【E 資格学習レポート】深層学習後編(day3、day4)レポート

深層学習後編(day3)については以下７つの科目でレポートする

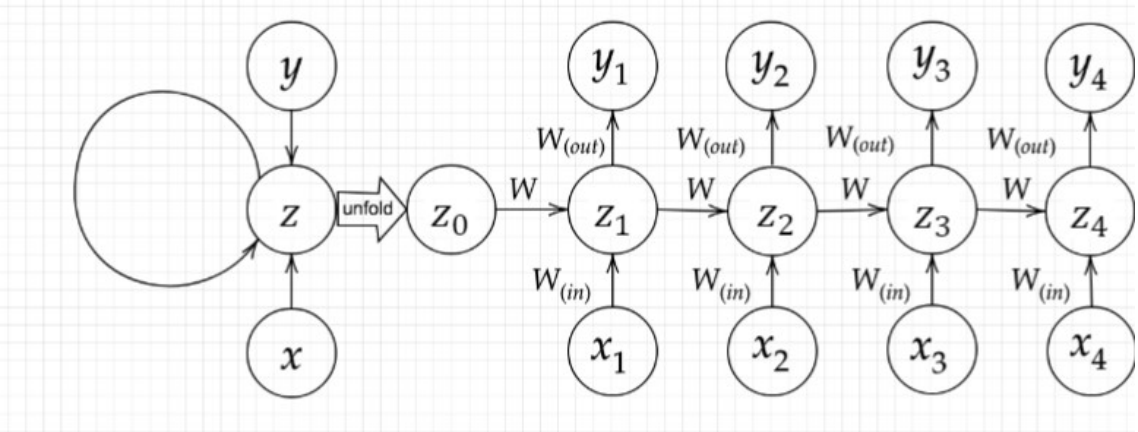
Section１：再帰型ニューラルネットワークの概念

・ＲＮＮ（Ｒｅｃｃｕｒｅｎｔ　Ｎｅｕｒａｌ　Ｎｅｔｏｗｒｋ）

・時系列データ（音声データ、テキストデータ）に対応可能なニューラルネットワーク

（時系列データとは？時間的順序を追って一定間隔ごとに観察され，しかも相互に統計的依存関係が認められるようなデータの系列）

・RNN の全体像



・ＲＮＮの数学的記述

・ＲＮＮの特徴（再帰的構造）

・ＢＰＴＴ（通時的誤差逆伝播法(back-propagation through time 法)）とは：ＲＮＮにおいてのパラメータ調整方法の一種

・ＢＰＴＴの数学的記述

パラメータの更新式

・時点ｔと時点ｔ－１には関係がある事が見てとれる。

・ＲＮＮが過去の時間を再帰的にたくわえている事を示している。

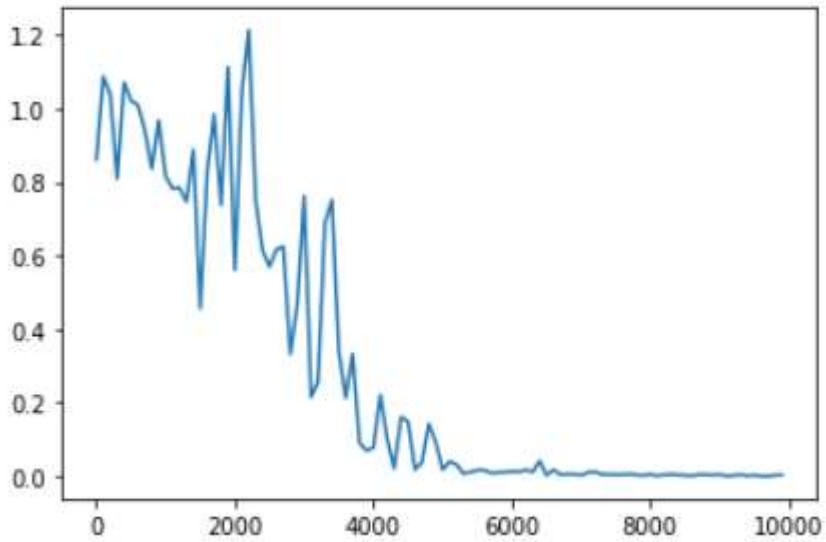
・RNN では中間層出力 h_{t}が過去の中間層出力 h_{t-1},.., h_{1}に依存する。RNN において損失関数を重み W や U に関して偏微分するときは、それを考慮する必要がある。

・BPTT の全体像

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

実行結果：

```
------------
iters:9900
Loss:0.0041168597621107856
Pred:[1 0 0 1 1 1 1 0]
True:[1 0 0 1 1 1 1 0]
54 + 104 = 158
------------
```



参照コード：

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

```
import sys
sys.path.append('/content/drive/My Drive/DNN_code_3_4')
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.ui
nt8).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_l
ayer_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_lay
er_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]:
        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_ou
t[:,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()
```

確認テスト 1

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。

回答：中間層から中間層へ至るところの重み。

確認テスト 2

連鎖律の原理を使い、dz/dxを求めよ。
（5分）

$z = t^2$
$t = x + y$

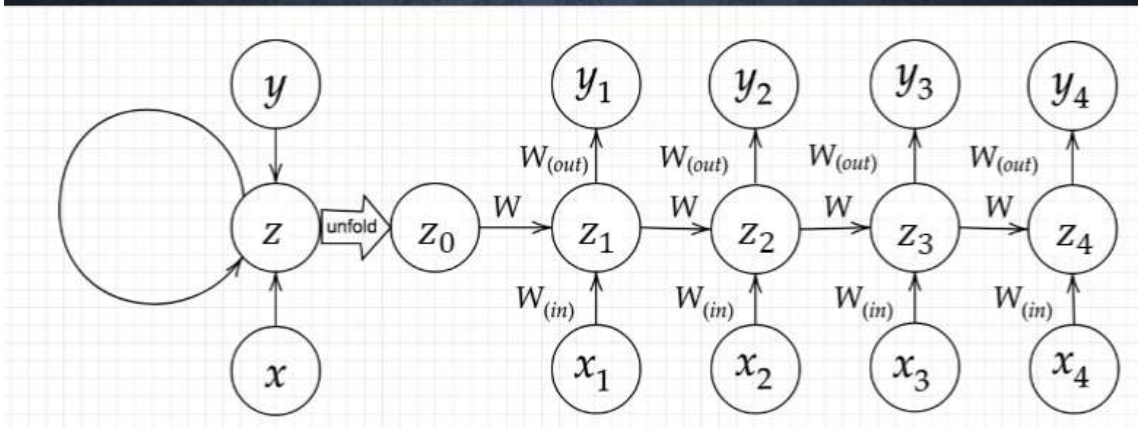
回答：dz/dt=2t

dt/dx=1

dz/dx=dz/dt dt/dx=2t*1=2(x + y)

確認テスト 3

下図のy_1をx・s_0・s_1・w_{in}・w・w_{out}を用いて数式で表せ。
※バイアスは任意の文字で定義せよ。
※また中間層の出力にシグモイド関数g(x)を作用させよ。
（7分）



回答：$y_1=g(w_{out}・s_1 + c)$

$s_1=f(w_{in}・x_1 + w・s_0 + b)$

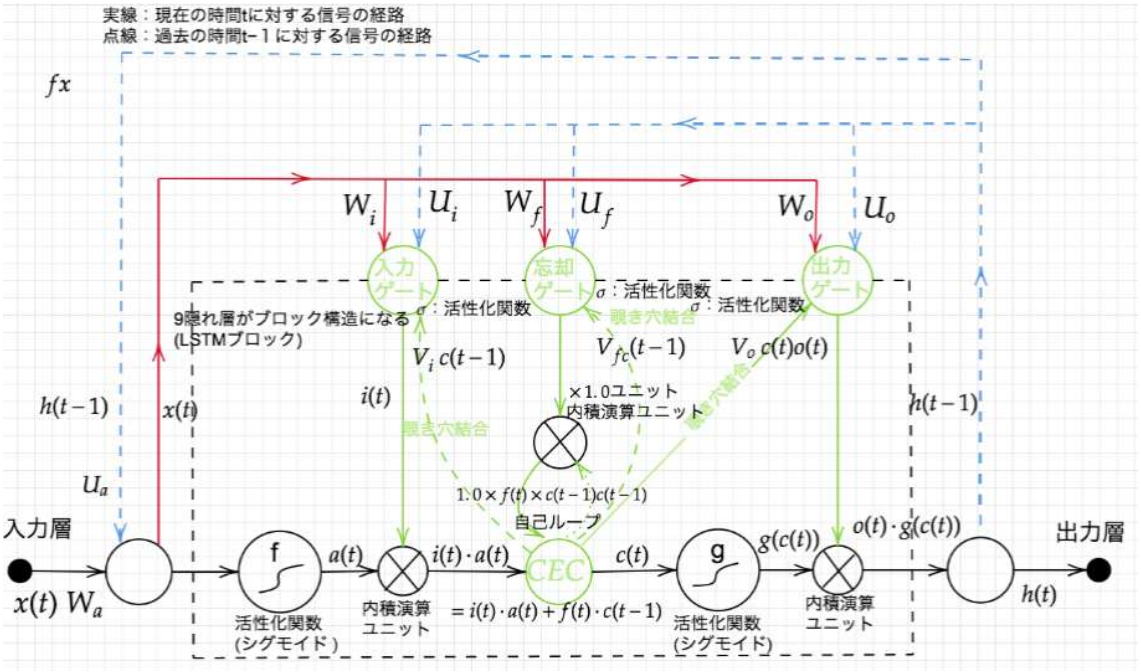
Section 2 ：ＬＳＴＭ

・ＲＮＮの課題の解決：長い時系列の学習が困難。これまでに触れた勾配消失の解決方法（ｔａｎｈ等の勾配消失問題に強い活性化関数を用いる、あるいは正則化を用いる等）とは別に、構造自体を変えて解決したものがＬＳＴＭ。

・勾配消失問題：誤差逆伝播法が下位層に進んで行くに連れて、勾配がどんどん緩やかになっていく。そのため、勾配降下法による更新では、下位層のパラメータはほとんど変わらず、訓練は最適解に収束しなくなる。（例）活性化関数: シグモイド関数

・勾配爆発：勾配が層を逆伝播するごとに指数関数的に大きくなっていく⇒（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）

・ＬＳＴＭの全体図



・ＣＥＣ（Constant Error Carousel）記憶機能だけを持つもの

勾配消失および勾配爆発の解決方法とする。

・ＣＥＣの課題：入力データについて、時間依存に関係なく一律である（ニューラルネットワークの学習特性が無いという事）

・入力ゲートと出力ゲートを追加する事で、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。（ＣＥＣの課題を解決）

