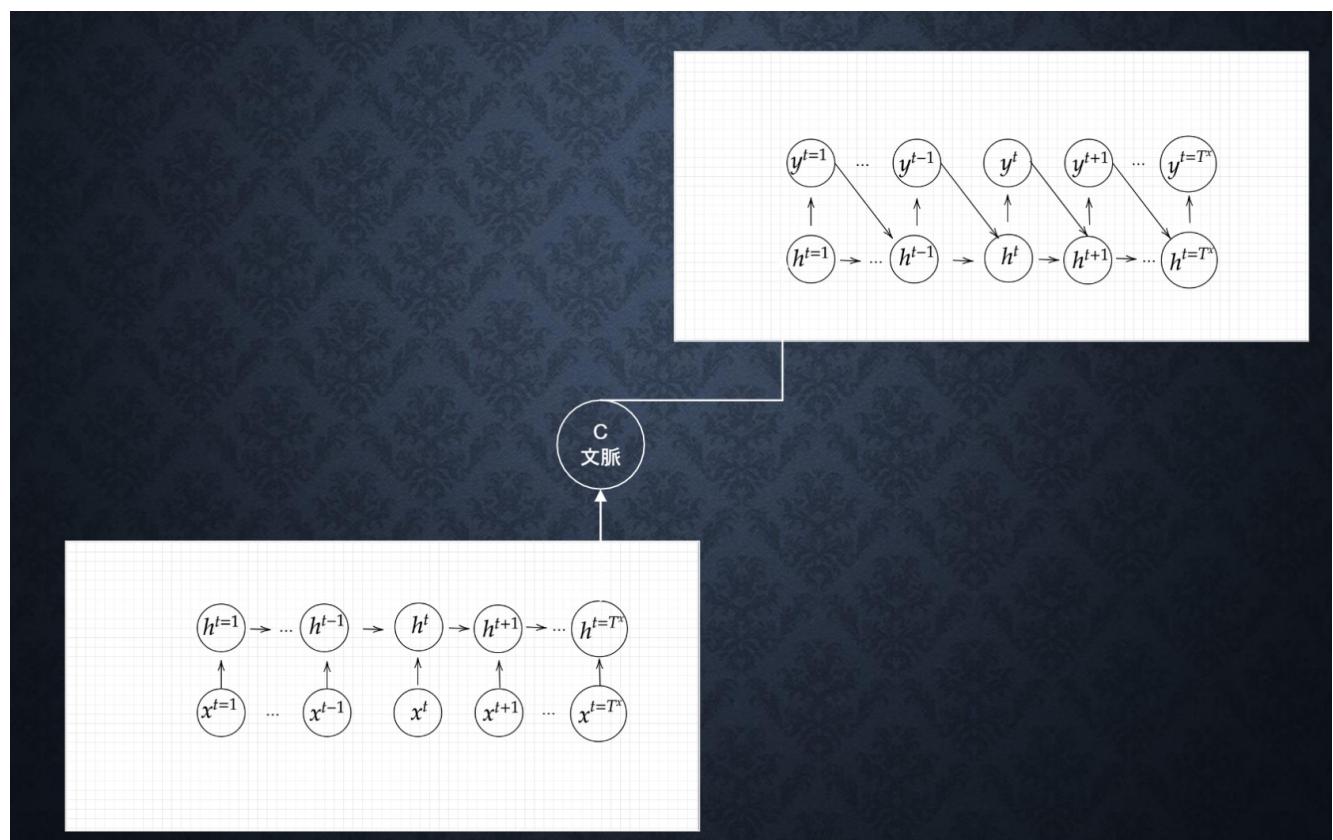
- 双方向RNNの順伝播のプログラム
- 入力から中間層への重み W_f
- ステップ前の中間層出力から中間層への重み U_f
- 逆方向に関しては同様にパラメータ W_b, U_b
- 両者の中間層表現を合わせた特徴から出力層への重みは V

- rnn関数はRNNの順伝播を表し中間層の系列を返す関数
 (か) あてはまるものは
 ⇒ hs_f と hs_b の結合 (axis = 1) (4)
 *時間的に同じものは同じ個所に
- 双方向RNNは2つの学習器を持つ
 - Foward LSTM
 - 通常のLSTM
 - Backward LSTM
 - 後ろの単語から学習する
 - 二つの学習器を組み合わせて学習する

▼ RNNでの自然言語処理

Section5:Seq2Sq

- 全体像



下 エンコーダー

上 デコーダー

- seq2seqとは
 - エンコーダーデコーダーモデルの一種
 - 用途
 - 機械対話、機械翻訳などに使用される
 - 時系列のデータをとって時系列のデータを出力する
- 5-1 Encoder RNN
 - 文の意味を区切る
 例) 昨日 食べた 刺身 大丈夫 でしたか 。
 - 文の全体の意味が最後のベクトルに入る
 - vec1をRNNに入力し、hidden stateを出力
 - このhidden stateと次の入力vec2をまたRNNに入力してきたhidden stateを出力という流れを繰り返す
 - 最後のvecをいれたときのhidden stateをfinal stateとしてとておく。
 このfinal stateがthought vectorとよばれ、入力した分の意味を表すベクトルになる。

		one-hotベクトル	embedding
私	1	[1.0.0...0] ↑ ⁰⁰⁰⁰²	[0.2.0.4 0.6 ... 0.1] ~野菜が好きです。
は	2	[0.1.0. ~ 0]	[-]
刺身	3	[0.0.1. ~ 0]	[-]
;	;	;	;
XXX	10.000	[0.0.0...0.1]	[...]

embedding表現が似ている⇒意味が似ている。

MLM - Masked Language Model (Google)

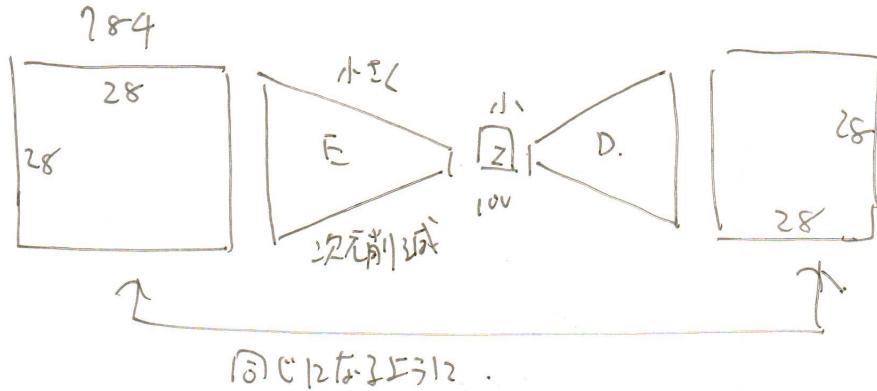
私 は 日曜 ラーメン を 食べ ました
 [-] [-] [-] [-] [-] [-] [-]
 フルを見たい
 ニコニコしながら
 考想する