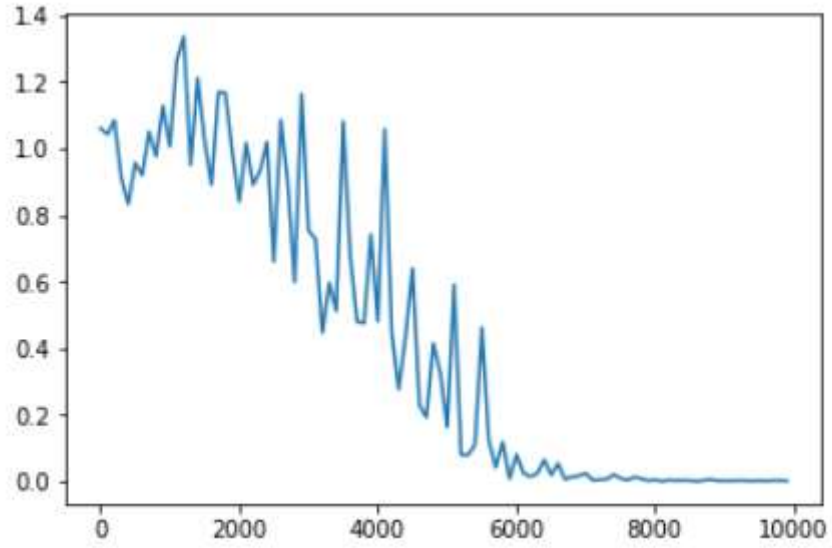
・ＬＳＴＭの現状⇒（ＣＥＣには過去の情報が全て保管されている。）

・ＬＳＴＭブロックの課題：過去の情報が要らなくなった場合、そのタイミングで情報を忘却する機能が必要⇒忘却ゲート

・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構造

覗き穴結合によって、ゲートが定誤差カルーセル（CEC。その活性化がセル状態である）へアクセスすることが可能となる

実行結果：



参照コード：

```
from google.colab import drive
drive.mount('/content/drive')
import sys
sys.path.append('/content/drive/My Drive/DNN_code_3_4')
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
```

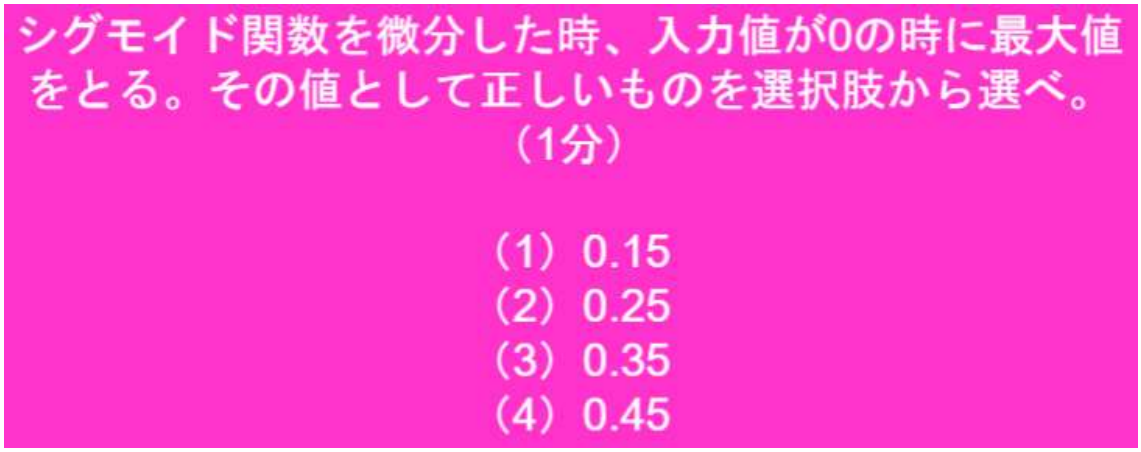
```
# 2進数の桁数
binary_dim = 8
# 最大値 + 1
largest_number = pow(2, binary_dim)
# largest_numberまで2進数を用意
binary = np.unpackbits(np.array([range(largest_number)],dtype=np.uint8).T,axis=1)

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

# 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
# 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.s
qrt(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] = 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))


        #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] = (np.dot(delta[:,t+1].T, W.T) + np.dot(delta_ou
t[:,t].T, W_out.T)) * 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])
```

```
#         delta[:,t] = (np.dot(delta[:,t+1].T, W.T) + np.dot(delta_
out[:,t].T, W_out.T)) * d_tanh(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()
```

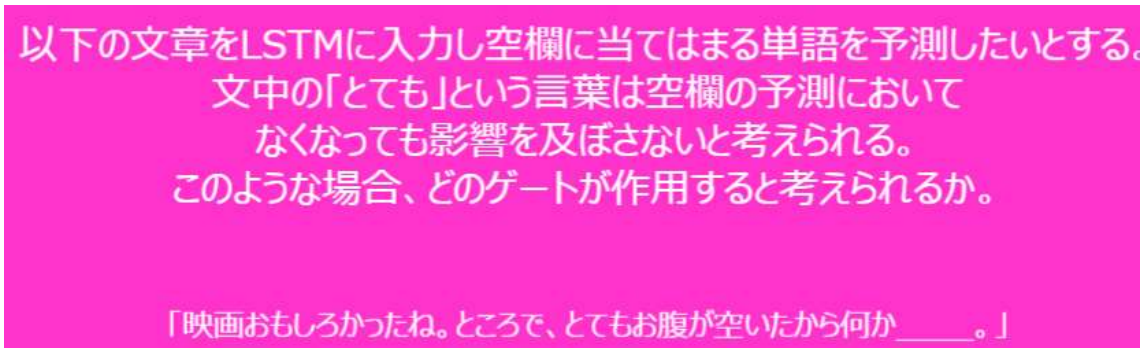
確認テスト１




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

def sigmoid_d(x):
  return (1-sigmoid(x)) * sigmoid(x)

sigmoid(0)=(1/(1+1))=0.5
sigmoid_d(0)=(1-0.5)*0.5=0.25
```

回答：(2)0.25

確認テスト２



回答：忘却ゲート

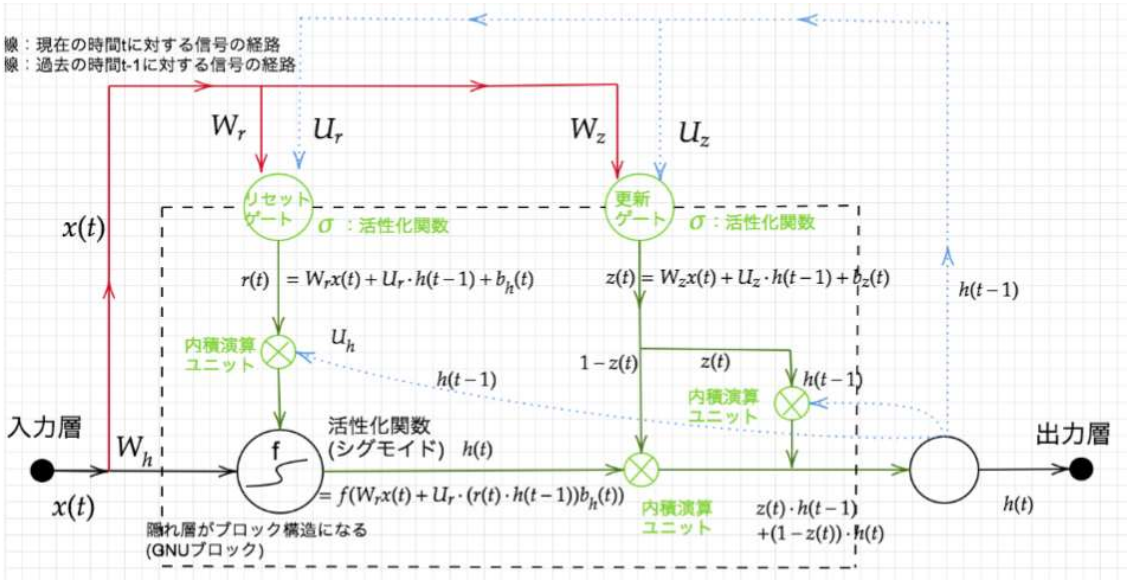
Section 3 ：ＧＲＵ

・ＧＲＵ（Gated recurrent unit）ゲート付き回帰型ユニット

・ＬＳＴＭの改良版：ＬＳＴＭでは、パラメータ数が多く、計算負荷が高くなる問題があった。

・そのため、ＧＲＵはパラメータを大幅に削減し（メリット：計算負荷が低い）、精度は同等またはそれ以上が望める様になった構造を持つ。

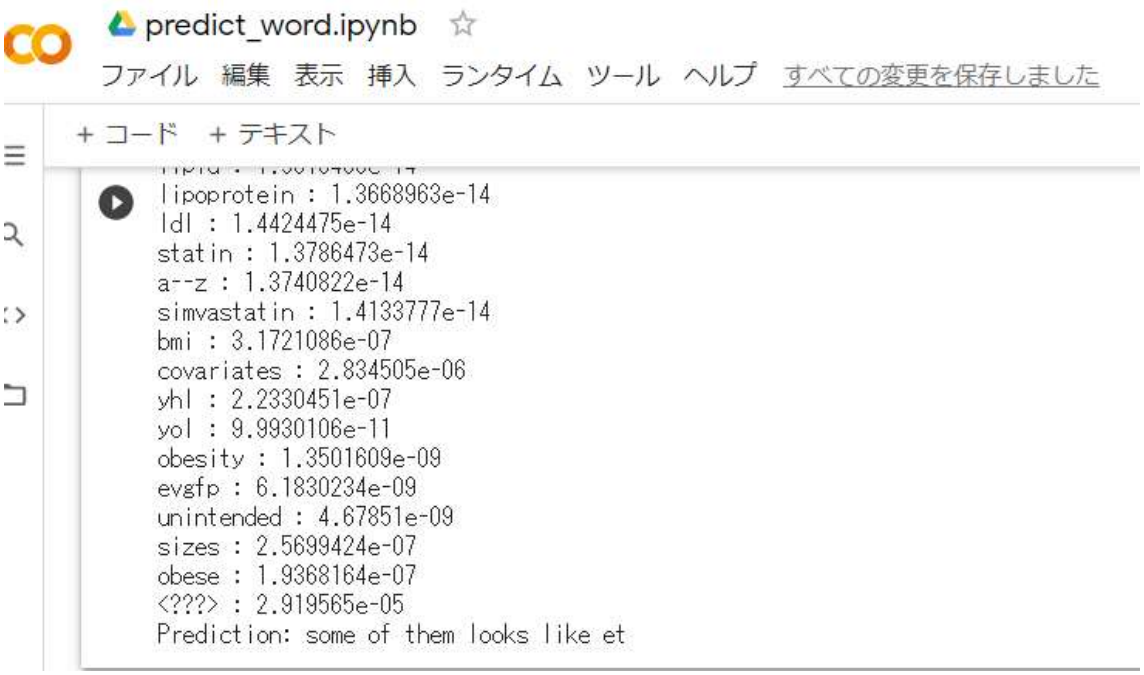
・GRU の全体像



実行結果：