→ どうなべクトルが出てきたら自然か。
 答えます。

- 5-2 Decoder RNN
 - Encoderが作ったinputをもとに解答の文をつくる。
 - 1. Decoder RNN:

- Encoder RNN のfinal state (thought vector) から、各token の生成確率を出力する。
final state をDecoder RNN のinitial state として設定し、Embedding を入力。
- 2.Sampling:
 - 生成確率にもとづいてtoken をランダムに選びます。
- 3.Embedding:
 - 2で選ばれたtoken をEmbedding してDecoder RNN への次の入力とします。
- 4.Detokenize:
 - 1 -3 を繰り返し、2で得られたtoken を文字列に直します。
- Encoderと逆の処理をする。
- 確認テスト8
 - 次からseq2seqについて説明しているものは
 - (1):双方向RNNの説明
 - (2):○
 - (3):構文木の説明
 - (4):LSTMの説明
- 演習チャレンジ
 - (1) $E \cdot \text{dot}(w)$
単語に対してドット積をとる

(ここから先は文脈に対するもの)
- 5-3 HRED
 - HREDとは
 - seq2seq + Context RNN
 - HREDの課題
 - ありがちな回答しかしなくなる
⇒ VHREDへ
- 5-4 VHRED
 - 文脈の意味にバリエーションを持たせられないか
- 5-5 VAE
 - 5-5-1 オートエンコーダー
 - 教師なし学習のひとつ
 - エンコーダとデコーダからなる
 - Z という潜在変数をとる
 - MNISTの場合
 - 情報量がずっと小さい中間層を通って、出力層へ
- 確認テスト8 . 5
 - seq2seqとHRED、HREDとVHREDの違いを簡潔に述べよ。
 - seq2seqとHRED
 - 1文の1問1答ができるように変換できるネットワークに対して、文脈の意味ベクトルを加えたもの。
 - HREDとVHRED
 - HREDが当たり障りのない単語以上の回答を出せるよう改良したのがVHRED
- 5-5-2 VAE
 - オートエンコーダーの工夫したバージョン
 - 通常のオートエンコーダー
 - ⇒ 正則化を使って Z を求める
 - 今までではデコーダーが復元できるように都合の良い値をとっていた
形同士の近さが求まるとは限らない
 - うまいことできるようにしたのがVAE
 - Z は元のデータの特徴を持ちやすくなる
- 確認テスト9
 - VAEに関する下の空欄を埋めよ
 - 自己符号化器の潜在変数に（確率分布）を導入したもの。



Section6:Word2vec

- 単語をベクトルにする方法
- 単語から使われている単語をピックアップする
例) I want to eat apples. i like apples.
 $\{apples, eat, like, to, want\} \Rightarrow$ 番号を振る
- 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で実現可能
(ONE-HOT-Vectorを使っていても中身がほとんど0の単語数からなるベクトルがいる)
 - 変換のことをWord2vecという (この重みを見つける方法)

Section7:Attention Mechanism

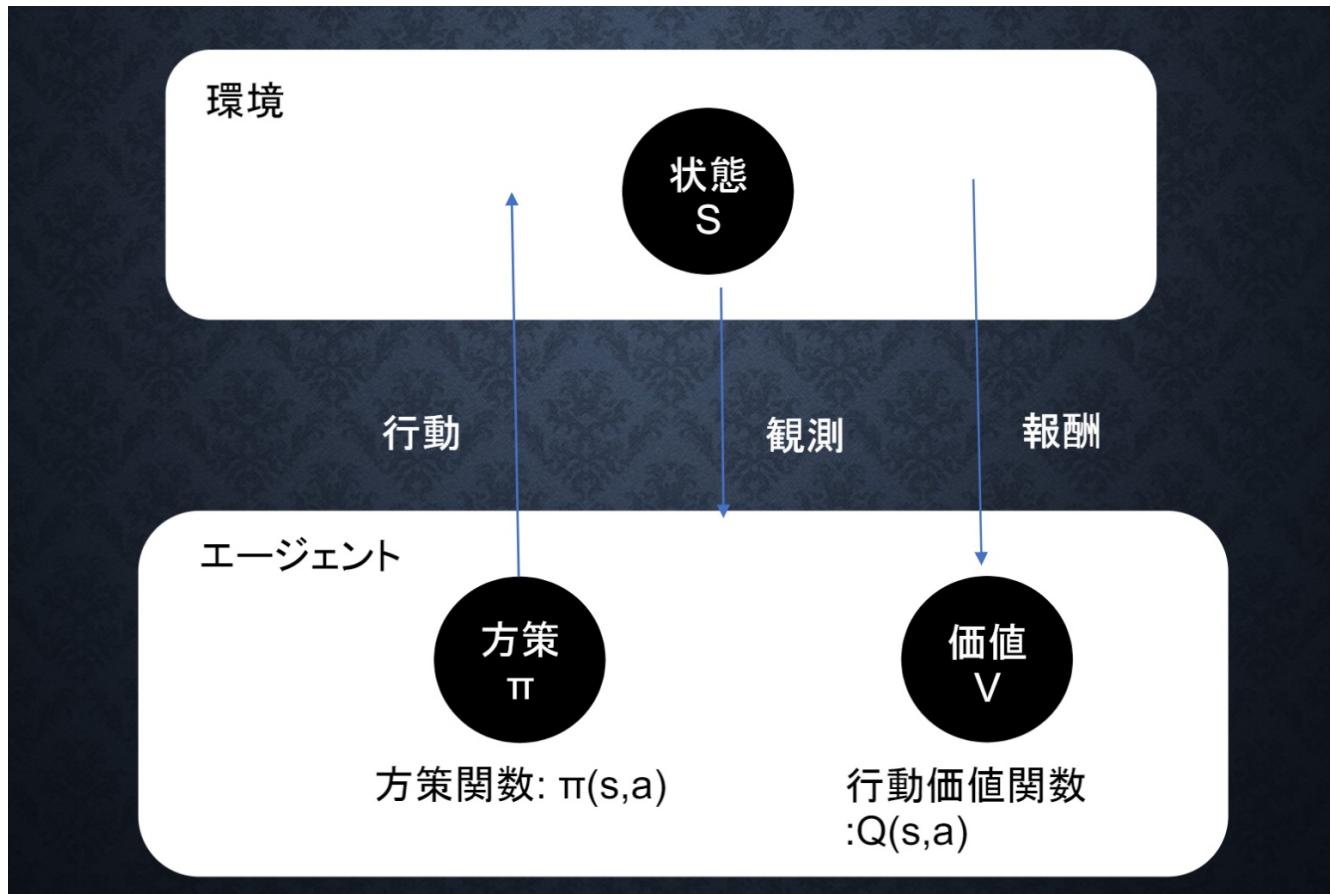
- 課題
 - seq2seqの問題は、長い文章への対応が難しい。
 - 2単語でも、100単語でも、固定次元ベクトルの中に入力する必要がある
 - 中間層が同じなため
- 解決策
 - 1文の中で特に重要な単語を見分ける
 - 文章が長くなるほどそのシーケンスの内部表現の次元も大きくなる仕組みが必要
⇒ Attention Mechanism
 - ⇒ 「入力と出力のどの単語が関連しているのか」
 - 関連度を学習する。
 - 近年使われているのはほぼAttention Mechanism
- 確認テスト1 0
 - RNNとword2vecの違い
 - RNN
 - 時系列データを処理するのに適したネットワーク
 - word2vec
 - 単語の分散表現ベクトルを得る手法
 - seq2seqとAttention Mechanismの違い
 - 1つの時系列データから1つの時系列データを求める方法
 - 時系列データそれぞれの中身に重みを付ける方法

▼ 深層学習Day4

Section1:強化学習

- 1-1 強化学習とは
 - 教師あり、教師なし、教科学習 の分類の1つ

- 行動の結果として与えられる利益（報酬）をもとに、行動を決定する原理を改善していく仕組み
 - 仕事がうまい人は強化学習が得意
⇒ 目標に向かって努力する
- 1-2 強化学習の応用例
 - マーケティングの場合
 - 環境；会社の販売促進部
 - エージェント：プロフィール、購入履歴に基づいて、キャンペーンメールを送る
 - 行動：メールを送る、メールを送らない
 - 報酬：キャンペーンのコストとキャンペーンで生まれる売り上げ
⇒ 売り上げを最大化する。
- 1-3 探索と利用のトレードオフ
 - 最初は当てずっぽうに動くしかない コストがかかる
 - 知識がたまってきたら ベターな方向に動くようになる
- 1-4 強化学習のイメージ



- 1-5 強化学習の差分
 - 教師あり、教師なし学習との違い
 - 目標が違う
 - 教師あり、教師なし学習 - パターンを見つける
 - 教科学習 優れた方策を見つける
- 1-6 行動価値関数
 - 価値観数とは
 - 価値を表す関数 次の2種類がある
 - ある状態の価値に注目する場合 状態価値関数
 - 状態と価値を組み合わせた価値に注目する場合 行動価値関数
⇒ 最近はこっちが人気が高い
 - 式
$$V^\pi(S) : \text{状態関数 } Q^\pi(S, a) : \text{状態} + \text{行動関数}$$
 - ゴールまで 今までの方策を続けたときの報酬の予測値が得られる
⇒ やり続ければ最終的にどうなるか
- 1-7 方策関数

- 方策関数とは
 - エージェントがどんな行動をとるのか決める関数
 - 値関数を最大化するように方策関数は決める
 - エージェントは方策に基づいて行動する
 - $\pi(s,a) : V \text{や} Q \text{をもとにどういう行動をとるか}$
⇒ 経験を生かす OR チャレンジするなど
⇒ その瞬間、その瞬間の行動をどうするか
 - 式

$$\text{方策関数} : \pi(S) = a$$

- 1-8 方策勾配法

- 状態関数と方策関数をうまく学習するための方法
⇒ 関数があったらニューラルネットワークを使えないか

- 方策をモデル化して最適化する手法

- 式

$$\Theta^{(t+1)} = \Theta^{(t)} + \varepsilon \nabla j(\Theta)$$

- Jとは

- 方策の良さ (定義しなくてはならない)

- Jの定義方法

- 平均報酬、割引報酬和

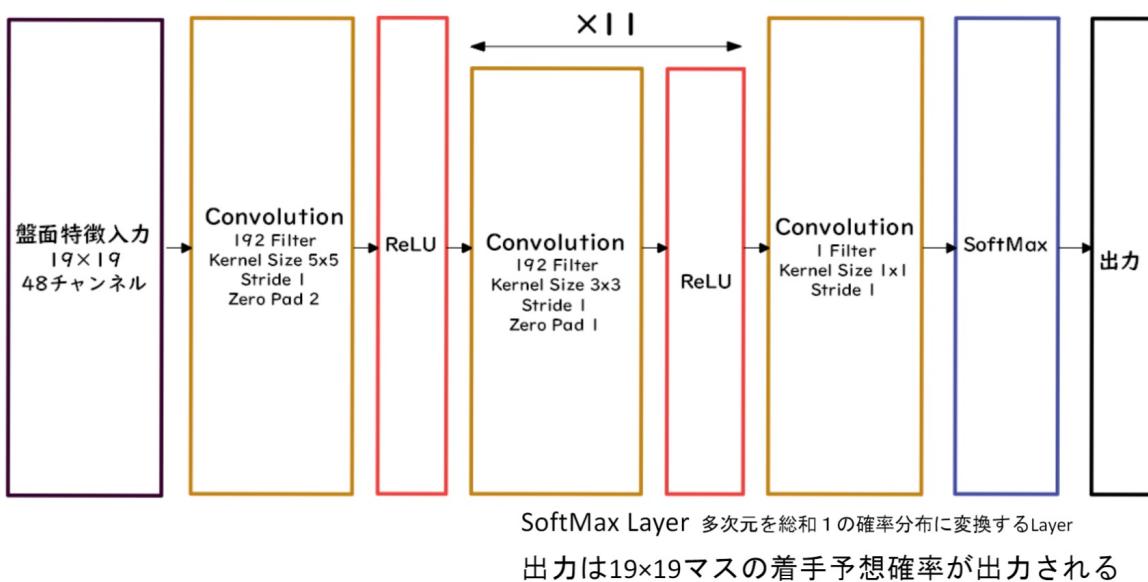
- 上記の定義に対応して、行動価値関数 : $Q(s,a)$ の定義をおこない、方策勾配定理が成り立つ。

$$\begin{aligned} \nabla_{\Theta} J(\Theta) &= \nabla_{\Theta} \sum_{a \in A} \pi_{\Theta}(a|s) Q^{\pi}(s, a) \\ &= E_{\pi_{\Theta}}[(\nabla_{\Theta} \log \pi_{\Theta}(a|s) Q^{\pi}(s, a))] \quad (\text{この式変形は難しいので省略}) \end{aligned}$$