```
# 保存したモデルを使って単語の予測をする
ln.predict("some of them looks like")
```

```
ストリーミング出力は最後の 5000 行に切り捨てられました。
beating : 1.3192031e-14
authentic : 1.4901815e-14
glow : 1.5293715e-14
oy : 1.4680216e-14
emotion : 1.4983698e-14
delight : 1.40041135e-14
nuclear : 1.4859978e-14
dropped : 1.4963249e-14
hiroshima : 1.3928354e-14
```



参照コード：

```
%tensorflow_version 1.x
from google.colab import drive
drive.mount('/content/drive')
import os
os.chdir('drive/My Drive/DNN_code/lesson_3/3_2_tf_languagemodel/')
import tensorflow as tf
import numpy as np
import re
import glob
import collections
import random
import pickle
import time
import datetime
import os

# logging levelを変更
tf.logging.set_verbosity(tf.logging.ERROR)

class Corpus:
```

```
    def __init__(self):
        self.unknown_word_symbol = "<???>" # 出現回数の少ない単語は未知語として定義しておく
        self.unknown_word_threshold = 3 # 未知語と定義する単語の出現回数の閾値
        self.corpus_file = "./corpus/**/*.txt"
        self.corpus_encoding = "utf-8"
        self.dictionary_filename = "./data_for_predict/word_dict.dic"
        self.chunk_size = 5
        self.load_dict()

        words = []
        for filename in glob.glob(self.corpus_file, recursive=True):
            with open(filename, "r", encoding=self.corpus_encoding) as f:

                # word breaking
                text = f.read()
                # 全ての文字を小文字に統一し、改行をスペースに変換
                text = text.lower().replace("¥n", " ")
                # 特定の文字以外の文字を空文字に置換する
                text = re.sub(r"[^a-z '¥-]", "", text)
                # 複数のスペースはスペース一文字に変換
                text = re.sub(r"[ ]+", " ", text)

                # 前処理: '-' で始まる単語は無視する
                words = [ word for word in text.split() if not word.startswith("-")]


        self.data_n = len(words) - self.chunk_size
        self.data = self.seq_to_matrix(words)

    def prepare_data(self):
        """
```

```
        訓練データとテストデータを準備する。
        data_n = ( text データの総単語数 ) - chunk_size
        input: (data_n, chunk_size, vocabulary_size)
        output: (data_n, vocabulary_size)
        """

        # 入力と出力の次元テンソルを準備
        all_input = np.zeros([self.chunk_size, self.vocabulary_size
, self.data_n])
        all_output = np.zeros([self.vocabulary_size, self.data_n])

        # 準備したテンソルに、コーパスの one-hot 表現(self.data) のデータを埋
めていく
        # i 番目から ( i + chunk_size - 1 ) 番目までの単語が1組の入力となる
        # このときの出力は ( i + chunk_size ) 番目の単語
        for i in range(self.data_n):
            all_output[:, i] = self.data[:, i + self.chunk_size] #
(i + chunk_size) 番目の単語の one-hot ベクトル
            for j in range(self.chunk_size):
                all_input[j, :, i] = self.data[:, i + self.chunk_si
ze - j - 1]

        # 後に使うデータ形式に合わせるために転置を取る
        all_input = all_input.transpose([2, 0, 1])
        all_output = all_output.transpose()

        # 訓練データ:テストデータを 4 : 1 に分割する
        training_num = ( self.data_n * 4 ) // 5
        return all_input[:training_num], all_output[:training_num],
 all_input[training_num:], all_output[training_num:]


    def build_dict(self):
        # コーパス全体を見て、単語の出現回数をカウントする
        counter = collections.Counter()
        for filename in glob.glob(self.corpus_file, recursive=True)
:
```

```
            with open(filename, "r", encoding=self.corpus_encoding)
 as f:

                # word breaking
                text = f.read()
                # 全ての文字を小文字に統一し、改行をスペースに変換
                text = text.lower().replace("¥n", " ")
                # 特定の文字以外の文字を空文字に置換する
                text = re.sub(r"[^a-z '¥-]", "", text)
                # 複数のスペースはスペース一文字に変換
                text = re.sub(r"[ ]+", " ", text)

                # 前処理: '-' で始まる単語は無視する
                words = [word for word in text.split() if not word.
startswith("-")]

                counter.update(words)

        # 出現頻度の低い単語を一つの記号にまとめる
        word_id = 0
        dictionary = {}
        for word, count in counter.items():
            if count <= self.unknown_word_threshold:
                continue

            dictionary[word] = word_id
            word_id += 1
        dictionary[self.unknown_word_symbol] = word_id

        print("総単語数:", len(dictionary))

        # 辞書を pickle を使って保存しておく
        with open(self.dictionary_filename, "wb") as f:
            pickle.dump(dictionary, f)
            print("Dictionary is saved to", self.dictionary_filenam
e)
```

```
        self.dictionary = dictionary

        print(self.dictionary)

    def load_dict(self):
        with open(self.dictionary_filename, "rb") as f:
            self.dictionary = pickle.load(f)
            self.vocabulary_size = len(self.dictionary)
            self.input_layer_size = len(self.dictionary)
            self.output_layer_size = len(self.dictionary)
            print("総単語数: ", self.input_layer_size)

    def get_word_id(self, word):
        # print(word)
        # print(self.dictionary)
        # print(self.unknown_word_symbol)
        # print(self.dictionary[self.unknown_word_symbol])
        # print(self.dictionary.get(word, self.dictionary[self.unkn
own_word_symbol]))
        return self.dictionary.get(word, self.dictionary[self.unkno
wn_word_symbol])

    # 入力された単語を one-hot ベクトルにする
    def to_one_hot(self, word):
        index = self.get_word_id(word)
        data = np.zeros(self.vocabulary_size)
        data[index] = 1
        return data

    def seq_to_matrix(self, seq):
        print(seq)
        data = np.array([self.to_one_hot(word) for word in seq]) #
(data_n, vocabulary_size)
        return data.transpose() # (vocabulary_size, data_n)
```

```
class Language:
    """
    input layer: self.vocabulary_size
    hidden layer: rnn_size = 30
    output layer: self.vocabulary_size
    """

    def __init__(self):
        self.corpus = Corpus()
        self.dictionary = self.corpus.dictionary
        self.vocabulary_size = len(self.dictionary) # 単語数
        self.input_layer_size = self.vocabulary_size # 入力層の数
        self.hidden_layer_size = 30 # 隠れ層の RNN ユニットの数
        self.output_layer_size = self.vocabulary_size # 出力層の数
        self.batch_size = 128 # バッチサイズ
        self.chunk_size = 5 # 展開するシーケンスの数。
c_0, c_1, ..., c_(chunk_size - 1) を入力し、c_(chunk_size) 番目の単語の
確率が出力される。
        self.learning_rate = 0.005 # 学習率
        self.epochs = 1000 # 学習するエポック数
        self.forget_bias = 1.0 # LSTM における忘却ゲートのバイアス
        self.model_filename = "./data_for_predict/predict_model.ckp
t"
        self.unknown_word_symbol = self.corpus.unknown_word_symbol

    def inference(self, input_data, initial_state):
        """
        :param input_data: (batch_size, chunk_size, vocabulary_size
) 次元のテンソル
        :param initial_state: (batch_size, hidden_layer_size) 次元の
行列
        :return:
        """
        # 重みとバイアスの初期化
        hidden_w = tf.Variable(tf.truncated_normal([self.input_laye
r_size, self.hidden_layer_size], stddev=0.01))
```

```
        hidden_b = tf.Variable(tf.ones([self.hidden_layer_size]))
        output_w = tf.Variable(tf.truncated_normal([self.hidden_lay
er_size, self.output_layer_size], stddev=0.01))
        output_b = tf.Variable(tf.ones([self.output_layer_size]))

        # BasicLSTMCell, BasicRNNCell は (batch_size, hidden_layer_
size) が chunk_size 数ぶんつながったリストを入力とする。
        # 現時点での入力データ
は (batch_size, chunk_size, input_layer_size) という3次元のテンソルなので
        # tf.transpose や tf.reshape などを駆使してテンソルのサイズを調整す
る。

        input_data = tf.transpose(input_data, [1, 0, 2]) # 転置。
(chunk_size, batch_size, vocabulary_size)
        input_data = tf.reshape(input_data, [-
1, self.input_layer_size]) # 変形。
(chunk_size * batch_size, input_layer_size)
        input_data = tf.matmul(input_data, hidden_w) + hidden_b #
重み W とバイアス B を適用。 (chunk_size, batch_size, hidden_layer_size)
        input_data = tf.split(input_data, self.chunk_size, 0) # リス
トに分割。chunk_size * (batch_size, hidden_layer_size)

        # RNN のセルを定義する。RNN Cell の他に LSTM のセルや GRU のセルなど
が利用できる。
        cell = tf.nn.rnn_cell.BasicRNNCell(self.hidden_layer_size)
        outputs, states = tf.nn.static_rnn(cell, input_data, initia
l_state=initial_state)

        # 最後に隠れ層から出力層につながる重みとバイアスを処理する
        # 最終的に softmax 関数で処理し、確率として解釈される。
        # softmax 関数はこの関数の外で定義する。
        output = tf.matmul(outputs[-1], output_w) + output_b

        return output

    def loss(self, logits, labels):
```

```
        cost = tf.reduce_mean(tf.nn.softmax_cross_entropy_with_logi
ts(logits=logits, labels=labels))

        return cost

    def training(self, cost):
        # 今回は最適化手法として Adam を選択する。
        # ここの AdamOptimizer の部分を変えることで、Adagrad, Adadelta などの他の最適化手法を選択することができる
        optimizer = tf.train.AdamOptimizer(learning_rate=self.learn
ing_rate).minimize(cost)

        return optimizer

    def train(self):
        # 変数などの用意
        input_data = tf.placeholder("float", [None, self.chunk_size
, self.input_layer_size])
        actual_labels = tf.placeholder("float", [None, self.output_
layer_size])
        initial_state = tf.placeholder("float", [None, self.hidden_
layer_size])

        prediction = self.inference(input_data, initial_state)
        cost = self.loss(prediction, actual_labels)
        optimizer = self.training(cost)
        correct = tf.equal(tf.argmax(prediction, 1), tf.argmax(actu
al_labels, 1))
        accuracy = tf.reduce_mean(tf.cast(correct, tf.float32))

        # TensorBoard で可視化するため、クロスエントロピーをサマリーに追加
        tf.summary.scalar("Cross entropy: ", cost)
        summary = tf.summary.merge_all()