▼ Section2:AlphaGo

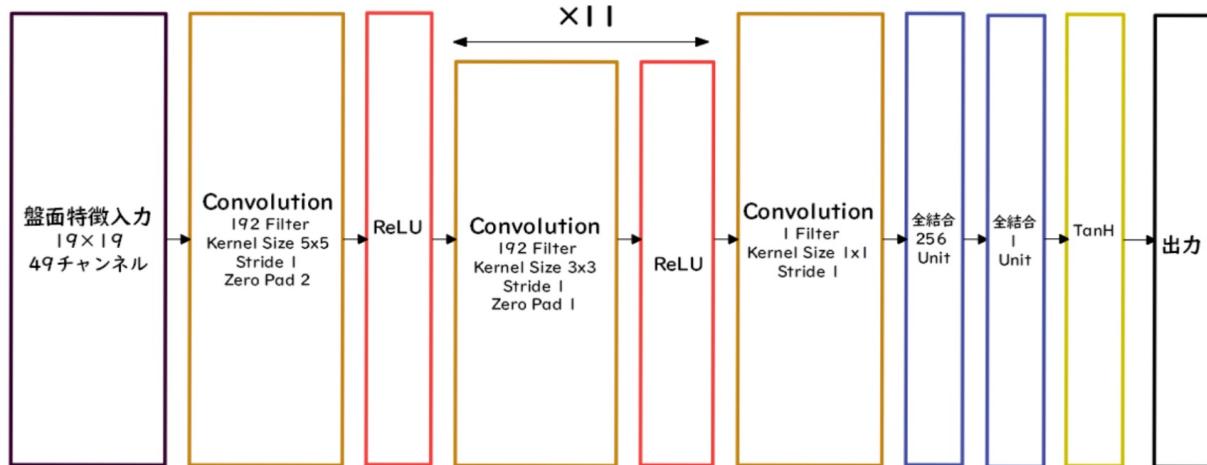
- この成果により、強化学習が非常に注目を受けた
- AlphaGo Lee
 1. AlphaGo LeeのPolicy Net (方策関数にあたる方)

Alpha Go (Lee)のPolicyNet



2. AlphaGo LeeのPolicy Net (方策関数にあたる方)

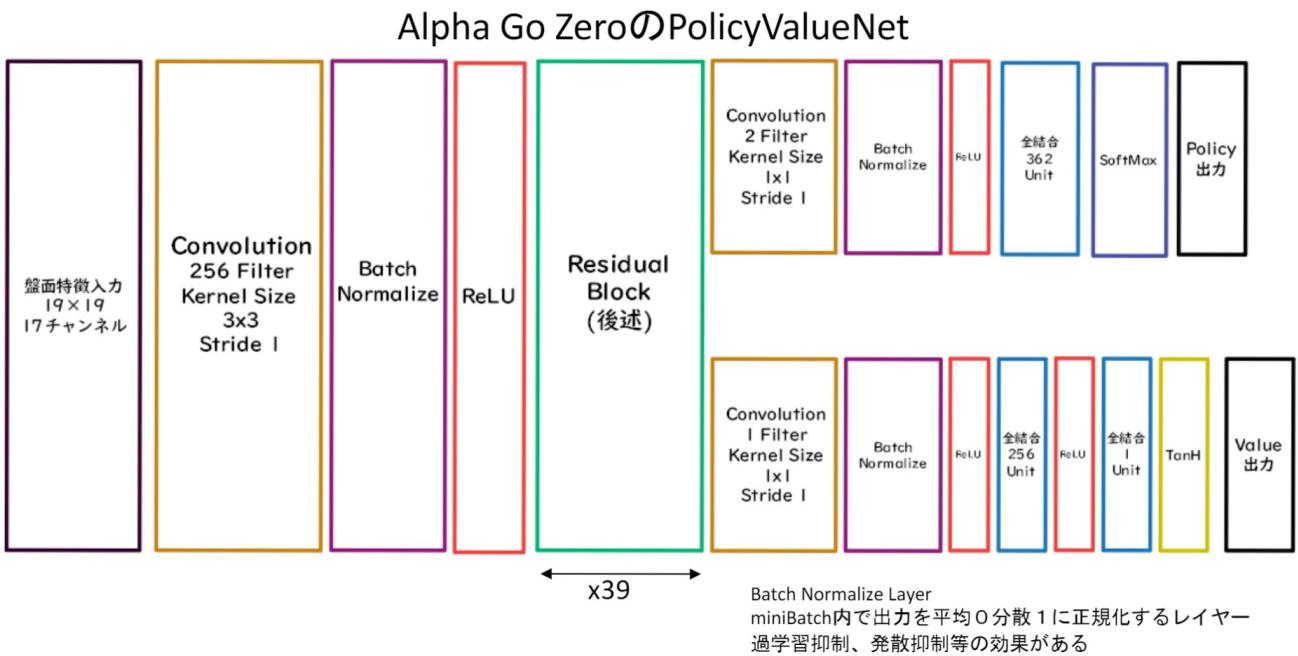
Alpha Go (Lee) のValueNet



TanH Layer $-\infty \sim \infty$ の値を-1~1までに変換するLayer

出力は現局面の勝率を-1~1で表したものが出力される

- Alpha Go の学習
 - 学習ステップ
 1. 教師あり学習によるRollOutPolicyとPolicyNetの学習（工夫）
 - 57%人間と同じ手を打つようになった。
 2. 強化学習によるPolicyNetの学習
 - たくさんPolicyNetがあるプールから何回も学習させる
 3. 強化学習によるVolumeNetの学習
 - PolicyNetの学習のあとにVolumeNetの学習を行う
 - まずは教師あり学習で計算した後、強化学習を行うことで高速化をする
 - RollOutPolicy
 - NNではなく線形の方策関数
 - 探索中に拘束に着手確率を出すために使用される
(計算量が多いときにどうしよう ⇒ RollOutPolicy) (制度は落ちるがスピード重視にする) (PolicyNetの代わりに時間がかかる際に使用する)
 - モンテカルロ木探索
 - 値函数を更新する方法
- AlphaGo Zero
 - AlphaGo LeeとAlphaGo Zeroの違い
 - 教師あり学習を一切行わず、強化学習のみ使用する
 - 入力に関して
 - AlphaGo Lee
 - ニューラルネットワークに入れる入力を人間が決めていた
 - AlphaGo Zero
 - 入力は石の配置のみ
 - PolicyNetとValueNetを1つに統合した
 - Residual Netを導入した
 - モンテカルロ木探索からRollOutシミュレーションをなくした



- ResidualBlock
 - ネットワークに入る前にショートカットを作り、この層をいくつか重ね合わせる
 - 重ね合わせた構造を39個あわせた
 - ショートカットがあるおかげで、深いネットワークであっても、勾配の爆発、消失を抑えることができる
 - スキップがある分いろいろなタイプのネットワークを表現できる
⇒ アンサンブル効果
- AlphaGo Zeroの工夫点 (E資格はこういうことがよく出る) (基本的なパートは今までと一緒だが工夫点に名前を付けただけのもの)
 - Resudual Blockの工夫
 - Bottleneck
1×1 KernelのConvolutionを利用し、1層目で次元削減を行って3層目で次元を復元する3層構造にし、2層のものと比べて計算量はほぼ同じだが1層増やせるメリットがある、としたもの
 - PreActivation
ResidualBlockの並びをBatchNorm→ReLU→Convolution→BatchNorm→ReLU→Convolution→Addとすることにより性能が上昇したとするもの
 - Network構造の工夫
 - WideResNet
ConvolutionのFilter数をk倍にしたResNet。1倍→k倍×ブロック→2*k倍ブロックと段階的に幅を増やしていくのが一般的。Filter数を増やすことにより、浅い層数でも深い層数のものと同等以上の精度となり、またGPUをより効率的に使用できるため学習も早い
 - PyramidNet
WideResNetで幅が広がった直後の層に過度の負担がかかり精度を落とす原因となっているとし、段階的にではなく、各層でFilter数を増やしていくResNet。

Section3:軽量化・高速化技術

- 現在技術的にどうにかしたいので研究が盛んな分野
- モデル高速化
 - 分散深層学習とは
 - 深層学習はとても計算に時間がかかる
(毎年10倍の大きさで計算量が増えている)
 - 分散させられないか (並列: 数を増やす)
 - 3-1 モデル並列
 - 親モデルを各ワーカに分割してそれぞれに学習させる
 - すべてのデータでの楽手が終わったら1つのモデルに復元する

- モデルのパラメータが多いほど、スピードアップの効果も向上する
(逆にあまりにも小さいと効率が悪い)
- 参考論文
 - Large Scale Distributed Deep Networks(Google 2016)
Tensorflowの前身といわれている
- 3-2 データ並列
 - 親モデルを各ワーカーに子モデルとしてコピー
 - データを分割して各ワーカーごとに計算
 - ネットワークでつないでうまいことやる
 - CPU、GPUが複数ついているパソコンを使用して1つ1つをワーカーとする
 - 寝てる間のスマホを使うなど（研究段階）
- モデルが大きいとき ⇒ モデル並列化
- データが大きいとき ⇒ データ並列化
- 3-3 GPU
 - CPU
 - 高性能なコアが少数
 - 複雑で連続的な処理が得意
 - GPGPU
 - グラフィック以外の用途で使用されるGPUの総称
 - GPU
 - 比較的低性能なコアが多数
 - 簡単な並列処理が得意
 - ニューラルネットの学習は単純な行列演算が多いので、高速化が可能
⇒ この機能がDeep Learningの学習にちょうど良かった。
 - GPGPU開発環境
 - CUDA(現在はほぼこっち)
 - NVIDIA社が開発しているGPUのみで使用可能
 - OpenCl
 - NVIDIA以外のGPUでも使用可能
 - DeepLearningフレームワーク（Tensorflow）内で実装されているので、使用する際に指定するのみでよい。
- 非力なマシーン（例えばスマホ）などで動かせるようにする工夫
 - 3-4 量子化
 - ネットワークが大きくなる
 - ⇒ 大量のパラメータが必要、学習や推論に多くのメモリと演算処理が必要
 - ⇒ パラメータの64bit浮動小数点を32bitなど下位の精度に落とす（メモリと演算処理の削減を行う）
 - 利点
 - 計算の高速化
 - 省メモリ
 - 欠点
 - 精度が落ちる
 - 3-5 蒸留
 - 精度の高いモデルはニューロンの規模が大きなモデルになっている、そのため推論に多くのメモリと演算処理が必要
 - ⇒ 規模の大きなモデルの知識を使い軽量なモデルの作成を行う
 - モデルの簡略化
 - 精度の高いモデルに近い性能の 軽量なモデルが欲しい
 - ⇒ 知識の継承
 - 教師モデルと生徒モデル
 - 教師モデル
 - 元の重たいモデル
 - 生徒モデル
 - 軽量なモデル
 - 教師モデルの重みを固定して、生徒モデルの重みを更新していく
誤差は教師モデルと生徒モデルのそれぞれの誤差を使い、重みを更新する
 - 3-6 プルーニング
 - ネットワークが大きくなったとき、パラメータ全部が役に立っていないので、役に立っていないパラメータは消す手法
 - ニューロンの削減
 - 重みが閾値以下のニューロンを消す

- 想像以上にニューロンは消せる

Section4:応用モデル

• 4-1 MobileNet

- 一般的な畳み込みレイヤー

- 入力特徴マップ（チャネル数）： $H * W * C$
(Cはカラー画像だと3枚)
- 畳み込みカーネルのサイズ： $K * K * C$
- 出力チャネル数（フィルター数）： M
- ストライド1でパディングを適用した場合の畳み込みの計算の計算量
 $H * W * K * K * C * M$

- MoboleNet(は

- 一般的な畳み込みレイヤーは計算量が多い
- 組み合わせで軽量化を実現（Depthwise ConvolutionとPointwise Convolution）

- Depthwise Convolution
 - フィルター：1固定
 - カーネルだけで処理
 - 1チャンネルごとにしか畳み込みしない
 - 出力マップは
 $H * W * C * K * K$
- Pointwise Convolution
 - カーネルを $1 * 1$ で固定で畳み込み
 - フィルターを動かす
 - 出力マップは
 $H * W * C * M$

- 2つ（Depthwise ConvolutionとPointwise Convolution）に分けることで計算量を減らす

DepthwiseConvolutionはチャネル毎に空間方向へ畳み込む。すなわち、チャネル毎に $D_K \times D_K \times 1$ のサイズのフィルターをそれは $(H \times W \times C \times K \times K)$ となる。

次にDepthwiseConvolutionの出力をPointwiseConvolutionによってチャネル方向に畳み込む。すなわち、出力チャネル毎にフィルターをそれぞれ用いて計算を行うため、その計算量は $(H \times W \times C \times M)$ となる。

• 4-2 DenseNet

- Dense Blockが特徴

- 出力層に前の層の入力を足し合わせて、処理する
- 特徴マップの入力に対して

- Batch正規化
- Relu関数による変換
- $3 * 3$ 畳み込み層による処理

- Dense Blockの中の計算

- 前の層での値にこの層での出力を付け加える

$$\begin{aligned} \text{入力 : } & K_0 \quad \text{出力 : } k_0 + k \\ \text{入力 : } & K_0 + k \quad \text{出力 : } k_0 + k \\ \text{入力 : } & K_0 + 2k \quad \text{出力 : } k_0 + k \\ \text{入力 : } & K_0 + 3k \quad \text{出力 : } k_0 + k \\ \text{入力 : } & K_0 + 4k \quad \text{出力 : } k_0 + k \end{aligned}$$