        # 訓練・テストデータの用意
        # corpus = Corpus()
```

```
        trX, trY, teX, teY = self.corpus.prepare_data()
        training_num = trX.shape[0]

        # ログを保存するためのディレクトリ
        timestamp = time.time()
        dirname = datetime.datetime.fromtimestamp(timestamp).strfti
me("%Y%m%d%H%M%S")

        # ここから実際に学習を走らせる
        with tf.Session() as sess:
            sess.run(tf.global_variables_initializer())
            summary_writer = tf.summary.FileWriter("./log/" + dirna
me, sess.graph)

            # エポックを回す
            for epoch in range(self.epochs):
                step = 0
                epoch_loss = 0
                epoch_acc = 0

                # 訓練データをバッチサイズごとに分けて学習させ
る (= optimizer を走らせる)
                # エポックごとの損失関数の合計値や(訓練データに対する)精度も計算
しておく
                while (step + 1) * self.batch_size < training_num:
                    start_idx = step * self.batch_size
                    end_idx = (step + 1) * self.batch_size

                    batch_xs = trX[start_idx:end_idx, :, :]
                    batch_ys = trY[start_idx:end_idx, :]

                    _, c, a = sess.run([optimizer, cost, accuracy],
                                       feed_dict={input_data: batch
_xs,
                                                  actual_labels: ba
tch_ys,
```

```
                                                    initial_state: np
.zeros([self.batch_size, self.hidden_layer_size])
                                                    }
                                            )
                    epoch_loss += c
                    epoch_acc += a
                    step += 1

                # コンソールに損失関数の値や精度を出力しておく
                print("Epoch", epoch, "completed ouf of", self.epoc
hs, "-- loss:", epoch_loss, " -- accuracy:",
                      epoch_acc / step)

                # Epoch が終わるごとに TensorBoard 用に値を保存
                summary_str = sess.run(summary, feed_dict={input_da
ta: trX,
                                                           actual_l
abels: trY,
                                                           initial_
state: np.zeros(
                                                               [trX
.shape[0],
                                                                sel
f.hidden_layer_size]
                                                           )
                                                           }
                                            )
                summary_writer.add_summary(summary_str, epoch)
                summary_writer.flush()

            # 学習したモデルも保存しておく
            saver = tf.train.Saver()
            saver.save(sess, self.model_filename)

            # 最後にテストデータでの精度を計算して表示する
```

```
            a = sess.run(accuracy, feed_dict={input_data: teX, actual_labels: teY,
                                              initial_state: np.zeros([teX.shape[0], self.hidden_layer_size])})
            print("Accuracy on test:", a)


    def predict(self, seq):
        """
        文章を入力したときに次に来る単語を予測する
        :param seq: 予測したい単語の直前の文字列。chunk_size 以上の単語数が必要。
        :return:
        """

        # 最初に復元したい変数をすべて定義してしまいます
        tf.reset_default_graph()
        input_data = tf.placeholder("float", [None, self.chunk_size, self.input_layer_size])
        initial_state = tf.placeholder("float", [None, self.hidden_layer_size])
        prediction = tf.nn.softmax(self.inference(input_data, initial_state))
        predicted_labels = tf.argmax(prediction, 1)

        # 入力データの作成
        # seq を one-hot 表現に変換する。
        words = [word for word in seq.split() if not word.startswith("-")]
        x = np.zeros([1, self.chunk_size, self.input_layer_size])
        for i in range(self.chunk_size):
            word = seq[len(words) - self.chunk_size + i]
            index = self.dictionary.get(word, self.dictionary[self.unknown_word_symbol])
            x[0][i][index] = 1
        feed_dict = {
            input_data: x, # (1, chunk_size, vocabulary_size)
```

```
            initial_state: np.zeros([1, self.hidden_layer_size])
        }

        # tf.Session()を用意
        with tf.Session() as sess:
            # 保存したモデルをロードする。ロード前にすべての変数を用意しておく必要がある。
            saver = tf.train.Saver()
            saver.restore(sess, self.model_filename)

            # ロードしたモデルを使って予測結果を計算
            u, v = sess.run([prediction, predicted_labels], feed_dict=feed_dict)

            keys = list(self.dictionary.keys())


            # コンソールに文字ごとの確率を表示
            for i in range(self.vocabulary_size):
                c = self.unknown_word_symbol if i == (self.vocabulary_size - 1) else keys[i]
                print(c, ":", u[0][i])

            print("Prediction:", seq + " " + ("<???>" if v[0] == (self.vocabulary_size - 1) else keys[v[0]]))

        return u[0]


def build_dict():
    cp = Corpus()
    cp.build_dict()

if __name__ == "__main__":
    #build_dict()

    ln = Language()
```

```
# 学習するときに呼び出す
#ln.train()

# 保存したモデルを使って単語の予測をする
ln.predict("some of them looks like")
```

確認テスト１：

LSTMとCECが抱える課題について、それぞれ簡潔に述べよ。

回答：ＬＳＴＭは入力ゲート、出力ゲート、忘却ゲート、ＣＥＣ、それぞれ４つの部品を持つ事で構成されていた。

そのため、LSTM では、パラメータ数が多く、計算負荷が高くなる問題があった。

その中にあるＣＥＣであるが、問題点としてはニューラルネットワークの学習特性が無いということです。（そのために３つのゲートを周りに付け、学習機能を持たせている、というのがＣＥＣの根本的な動きになっている。）

確認テスト２：

LSTMとGRUの違いを簡潔に述べよ。

回答：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。LSTM よりパラメターが少ないため、ＬＳＴＭよりＧＲＵの方が計算量は少ない。

Section４：双方向ＲＮＮ

・双方向ＲＮＮ（Bidirectional RNN）双方向性再帰型ニューラルネットワーク

・過去の情報だけでなく、未来の情報を加味する事で、精度を向上させるためのモデル。例）文章の推敲や、機械翻訳等

・Bidirectional RNN は、中間層の出力を、未来への順伝播と過去への逆伝播の両方向に伝播するネットワークである。BRNN では、学習時に、過去と未来の情報の入力を必要とすることから、運用時も過去から未来までのすべての情報を入力してはじめて予測できるようになる。そのため、BRNN の応用範囲が限定される。例えば、DNA 塩基を k-mer ごとに区切れば、塩基配列解析に BRNN が使えるようになる。あるいは、1 文全体を入力して、文中にある誤字・脱字の検出などに応用されている。

実施結果

該当なし

確認テスト

該当なし

Section５：Ｓｅｑ２Ｓｅｑ

・Ｓｅｑ２ｓｅｑ（Sequence to sequence）⇒系列(Sequence)を入力として、系列を出力するもの。

・Ｓｅｑ２ｓｅｑはEncoder-Decoderモデルの一種を指します。

・Ｓｅｑ２ｓｅｑの具体的な用途とは機械対話や機械翻訳等に使用されています。

・Ｅｎｃｏｄｅｒ　ＲＮＮ：ユーザーがインプットしたテキストデータを、単語等のトークンに区切って渡す構造。

・Ｔａｋｉｎｇ：文章を単語等のトークン事に分割し、トークン毎のＩＤに分割する。

・Ｅｍｂｅｄｄｉｎｇ：ＩＤからそのトークンを表す分散表現ベクトルに変換。数字の並びが似かよっている事は似かよった意味を持つ単語という事になる ⇒ 単語の意味を抽出したベクトル。

・Ｅｎｃｏｄｅｒ　ＲＮＮ：ベクトルを順番にＲＮＮに入力していく。

・Ｅｎｃｏｄｅｒ　ＲＮＮ処理手順

1.Decoder RNN: Encoder RNN の final state (thought vector) から、各 token の生成確率を出力していきます 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。

・ＨＲＥＤ：文章の過去の文脈というものを何かしら取れるようにできないかという試み。Ｓｅｑ２ｓｅｑが単語に扱ったのに加えて、文脈自体（１文、１文も）同様に扱ってみたら良いのではないか。

・ＨＲＥＤとは：Ｓｅｑ２ｓｅｑ＋ＣｏｎｔｅｘｔＲＮＮ

ＣｏｎｔｅｘｔＲＮＮ：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クスト全体を表すベクトルに変換する構造。 ⇒ 過去の発話の履歴を加味した返答ができる。

・ＨＲＥＤの課題：HRED は確率的な多様性が字面にしかなく、会話の「流れ」のような多様性が無い。短いよくある答えを学ぶ傾向がある。例)「うん」「そうだね」等。

・「ＶＨＲＥＤ（Latent Variable Hierarchical. Recurrent Encoder-Decoder）」階層型潜在変数付きエンコーダ・デコーダ

・ＨＲＥＤHRED に、VAE の潜在変数の概念を追加したもの。

・オートエンコーダ：教師なし学習の一つ。そのため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。

入力データから潜在変数ｚに変換するニューラルネットワークを Encoder 逆に潜在変数ｚをインプットとして元画像を復元するニューラルネットワークを Decoder。次元削減のメリットがある。

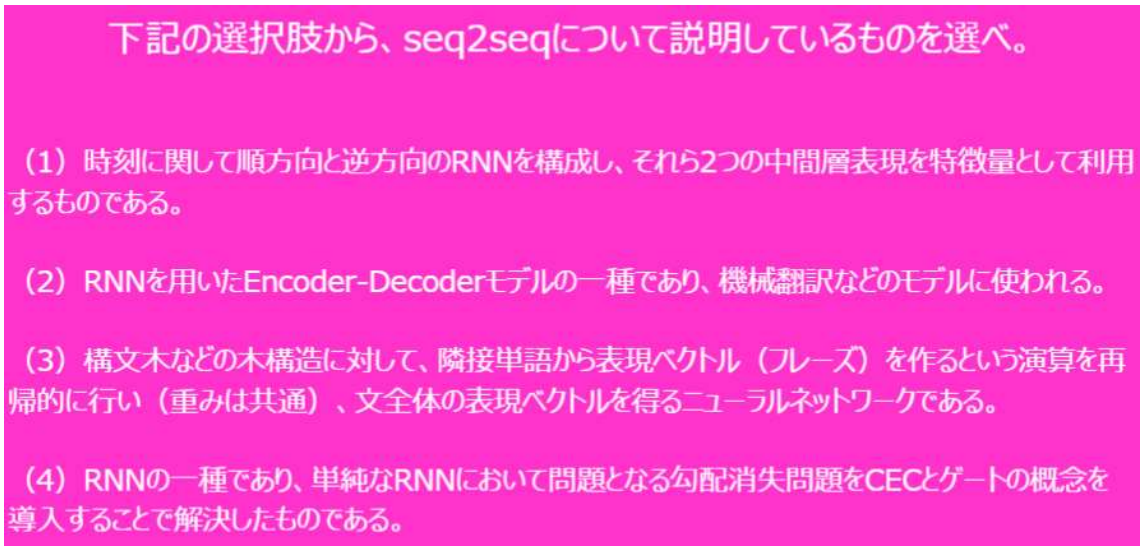
・ＶＡＥ（Variational Autoencoder）変分オートエンコーダー

確率分布に対するパラメーター最適化アルゴリズム。潜在変数ｚを正規化する（確率分布 z~N(0,1)）。より汎用性の高い特徴を掴む。

実施結果

該当なし

確認テスト１：



回答：(2)

確認テスト２：

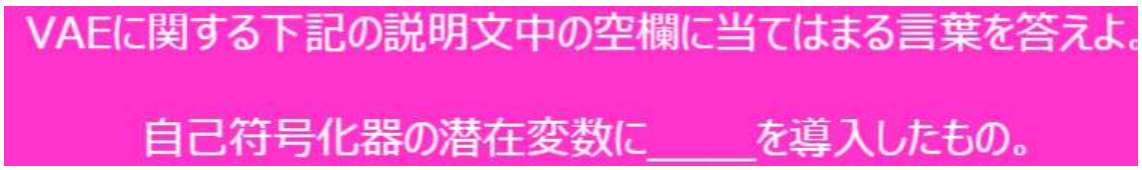
回答：Ｓｅｑ２ｓｅｑは１問１答しかできない（問に対して文脈も何もなく、ただ応答が行われ続ける）。

ＨＲＥＤは文章の過去の文脈というものを考慮できる。過去 n-1 個の発話から次の発話を生成する。Seq2Seq+ Context RNN

HRED は確率的な多様性が字面にしかなく、会話の「流れ」のような多様性が無い。
例）「うん」「そうだね」等。

ＶＨＲＥＤは文 HRED に、VAE の潜在変数の概念を追加したもの。HRED の課題を VAE の潜在変数の概念を追加することで解決した構造。

確認テスト３：



回答：確率分布

Section６：Ｗｏｒｄ２ｖｅｃ

・Word2vec は、 単語の埋め込みを生成するために使用される一連のモデル群である。学習データからボキャブラリを作成。

・課題：RNN では、単語のような可変長の文字列を NN に与えることはできない。

⇒固定長形式で単語を表す必要がある。

・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量で実現可能にした。

Section７：Ａｔｔｅｎｔｉｏｎ　Ｍｅｃｈａｎｉｓｍ

・課題：課題:seq2seq の問題は長い文章への対応が難しいです。seq2seq では、2 単語でも、100 単語でも、固定次元ベクトルの中に入力しなければならない。

・解決策：Ａｔｔｅｎｔｉｏｎ Ｍｅｃｈａｎｉｓｍ。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仕組み。１文の中で特に重要な単語というのを自力で見つけられるようにする機構。途中の中間層の情報量が一定でも重要な情報だけ拾い集める。

・近年特に性能が上がっている自然言語処理のモデルは全部Ａｔｔｅｎｔｉｏｎ Ｍｅｃｈａｎｉｓｍである程、強力である。

確認テスト：

RNNとword2vec、seq2seqとAttentionの違いを簡潔に述べよ。

回答：ＲＮＮは時系列データに対応可能な、ニューラルネットワークです。

ｗｏｒｄ２ｖｅｃは単語の分散表現ベクトルを得る手法です。

ｓｅｑ２ｓｅｑは１つの時系列データから別の時系列データを得るネットワークです。

Ａｔｔｅｎｔｉｏｎ Ｍｅｃｈａｎｉｓｍ は時系列データの中身に対して関連性に重みを付ける手法になります。

深層学習後編(day4)については以下６つの科目でレポートする

Section 1 ：強化学習

・「強化学習」
　強化学習と通常の教師あり、教師なし学習と違います。教師なし、あり学習では、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事が目標。試行錯誤を繰り返し数値化された報酬信号を最大にするために何をするべきかを学習していく教師なしの機械学習である。
・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）をもとに行動を決定する原理を改善していく仕組みです。
・探索と利用のトレードオフ：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。
・強化学習で学習するエージェント：方策関数: π(s,a)、行動価値関数:Q(s,a)
・強化学習の目標⇒得られる報酬の総量を最大化することです。
・強化学習の歴史
　計算速度の進展により大規模な状態をもつ場合の、強化学習を可能としつつある。
　関数近似法（価値関数や方策関数を関数近似する手法のこと）とQ学習（行動価値関数を、行動する毎に更新する事により学習を進める方法）を組み合わせる手法の登場。
・価値関数：価値を表す関数としては、状態価値関数と行動価値関数の２種類がある。
　ある状態の価値に注目する場合は、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は、行動価値関数
・方策関数：エージェントが取る行動を決定するための関数。
・方策勾配法について
　方策反復法：方策をモデル化して最適化する手法 ⇒ 方策勾配法
　定義方法：平均報酬、割引報酬和に対応して、行動価値関数の定義を行い、方策勾配定理が成立する。

実施結果

該当なし

確認テスト

該当なし

Section２：Ａｌｐｈａ Ｇｏ

・Alpha Go (Lee)

Google DeepMind によって開発されたコンピュータ囲碁プログラムである。

Alpha Go (Lee)の PolicyNet（方策関数)、ValueNet（価値関数）は共に畳み込みニューラルネットワーク。

PolicyNet（SoftMax Layer）出力は１９ｘ１９マスの着手予想確率が出力される。

ValueNet（TanH 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

教師あり学習による RollOutPolicy(NN ではなく線形の方策関数探索中に高速に着手確率を出すために使用される)と PolicyNet の学習。

強化学習による PolicyNet の学習。

強化学習による ValueNet の学習。

PolicyNet の強化学習

ValueNet の学習(PolicyNet を使用して対局シミュレーションを行い、その結果の勝敗を教師として学習)

PolicyNet と RollOutPolicy の教師あり学習。

モンテカルロ木探索（強化学習の学習方法）（価値関数を学習させる時に用いる方法。価値関数をどういうふうに更新するか。）

・AlphaGo(Lee) と AlphaGoZero の違い

教師あり学習を一切行わず、強化学習のみで作成。

特徴入力からヒューリスティックな要素を排除し、石の配置のみにした。

PolicyNet と ValueNet を１つのネットワークに統合した。

Residual Net（後述）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。

・Residual Network

ネットワークにショートカット構造を追加して、勾配の爆発、消失を抑える効果を狙ったもの Residula Network を使うことにより、100 層を超えるネットワークでの安定した学習が可能となった。

基本構造は

Convolution→BatchNorm→ReLU→Convolution→BatchNorm→Add→ReLU の Block を１単位にして積み重ねる形となるまた、Resisual Network を使うことにより層数の違う Network のアンサンブル効果が得られているという説もある。

ResidualNetwork の性能⇒ResidualNetwork を ImageNet の画像分類に使ったときのネットワーク構造、およびエラーレートである。18、34Layer のものは基本形、50、101、152Layer のものは BottleNeck 構造である。性能は従来の VGG-16 と比較して Top5 で 4%程度ものエラー率の低減に成功している。

・Alpha Go Zero の学習法
Alpha Go の学習は自己対局による教師データの作成、学習、ネットワークの更新の 3 ステップで構成される。

実施結果

該当なし

確認テスト

該当なし

Section 3 ：軽量化・高速化技術

・分散深層学習：データ並列化、モデル並列化、GPU

・データ並列化：データを分割し、各ワーカーごとに計算させる。

同期型：同期型のパラメータ更新の流れ。各ワーカーが計算終わるのを待ち、全ワーカーの勾配が出たところで勾配の平均を計算し、親モデルのパラメータを更新する。

非同期型：非同期型のパラメータ更新の流れ。各ワーカーはお互いの計算を待たず、各子モデル毎に更新を行う。学習が終わった子モデルはパラメータサーバに Push される。新たに学習を始める時は、パラメータサーバから Pop したモデルに対して学習していく。