という具合に

- Transition Layer
- 特徴量を計算して、出力のレイヤーを減らす
- DenseNetとResNetの違い
- DenseBlockでは前方の各層からの出力すべてが後方の層への入力として用いられる
- RessidualBlockでは前の1層の入力のみ後方への層へ入力
- DenseNet内で使用されるDenseBlockと呼ばれるモジュールでは、成長率（Growth Rate）と呼ばれるハイパーパラメータが存在する

• 4-3 Layer Norm/Instance Norm

(はつきりした要因はわからないが、こうやつたらうまくいったというもの)

- Batch Norm
 - ミニバッチに含まれるサンプルの同一チャネルが同一分布に従うよう正規化
 - ミニバッチが厄介で、ハードウェアによってミニバッチのサイズを変えざる負えない
- Layer Norm
 - それぞれのサンプルのすべてのピクセルが同一分布に従うように正規化
 - (ミニバッチの影響を受けない)
- Instance Norm
 - さらにチャネルも同一分布にしたがうよう正規化
 - (ミニバッチの影響を受けない)
- 4-4 Wavenet
 - 生の音声生成モデル
 - 置込みを行うことで音声を生成できる
 - 時系列データに対して置込み（Dilated Convolution）を適用する
 - 相が深くなるにつれて置込みリンクを離す（とばす）
 - 情報量を一定にできる

Section5:Transformer

seq2seq

Transformerを理解するのにseq2seqの知識が必要

- Seq2seqとは
 - 系列（Sequence）を入力として、系列を出力するもの
 - 入力系列がEncode（内部状態に変換）され、内部状態からDecode（系列に変換）
 - 例として
 - 翻訳（英語⇒日本語）
 - 音声認識（波形⇒テキスト）
 - チャットボット（テキスト⇒テキスト）
- 大きく2つの理解が必要
 - RNNの理解（既出）
 - 言語モデルの理解
 - 時刻t-1までの情報で、時刻tの事後確率をもとめることが目標
⇒ これで同時確率が計算できる
- $$\underset{w \in V}{\operatorname{argmax}} \quad P(I, \text{have}, a, w)$$
- RNN * 言語モデル
 - 各地点で次にどの単語が来れば自然（事後確率最大）かを出力できる
 - 言語モデルを再現するようにRNNの重みが学習されていれば、ある時点のお次の単語を予測可能
- seq2seq
 - EncoderとDecoderからなる（既出）

Transformer

- ニューラル機械翻訳の問題点 ⇒ 長さに弱い
- Attention
 - 翻訳先の各単語を選択する際に、翻訳元の文中の各単語の隠れ状態を利用
 - 翻訳元の各単語の隠せ状態の加重平均

$$c_i = \sum_{j=1}^{T_x} a_{ij} h_j$$
 - 重み（すべて足すと1）重みはFFNNでもとめる

$$a_{ij} = \frac{\exp(e_{ij})}{\sum_{k=1}^{T_x} \exp(e_{ik})}$$

$$e_{ij} = a(s_{i-1}, h_j)$$
- Attentionは何をしているのか
 - Attentionは辞書オブジェクト
 - query(検索クエリ)に一致するkeyを索引し、対応するvalueを取り出す操作であるとみなすことができる。これは辞書オブジェクト

の機能と同じ。

- 文長が長くなってしまっても精度が落ちない

Transformerについて

- 2017年6月に登場
- RNNをつかわない 必要なのはAttentionだけ
- 計算量が当時のSOTAよりはるかに少ない
⇒英仏（3600万文）の学習を8GPUで3.5日で完了
- 注意機構には2種類ある

$$\text{softmax}(QK^T)V$$

- Source Target Attention
 - ソースターゲット注意機構

$$\sigma(\underbrace{[query \quad key]}_{\text{Target}}) \underbrace{[value]}_{\text{source}}$$

- Self-Attention
 - 自己注意機構

$$\sigma(\underbrace{[query \quad key]}_{\text{source}}) \underbrace{[value]}_{\text{source}}$$

- Transformer-Encoder
 - 自己注意機構により文脈を考慮して各単語をエンコード

- Self-Attention
 - 入力をすべて同じにして学習的に注意個所を決めていく

- Position-Wise Feed-Forward Networks
 - 位置情報を保持したまま順伝播させる

- Scaled dot product attention
 - 全単語に関するAttentionをまとめて計算する

- Multi-Head attention
 - 重みパラメタの異なる8個のヘッドを使用
 - 8個のScaled Dot-Product Attentionの出力をConcat
 - それぞれのヘッドが異なる種類の情報を収集

- Add & Norm
 - Add (Residual Connection)
 - 入出力の差分を学習させる
 - 実装上は出力に入力をそのまま加算するだけ
 - 効果：学習・テストエラーの低減
 - Norm (Layer Normalization)
 - 各層においてバイアスを除く活性化関数への入力を平均0、分散1に正則化
 - 効果：学習の高速化

- Position Encoding

RNNを用いないので単語列の語順情報を追加する必要がある

- 単語の位置情報をエンコード

$$PE_{(pos,2i)} = \sin\left(\frac{pos}{10000^{2i/512}}\right)$$

$$PE_{(pos,2i+1)} = \cos\left(\frac{pos}{10000^{2i/512}}\right)$$

- posの(ソフトな)2進数表現

▼ Section6: 物体検知・セグメンテーション

- 共通して必要なことのみ紹介する

- 物体認識

- 共通していること

- 入力：画像
- 出力：分類

画像に対して单一、複数のクラスラベル

物体検知	
Bounding Box(どこに何があるのか)	意味領域分割
バウンディングボックスの代わりに	
各ピクセルに対して単一のクラスラベル（インスタンスの区別なし）	
個体領域分割	
バウンディングボックスの代わりに	
各ピクセルに対して単一のクラスラベル（インスタンスの区別あり）	

- 代表的なデータセット

- VOC12
 - Visual Object Classes(コンペ自体は2012年に終了)
 - クラス数：20
 - Train+Val : 11540
 - 1画像当たりのBox数 : 2.4
 - (Instance Annotationあり)
- ILSVRC17
 - ImageNet Scale Visual Recognition Challenge (コンペ自体は2017年に終了)
 - クラス数 : 200
 - Train+Val : 476668
 - 1画像当たりのBox数 : 1.1
 - ImageNetのサブセット
 - (Instance Annotationあり)
- MS COCO18
 - MS COCO Object Detection Challenge
 - クラス数 : 80
 - Train+Val : 123287
 - 1画像当たりのBox数 : 7.3
 - (Instance Annotationあり)
- OICOD18
 - Open Images Challenge Object Detection
 - クラス数 : 500
 - Train+Val : 1743042
 - 1画像当たりのBox数 : 7.0
 - Open Image V4のサブセット
 - (Instance Annotationあり)