・同期型と非同期型の比較

処理のスピードは、お互いのワーカーの計算を待たない非同期型の方が早い。

非同期型は最新のモデルのパラメータを利用できないので、学習が不安定になりやすい。-> Stale Gradient Problem

現在は同期型の方が精度が良いことが多いので、主流となっている。

・モデル並列

親モデルを各ワーカーに分割し、それぞれのモデルを学習させる。全てのデータで学習が終わった後で、一つのモデルに復元。

モデルが大きい時はモデル並列化を、データが大きい時はデータ並列化をすると良い。

モデルのパラメータ数が多いほど、スピードアップの効率も向上する。

ある程度モデルが大きければ、分割した時の効果が目に見えるように出る。

・参照論文

Large Scale Distributed Deep Networks（Google 社が 2016 年に出した論文。Tensorflow の前身といわれている。）

並列コンピューティングを用いることで大規模なネットワークを高速に学習させる仕組みを提案。

主にモデル並列とデータ並列(非同期型)の提案をしている。

・GPU による高速化

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

・モデルの軽量化まとめ

量子化(Quantization)：重みの精度を下げることにより計算の高速化と省メモリ化を行う技術。

蒸留(Distillaion)：複雑で精度の良い教師モデルから軽量な生徒モデルを効率よく学習を行う技術。

プルーニング(Pruning)：寄与の少ないニューロンをモデルから削減し高速化と省メモ

リ化を行う技術。

実施結果

該当なし

確認テスト

該当なし

Section４：応用モデル

・MobileNet 概要

オンデバイスや組み込みアプリケーションの限られたリソースを意識しながら、効率的に精度を最大化することに重点をおいて設計された。

・MobileNet（画像認識のモデルで軽量化したモデル）

２０１７年に精度は最高レベルに達しており、そこから先は軽くて性能が良いモデルを研究する事が多くなった。MobileNet はその先駆け。あまり計算量を増やさずに比較的高性能である事が目標。

一般的な畳み込みレイヤ：ストライド１でパディングを適用した場合の畳み込み計算の計算量。

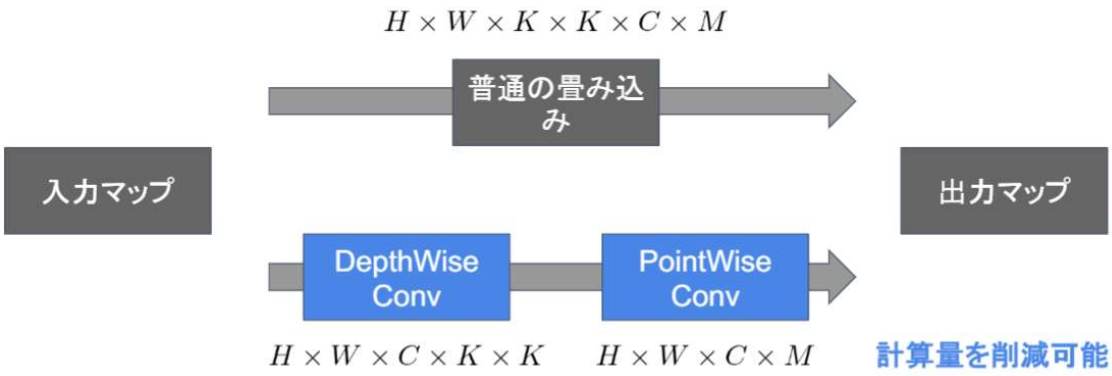
MobileNet はここから、これをどうにか減らして行こうという試み。

Depthwise Convolution と Pointwise Convolution の組み合わせで軽量化を実現。

Depthwise Convolution：入力マップのチャネルごとに畳み込みを実施 ⇒ 通常の畳み込みカーネルは全ての層にかかっていることを考えると計算量が大幅に削減。

Pointwise Convolution：1 x 1 conv とも呼ばれる(正確には 1 x 1 x c) ⇒ 計算量を大幅に削減可能。

MobileNet イメージ図



・DenseNet（画像認識のネットワーク）

DenseBlock

出力層に前の層の入力を足し合わせる。(層間の情報の伝達を最大にするために全ての同特徴量サイズの層を結合する)

DenseBlock 内の各ブロック毎に k 個ずつ特徴マップのチャネル数が増加していく時、k を成長率(Growth Rate)と呼ぶ。

ここで k はハイパーパラメータである。

k が大きくなる程ネットワークが大きくなるため、小さな整数に設定するのが良い。

Transition Layer

ＣＮＮでは中間層でチャネルサイズを変更する。

特徴マップのサイズを変更し、ダウンサンプリングを行うため、Transition Layer と呼ばれる層で Dence block をつなぐ。

プーリングは特徴マップのサイズを削減、各ブロック内で特徴マップのサイズは一致。

DenseNet と ResNet の違い

DenseBlock では前方の各層からの出力全てが後方の層への入力として用いられる。

RessidualBlock では前 1 層の入力のみ後方の層へ入力。

・Batch Norm Layer

Batch Norm の問題点：Batch Size が小さい条件下では、学習が収束しないことがあり、代わりに Layer Normalization などの正規化手法が使われることが多い。

レイヤー間を流れるデータの分布を、ミニバッチ単位で平均が 0・分散が 1 になるように正規化〇Batch Normalization はニューラルネットワークにおいて学習時間の短縮や初期値への依存低減、過学習の抑制など効果がある。

・Batch Norm

Batch Norm：ミニバッチに含まれる sample の同一チャネルが同一分布に従うよう正規化

Layer Norm：それぞれの sample の全ての pixels が同一分布に従うよう正規化

Instance Norm：各 sample の各チャンネルも同一分布に従うよう正規化

ミニバッチのサイズの影響を受けるのはミニバッチ数で正規化を行っているものだけなため、Batch Norm のみがミニバッチのサイズの影響を受ける。

・Wavenet

Aaron van den Oord et. al., 2016 らにより提案

生の音声波形を生成する深層学習モデル

Pixel CNN(高解像度の画像を精密に生成できる手法)を音声に応用したもの

単純な Convolution layer と比べて下記利点ある

パラメータ数に対する受容野が広い

受容野あたりのパラメータ数が多い

学習時に並列計算が行える

推論時に並列計算が行える

実施結果

該当なし

確認テスト 1

- **MobileNetのアーキテクチャ**
  - Depthwise Separable Convolutionという手法を用いて計算量を削減している。通常の畳込みが空間方向とチャネル方向の計算を同時に行うのに対して、Depthwise Separable ConvolutionではそれらをDepthwise ConvolutionとPointwise Convolutionと呼ばれる演算によって個別に行う。
  - Depthwise Convolition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ぞれ用いて計算を行うため、その計算量は（い）となる。
  - 次にDepthwise Convolutionの出力をPointwise Convolution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$1 \times 1 \times M$サイズの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（う）となる。

回答：

（い）

出力マップの計算量＝H x W x C x K x K

（う）

出力マップの計算量＝H x W x C x M

確認テスト 2

- 深層学習を用いて結合確率を学習する際に、効率的に学習が行えるアーキテクチャを提案したことが WaveNet の大きな貢献の 1 つである。
  提案された新しい Convolution 型アーキテクチャは
  （あ）と呼ばれ、結合確率を効率的に学習できるようになっている。
  - **Dilated causal convolution**
  - Depthwise separable convolution
  - Pointwise convolution
  - Deconvolution

回答：Dilated causal convolution

確認テスト３

- （あ）を用いた際の大きな利点は、単純な Convolution layer と比べて（い）ことである。
    - **パラメータ数に対する受容野が広い**
    - 受容野あたりのパラメータ数が多い
    - 学習時に並列計算が行える
    - 推論時に並列計算が行える

回答：パラメータ数に対する受容野が広い

Wavenet の工夫というのは、パラメータは既存の畳み込みの手法と同じくらいの数で作りたいのだけども、より長い時間的範囲を上手く使いたいという目標を持っていました。そこでスキップを上手くかませる事でより広い時間の情報を出力時に使用する事ができるようになります。

Section５：Ｔｒａｎｓｆｏｒｍｅｒ

・Transformer は RNN や CNN を使用せず、Attention のみを用いる Seq2Seq モデルです。並列計算が可能なため RNN に比べて計算が高速な上、Self-Attention と呼ばれる機構を用いることにより、局所的な位置しか参照できない CNN と異なり、系列内の任意の位置の情報を参照することを可能にしています。

・Self-Attention(自己注意機構)

データの流れ方自体を学習し決定するような方法である。もともと RNN 向けに提案されたが、CNN など他のニューラルネットワークにも利用されている。自己注意機構は強力であり機械翻訳、質問応答、画像生成など、多くの問題で最高精度を達成している。

ニューラル機械翻訳の問題点：長さに弱い。

翻訳元の文の内容をひとつのベクトルで表現する。文長が長くなると表現力が足りなくなる。

上の問題意識から生まれた「Attention（注意機構）(Bahdanau et al., 2015)」

情報量が多くなってきた時に何に注意を払って何に注意を払わないで良いかというのを学習的に決定していく、という機構が発明された。

Attention は何をしているのか

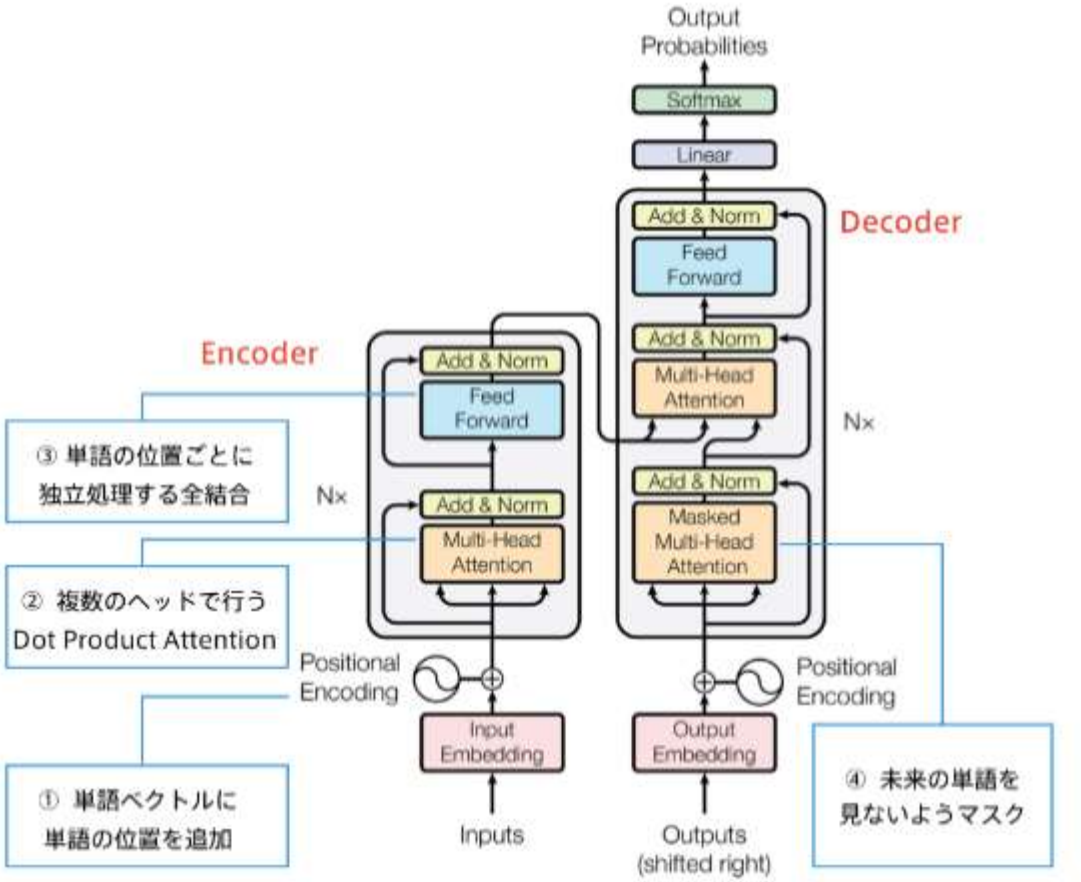
Attention は辞書オブジェクト：query(検索クエリ)に一致する Key を索引し、対応する value を取り出す操作であると見做す事ができるこれは辞書オブジェクトの機能と同じである。

・Transformer(Vaswani et al., 2017)

Attention is all you need

2017 年 6 月に登場で必要なのは Attention だけで RNN モデルを使用しない。

・Transformer 主要モジュール(構造図)



RNN を使用していないので文字の情報を保存できない ⇒ 単語ベクトルに単語の位置を付加。

大きく Encoder と Decoder に分かれていて、どちらも Self-Attention を使用しています。(Attention と言われていても、Transformer で使用されているものと、もともと Attention が発明された時に使用されていたものは別ものである。ソース・ターゲット注意機構。自己注意機構。)

Self-Attention のイメージとしては、CNN が 1 番良い。文脈を反映した自己表現を得られる。CNN はあるピクセルを中心にした時に、その周囲の情報を Convolution して抽象化した情報が出て来る。Attention もそれに近い事をしている。Self-Attention と CNN の対応関係はよく言われている事である。数式は違うが、どちらも文脈依存の Convolution である。2 つの違いは、CNN はあくまでウィンドウサイズがあるので、決められた範囲の Convolution しかおこなわないが、Self-Attention は Input された全ての情報からそれぞれを定義していくので、ウィンドウサイズが文の長さの Convolution と考えられる。

・Transformer-Encoder

系列をインプットして、その系列が位置情報を失わないまま Self-Attention 層を流れて行き、内部状態に変換される。

・Feed-Forward Networks
全結合層（位置情報を保持したまま順伝播させる）
使用目的としては線形変換をかける層で、最終的にアウトプットの次元をそろえないといけないのでマトリックスを整えるための全結合層。
・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 個のヘッドを使用し、8 個の Scaled Dot-Product Attention の出力を連結して。それぞれのヘッドが異なる種類の情報を収集。
・Decoder
Encoder と同じく 6 層で、各層で二種類の注意機構で、注意機構の仕組みは Encoder とほぼ同じ。
・Position Encoding
RNN を用いないので単語別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。
・Attention の可視化
注意状況を確認すると言語構造を捉えていることが多い。

実施結果 Seq2Seq