- 評価手法

- Confusion Matrix

真値 \ 予測	<i>Positive</i>	<i>Negative</i>
<i>Positive</i>	<i>TruePositive</i>	<i>FalseNegative</i>
<i>Negative</i>	<i>FalsePositive</i>	<i>TruePositive</i>
<i>Recall</i> (再現率) = $\frac{TP}{TP + FN}$		
<i>Precision</i> (適合率) = $\frac{TP}{TP + FN}$		

- IoU : Intersection over Union

- 物体検出においてはクラスラベルだけでなく、物体位置の予測精度も評価したい
- Confusion Matrixの要素を用いて表現

$$IoU = \frac{TP}{TP + FP + FN}$$

- AP : Avarage Precision Confの閾値を変える

conf. の閾値を β とするとき、
 $Pecall = R(\beta), Precision = P(\beta)$
 $\Rightarrow P = f(R)[Precision - Recallcurve]$

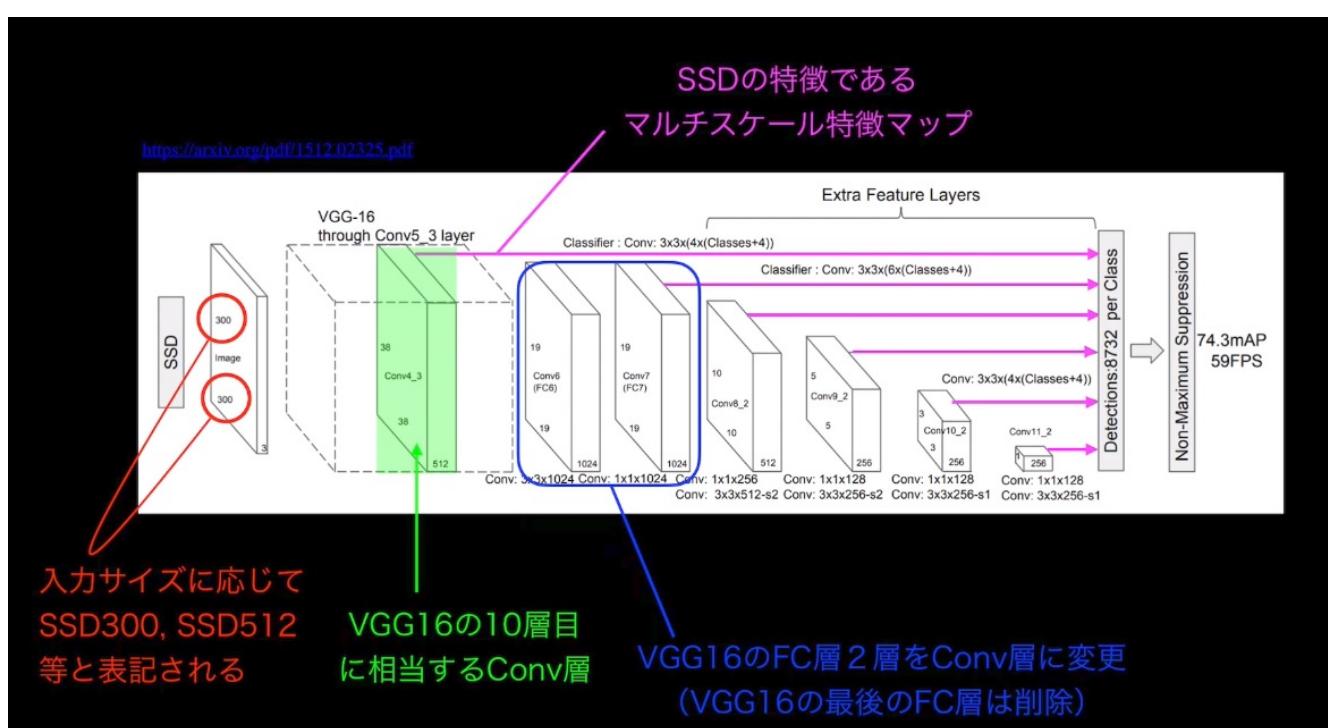
$$AP = \int_0^1 P(R)dR$$

- mAP : mean Average Precision

$$mAP = \frac{1}{C} \sum_{i=1}^c AP_i$$

- FPS: Frames per Second

- 応用上の要請から、検出精度に加えて検出速度も問題になる。
 - 物体検知のフレームワーク
 - 2段階検出器 (Two-stage detector)
 - 候補領域の検出とクラス推定を別々におこなう
 - 相対的に精度が高い
 - 相対的に計算量が大きく推論も遅い
(リアルタイムに向かない)
 - 例) RCNN, SPPNet, Fast RCNN, Faster RCNN, RFCN, FPN, Mask RCNN
 - 1段階検出器 (One-stage detector)
 - 候補領域の検出とクラス推定を同時におこなう
 - 相対的に精度が低い
 - 相対的に計算量が小さく推論も遅い
 - 例) DetectorNet, YOLO, SSD, YOLO9000, RetinaNet, CornerNet- SSD:Single Shot Detector
 - 1段階検出器の一つ
 - 初めに「デフォルトボックス」を用意する



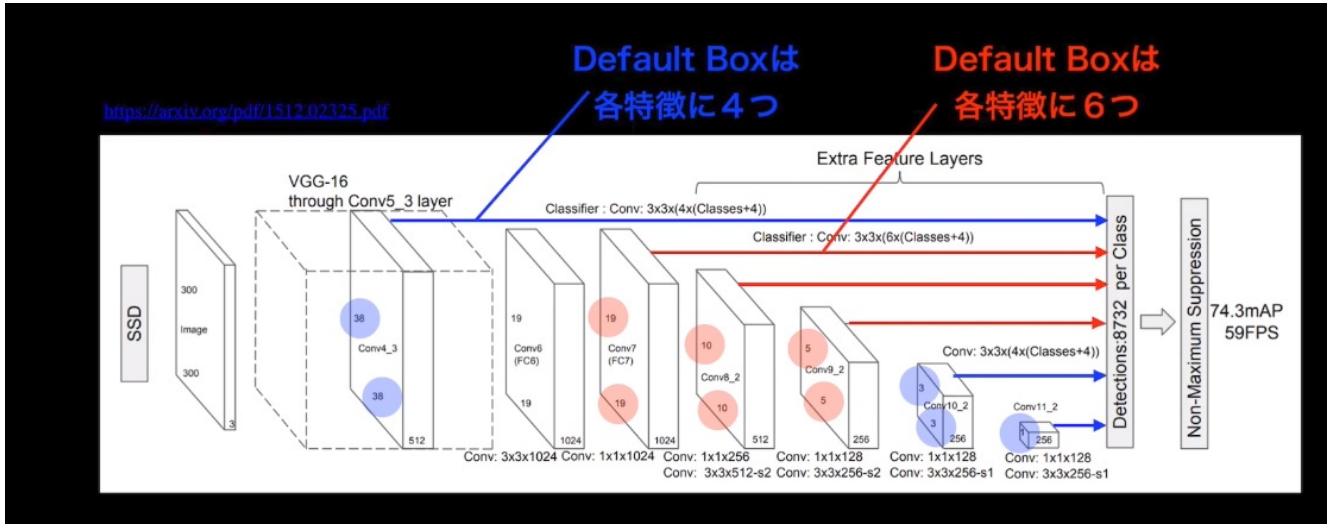
- 特徴マップからの出力
 - マップ中の1つの特徴量における1つのDefault Boxについて
出力サイズ: # Class + 4(オフセット項: $\Delta x, \Delta y, \Delta w, \Delta h$)
 - マップ中の各特徴量にKこのDefault Boxを用意するとき
出力サイズ: k (# Class + 4)
 - さらに、特徴マップのサイズがm*nであるとすると
出力サイズ: k (# Class + 4) mn
 SSD におけるオフセット項の注意

$$x = x_0 + 0.1 * \Delta x * w_0$$

$$y = y_0 + 0.1 * \Delta x * h_0$$

$$w = w_0 * \exp(0.2 * \Delta w)$$

$$h = h_0 * \exp(0.2 * \Delta h)$$



VOCデータセットでは、クラス数20に背景クラスが考慮され、# Class = 21となることに注意して

$$4 * (21 + 4) * 38 * 38 + 4 * (21+4) * 3 * 3 + 4 * (21 + 4) * 1 * 1$$

+

$$6 * (21 + 4) * 19 * 19 + 6 * (21+4) * 10 * 10 + 6 * (21 + 4) * 5 * 5$$

$$= 8732 * (21 + 4)$$

- 多数のDefault Boxを用意したことによる問題への対処

- Non-Maximum Suppression

1つの物体しか映っていないなくても、あたかも複数の物体があるように検知

⇒ IOUを計算して基準値以上のもののみ残す

- Hard Negative Mining

物体として映っているものと背景として映っているもののバランスが崩れる

⇒ 背景は物体の3倍までとするなど制約をつける

- 損失関数

$$L(x, c, l, g) = \frac{1}{N} (L_{couf}(x, c) + \alpha L_{loc}(x, l, g))$$

- confidenceに対する損失

$$L_{couf}(x, c) = - \sum_{i \in P_{os}}^N x_{ij}^p \log(\hat{c}_i^p) - \sum_{i \in N_{eg}}^N \log(\hat{c}_i^0) \quad \text{where } \hat{c}_i^p = \frac{\exp(c_i^p)}{\sum_p \exp(c_i^p)}$$

- 検出位置に対する損失

$$\begin{aligned} L_{loc}(x, l, g) &= \sum_{i \in P_{os}} \sum_{m \in \{cx, cy, w, h\}} x_{ij}^k \text{smooth}_{L1}(l_i^m - \hat{g}_j^m) \\ \hat{g}_j^{cx} &= (g_j^{cx} - d_i^{cx}) / d_i^w \\ \hat{g}_j^{cy} &= (g_j^{cy} - d_i^{cy}) / d_i^h \\ \hat{g}_j^w &= \log\left(\frac{g_j^w}{d_i^w}\right) \\ \hat{g}_j^h &= \log\left(\frac{g_j^h}{d_i^h}\right) \end{aligned}$$

- SSD 論文 <https://arxiv.org/abs/1512.02325>

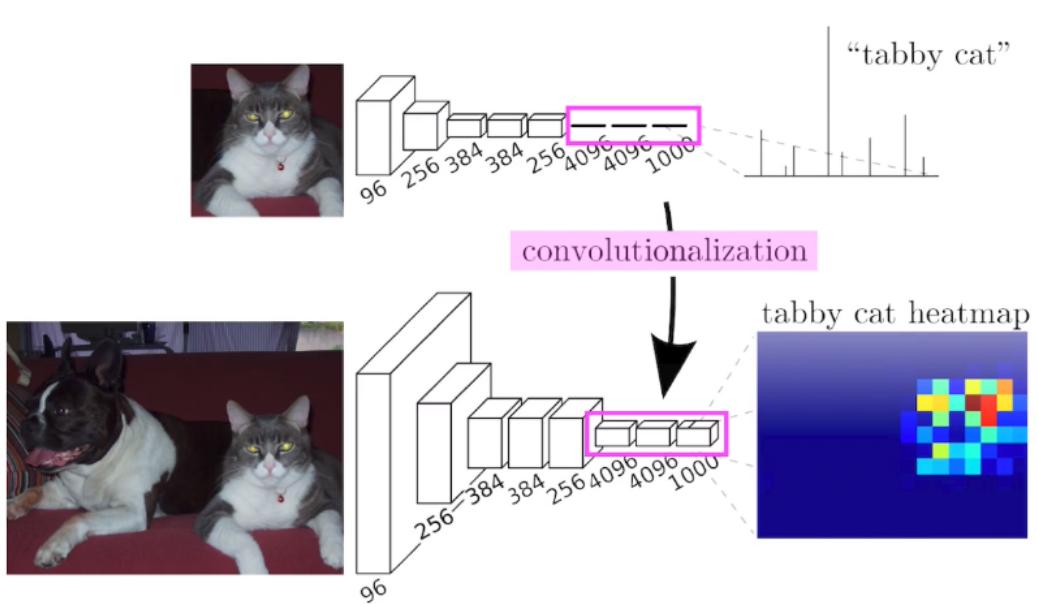
▼ Semantic Segmentationの概略

- 正しく認識するためには受容野にある程度の大きさが必要だった

- 受容野を広げる方法

1. 深いConv.層
2. プーリング

- FCN(Fully Connected Network)の基本アイデア



* 最後のFC層をConvolution層に置き換えている

- Deconvolution • Transposed convolution
 - 通常のConv.層と同様、kernel size, padding, strideを指定
 - 処理手順
 1. 特徴マップのpixel感覚を「stride」だけ空ける
 2. 特徴マップの周りに(kernel size -1) - paddingだけ余白を作る
 3. 置み込み演算を行う
- 輪郭情報の補完
 - poolingによりローカルな情報が失われていく
⇒ 低レイヤーPooling層の出力をelement-wise additionすることで、ローカルな情報を保管してからUp-samplingする
- Dilates Convolution
 - Convolutionの段階で受容野を広げる工夫