```
# beam decode
outputs, scores = beam_model(batch_X, lengths_X, max_length=20)
# scoreの良い順にソート
outputs, scores = zip(*sorted(zip(outputs, scores), key=lambda x: -x[1]))
for o, output in enumerate(outputs):
    output_sentence = ' '.join(ids_to_sentence(vocab_Y, output))
    print('out{}: {}'.format(o+1, output_sentence))
```

```
src: show your own business .
tgt: 自分 の 事 を しろ 。
out: <UNK> <UNK> <UNK> <UNK> <UNK> <UNK> <UNK> <UNK> <UNK> <UNK>
```

参照コード：

```
from os import path
from wheel.pep425tags import get_abbr_impl, get_impl_ver, get_abi_tag
platform = '{}{}-{}'.format(get_abbr_impl(), get_impl_ver(), get_abi_tag())
```

```
accelerator = 'cu80' if path.exists('/opt/bin/nvidia-
smi') else 'cpu'

!pip install -
q http://download.pytorch.org/whl/{accelerator}/torch-0.4.0-
{platform}-linux_x86_64.whl torchvision
import torch
print(torch.__version__)
print(torch.cuda.is_available())
 wget https://www.dropbox.com/s/9narw5x4uizmehh/utils.py
! mkdir images data

# data 取得
! wget https://www.dropbox.com/s/o4kyc52a8we25wy/dev.en -P data/
! wget https://www.dropbox.com/s/kdgskm5hzg6znuc/dev.ja -P data/
! wget https://www.dropbox.com/s/gyyx4gohv9v65uh/test.en -P data/
! wget https://www.dropbox.com/s/hotxwbgoe2n013k/test.ja -P data/
! wget https://www.dropbox.com/s/5lsftkmb20ay9e1/train.en -P data/
! wget https://www.dropbox.com/s/ak53qirssci6f1j/train.ja -P data/
import random
import numpy as np
from sklearn.model_selection import train_test_split
from sklearn.utils import shuffle
from nltk import bleu_score

import torch
import torch.nn as nn
import torch.nn.functional as F
import torch.optim as optim
from torch.nn.utils.rnn import pad_packed_sequence, pack_padded_seq
uence
from utils import Vocab

# デバイスの設定
device = torch.device("cuda" if torch.cuda.is_available() else "cpu
")
```

```
torch.manual_seed(1)
random_state = 42

print(torch.__version__)
def load_data(file_path):
    # テキストファイルからデータを読み込むメソッド
    data = []
    for line in open(file_path, encoding='utf-8'):
        words = line.strip().split()  # スペースで単語を分割
        data.append(words)
    return data
# 訓練データと検証データに分割
train_X, valid_X, train_Y, valid_Y = train_test_split(train_X, trai
n_Y, test_size=0.2, random_state=random_state)
# まず特殊トークンを定義しておく
PAD_TOKEN = '<PAD>'  # バッチ処理の際に、短い系列の末尾を埋めるために使
う（Padding）
BOS_TOKEN = '<S>'  # 系列の始まりを表す（Beggining of sentence）
EOS_TOKEN = '</S>'  # 系列の終わりを表す（End of sentence）
UNK_TOKEN = '<UNK>'  # 語彙に存在しない単語を表す（Unknown）
PAD = 0
BOS = 1
EOS = 2
UNK = 3
MIN_COUNT = 2  # 語彙に含める単語の最低出現回数 再提出現回数に満たない単語は
UNK に置き換えられる

# 単語を ID に変換する辞書の初期値を設定
word2id = {
    PAD_TOKEN: PAD,
    BOS_TOKEN: BOS,
    EOS_TOKEN: EOS,
    UNK_TOKEN: UNK,
    }
```

```
# 単語辞書を作成
vocab_X = Vocab(word2id=word2id)
vocab_Y = Vocab(word2id=word2id)
vocab_X.build_vocab(train_X, min_count=MIN_COUNT)
vocab_Y.build_vocab(train_Y, min_count=MIN_COUNT)
def sentence_to_ids(vocab, sentence):
    # 単語(str)のリストをID(int)のリストに変換する関数
    ids = [vocab.word2id.get(word, UNK) for word in sentence]
    ids += [EOS]  # EOSを加える
    return ids
def pad_seq(seq, max_length):
    # 系列(seq)が指定の文長(max_length)になるように末尾をパディングする
    res = seq + [PAD for i in range(max_length - len(seq))]
    return res


class DataLoader(object):

    def __init__(self, X, Y, batch_size, shuffle=False):
        """
        :param X: list, 入力言語の文章(単語IDのリスト)のリスト
        :param Y: list, 出力言語の文章(単語IDのリスト)のリスト
        :param batch_size: int, バッチサイズ
        :param shuffle: bool, サンプルの順番をシャッフルするか否か
        """
        self.data = list(zip(X, Y))
        self.batch_size = batch_size
        self.shuffle = shuffle
        self.start_index = 0

        self.reset()

    def reset(self):
        if self.shuffle:  # サンプルの順番をシャッフルする
            self.data = shuffle(self.data, random_state=random_stat
e)
```

```
        self.start_index = 0  # ポインタの位置を初期化する

    def __iter__(self):
        return self

    def __next__(self):
        # ポインタが最後まで到達したら初期化する
        if self.start_index >= len(self.data):
            self.reset()
            raise StopIteration()

        # バッチを取得
        seqs_X, seqs_Y = zip(*self.data[self.start_index:self.start
_index+self.batch_size])
        # 入力系列 seqs_X の文章の長さ順(降順)に系列ペアをソートする
        seq_pairs = sorted(zip(seqs_X, seqs_Y), key=lambda p: len(p
[0]), reverse=True)
        seqs_X, seqs_Y = zip(*seq_pairs)
        # 短い系列の末尾をパディングする
        lengths_X = [len(s) for s in seqs_X]  # 後述の Encoder の
pack_padded_sequence でも用いる
        lengths_Y = [len(s) for s in seqs_Y]
        max_length_X = max(lengths_X)
        max_length_Y = max(lengths_Y)
        padded_X = [pad_seq(s, max_length_X) for s in seqs_X]
        padded_Y = [pad_seq(s, max_length_Y) for s in seqs_Y]
        # tensor に変換し、転置する
        batch_X = torch.tensor(padded_X, dtype=torch.long, device=d
evice).transpose(0, 1)
        batch_Y = torch.tensor(padded_Y, dtype=torch.long, device=d
evice).transpose(0, 1)

        # ポインタを更新する
        self.start_index += self.batch_size

        return batch_X, batch_Y, lengths_X
```

```
# 系列長がそれぞれ 4,3,2 の 3 つのサンプルからなるバッチを作成
batch = [[1,2,3,4], [5,6,7], [8,9]]
lengths = [len(sample) for sample in batch]
print('各サンプルの系列長:', lengths)
print()

# 最大系列長に合うように各サンプルを padding
_max_length = max(lengths)
padded = torch.tensor([pad_seq(sample, _max_length) for sample in b
atch])
print('padding されたテンソル:¥n', padded)
padded = padded.transpose(0,1) # (max_length, batch_size)に転置
print('padding & 転置されたテンソル:¥n', padded)
print('padding & 転置されたテンソルのサイズ:¥n', padded.size())
print()
# PackedSequence に変換(テンソルを RNN に入力する前に適用する)
packed = pack_padded_sequence(padded, lengths=lengths) # 各サンプルの
系列長も与える
print('PackedSequence のインスタンス:¥n', packed) # テンソルの PAD 以外の値
(data)と各時刻で計算が必要な(=PAD に到達していない)バッチの数(batch_sizes)を有
するインスタンス
print()
# PackedSequence のインスタンスを RNN に入力する(ここでは省略)
output = packed

# テンソルに戻す(RNN の出力に対して適用する)
output, _length = pad_packed_sequence(output)  # PAD を含む元のテンソル
と各サンプルの系列長を返す
print('PAD を含む元のテンソル:¥n', output)
print('各サンプルの系列長:', _length)
class Encoder(nn.Module):
    def __init__(self, input_size, hidden_size):
        """
        :param input_size: int, 入力言語の語彙数
        :param hidden_size: int, 隠れ層のユニット数
        """
```

```
        super(Encoder, self).__init__()
        self.hidden_size = hidden_size

        self.embedding = nn.Embedding(input_size, hidden_size, padd
ing_idx=PAD)
        self.gru = nn.GRU(hidden_size, hidden_size)

    def forward(self, seqs, input_lengths, hidden=None):
        """
        :param seqs: tensor, 入力のバッ
チ, size=(max_length, batch_size)
        :param input_lengths: 入力のバッチの各サンプルの文長
        :param hidden: tensor, 隠れ状態の初期値, None の場合は 0 で初期化さ
れる
        :return output: tensor, Encoder の出
力, size=(max_length, batch_size, hidden_size)
        :return hidden: tensor, Encoder の隠れ状
態, size=(1, batch_size, hidden_size)
        """
        emb = self.embedding(seqs) # seqs はパディング済み
        packed = pack_padded_sequence(emb, input_lengths) # PackedS
equence オブジェクトに変換
        output, hidden = self.gru(packed, hidden)
        output, _ = pad_packed_sequence(output)
        return output, hidden
class Decoder(nn.Module):
    def __init__(self, hidden_size, output_size):
        """
        :param hidden_size: int, 隠れ層のユニット数
        :param output_size: int, 出力言語の語彙数
        :param dropout: float, ドロップアウト率
        """
        super(Decoder, self).__init__()
        self.hidden_size = hidden_size
        self.output_size = output_size
```

```
        self.embedding = nn.Embedding(output_size, hidden_size, pad
ding_idx=PAD)
        self.gru = nn.GRU(hidden_size, hidden_size)
        self.out = nn.Linear(hidden_size, output_size)

    def forward(self, seqs, hidden):
        """
        :param seqs: tensor, 入力のバッチ, size=(1, batch_size)
        :param hidden: tensor, 隠れ状態の初期値, None の場合は 0 で初期化さ
れる
        :return output: tensor, Decoder の出
力, size=(1, batch_size, output_size)
        :return hidden: tensor, Decoder の隠れ状
態, size=(1, batch_size, hidden_size)
        """
        emb = self.embedding(seqs)
        output, hidden = self.gru(emb, hidden)
        output = self.out(output)
        return output, hidden
class EncoderDecoder(nn.Module):
    """Encoder と Decoder の処理をまとめる"""
    def __init__(self, input_size, output_size, hidden_size):
        """
        :param input_size: int, 入力言語の語彙数
        :param output_size: int, 出力言語の語彙数
        :param hidden_size: int, 隠れ層のユニット数
        """
        super(EncoderDecoder, self).__init__()
        self.encoder = Encoder(input_size, hidden_size)
        self.decoder = Decoder(hidden_size, output_size)

    def forward(self, batch_X, lengths_X, max_length, batch_Y=None,
 use_teacher_forcing=False):
        """
        :param batch_X: tensor, 入力系列のバッ
チ, size=(max_length, batch_size)
```

```
        :param lengths_X: list, 入力系列のバッチ内の各サンプルの文長
        :param max_length: int, Decoder の最大文長
        :param batch_Y: tensor, Decoder で用いるターゲット系列
        :param use_teacher_forcing: Decoder でターゲット系列を入力とするフラグ
        :return decoder_outputs: tensor, Decoder の出力,
            size=(max_length, batch_size, self.decoder.output_size)
        """
        # encoder に系列を入力（複数時刻をまとめて処理）
        _, encoder_hidden = self.encoder(batch_X, lengths_X)

        _batch_size = batch_X.size(1)

        # decoder の入力と隠れ層の初期状態を定義
        decoder_input = torch.tensor([BOS] * _batch_size, dtype=torch.long, device=device) # 最初の入力には BOS を使用する
        decoder_input = decoder_input.unsqueeze(0)  # (1, batch_size)
        decoder_hidden = encoder_hidden  # Encoder の最終隠れ状態を取得

        # decoder の出力のホルダーを定義
        decoder_outputs = torch.zeros(max_length, _batch_size, self.decoder.output_size, device=device) # max_length 分の固定長

        # 各時刻ごとに処理
        for t in range(max_length):
            decoder_output, decoder_hidden = self.decoder(decoder_input, decoder_hidden)
            decoder_outputs[t] = decoder_output
            # 次の時刻の decoder の入力を決定
            if use_teacher_forcing and batch_Y is not None:  # teacher force の場合、ターゲット系列を用いる
                decoder_input = batch_Y[t].unsqueeze(0)
            else:  # teacher force でない場合、自身の出力を用いる
                decoder_input = decoder_output.max(-1)[1]
```

```
        return decoder_outputs
mce = nn.CrossEntropyLoss(size_average=False, ignore_index=PAD) # P
AD を無視する
def masked_cross_entropy(logits, target):
    logits_flat = logits.view(-1, logits.size(-
1)) # (max_seq_len * batch_size, output_size)
    target_flat = target.view(-1) # (max_seq_len * batch_size, 1)
    return mce(logits_flat, target_flat)
# ハイパーパラメータの設定
num_epochs = 10
batch_size = 64
lr = 1e-3  # 学習率
teacher_forcing_rate = 0.2  # Teacher Forcing を行う確率
ckpt_path = 'model.pth'  # 学習済みのモデルを保存するパス

model_args = {
    'input_size': vocab_size_X,
    'output_size': vocab_size_Y,
    'hidden_size': 256,
}
# データローダを定義
train_dataloader = DataLoader(train_X, train_Y, batch_size=batch_si
ze, shuffle=True)
valid_dataloader = DataLoader(valid_X, valid_Y, batch_size=batch_si
ze, shuffle=False)

# モデルと Optimizer を定義
model = EncoderDecoder(**model_args).to(device)
optimizer = optim.Adam(model.parameters(), lr=lr)
def compute_loss(batch_X, batch_Y, lengths_X, model, optimizer=None
, is_train=True):
    # 損失を計算する関数
    model.train(is_train)  # train/eval モードの切替え

    # 一定確率で Teacher Forcing を行う
```

```
    use_teacher_forcing = is_train and (random.random() < teacher_forcing_rate)
    max_length = batch_Y.size(0)
    # 推論
    pred_Y = model(batch_X, lengths_X, max_length, batch_Y, use_teacher_forcing)

    # 損失関数を計算
    loss = masked_cross_entropy(pred_Y.contiguous(), batch_Y.contiguous())

    if is_train:  # 訓練時はパラメータを更新
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()

    batch_Y = batch_Y.transpose(0, 1).contiguous().data.cpu().tolist()
    pred = pred_Y.max(dim=-1)[1].data.cpu().numpy().T.tolist()

    return loss.item(), batch_Y, pred
def calc_bleu(refs, hyps):
    """
    BLEU スコアを計算する関数
    :param refs: list, 参照訳。単語のリストのリスト（例: [['I', 'have', 'a', 'pen'], ...]）
    :param hyps: list, モデルの生成した訳。単語のリストのリスト（例: ['I', 'have', 'a', 'pen']）
    :return: float, BLEU スコア(0~100)
    """
    refs = [[ref[:ref.index(EOS)]] for ref in refs] # EOS は評価しないで良いので切り捨てる，refs のほうは複数なので list が一個多くかかっている
    hyps = [hyp[:hyp.index(EOS)] if EOS in hyp else hyp for hyp in hyps]
    return 100 * bleu_score.corpus_bleu(refs, hyps)
# 訓練
```

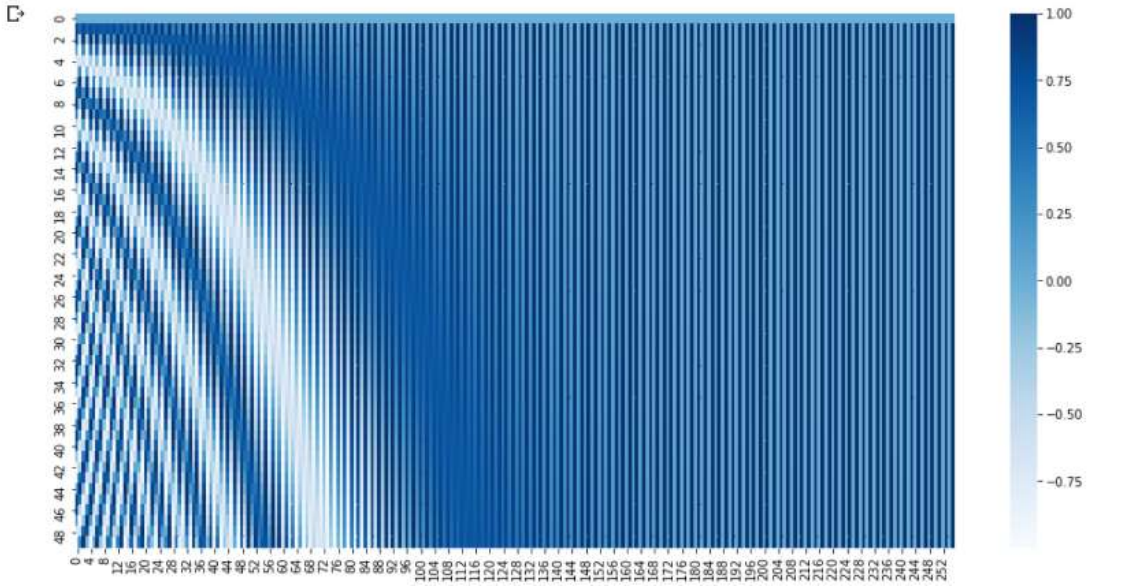
```
best_valid_bleu = 0.

for epoch in range(1, num_epochs+1):
    train_loss = 0.
    train_refs = []
    train_hyps = []
    valid_loss = 0.
    valid_refs = []
    valid_hyps = []
    # train
    for batch in train_dataloader:
        batch_X, batch_Y, lengths_X = batch
        loss, gold, pred = compute_loss(
            batch_X, batch_Y, lengths_X, model, optimizer,
            is_train=True
            )
        train_loss += loss
        train_refs += gold
        train_hyps += pred
    # valid
    for batch in valid_dataloader:
        batch_X, batch_Y, lengths_X = batch
        loss, gold, pred = compute_loss(
            batch_X, batch_Y, lengths_X, model,
            is_train=False
            )
        valid_loss += loss
        valid_refs += gold
        valid_hyps += pred
    # 損失をサンプル数で割って正規化
    train_loss = np.sum(train_loss) / len(train_dataloader.data)
    valid_loss = np.sum(valid_loss) / len(valid_dataloader.data)
    # BLEUを計算
    train_bleu = calc_bleu(train_refs, train_hyps)
    valid_bleu = calc_bleu(valid_refs, valid_hyps)
```

```
    # validation データで BLEU が改善した場合にはモデルを保存
    if valid_bleu > best_valid_bleu:
        ckpt = model.state_dict()
        torch.save(ckpt, ckpt_path)
        best_valid_bleu = valid_bleu

    print('Epoch {}: train_loss: {:5.2f}  train_bleu: {:2.2f}  valid_loss: {:5.2f}  valid_bleu: {:2.2f}'.format(
            epoch, train_loss, train_bleu, valid_loss, valid_bleu))

    print('-'*80)
# 学習済みモデルの読み込み
ckpt = torch.load(ckpt_path) # cpu で処理する場合は map_location で指定する必要があります。
model.load_state_dict(ckpt)
model.eval()
def ids_to_sentence(vocab, ids):
    # ID のリストを単語のリストに変換する
    return [vocab.id2word[_id] for _id in ids]

def trim_eos(ids):
    # ID のリストから EOS 以降の単語を除外する
    if EOS in ids:
        return ids[:ids.index(EOS)]
    else:
        return ids
# テストデータの読み込み
test_X = load_data('./data/dev.en')
test_Y = load_data('./data/dev.ja')
test_dataloader = DataLoader(test_X, test_Y, batch_size=1, shuffle=False)
# 生成
batch_X, batch_Y, lengths_X = next(test_dataloader)
sentence_X = ' '.join(ids_to_sentence(vocab_X, batch_X.data.cpu().numpy()[:-1, 0]))
```

```
sentence_Y = ' '.join(ids_to_sentence(vocab_Y, batch_Y.data.cpu().numpy()[:-1, 0]))
print('src: {}'.format(sentence_X))
print('tgt: {}'.format(sentence_Y))

output = model(batch_X, lengths_X, max_length=20)
output = output.max(dim=-1)[1].view(-1).data.cpu().tolist()
output_sentence = ' '.join(ids_to_sentence(vocab_Y, trim_eos(output)))
output_sentence_without_trim = ' '.join(ids_to_sentence(vocab_Y, output))
print('out: {}'.format(output_sentence))
print('without trim: {}'.format(output_sentence_without_trim))
```

実施結果 Transformer

```
pe = position_encoding_init(50, 256).numpy()
plt.figure(figsize=(16,8))
sns.heatmap(pe, cmap='Blues')
plt.show()
```



```
[82] src, tgt = next(test_dataloader)

     src_ids = src[0][0].cpu().numpy()
     tgt_ids = tgt[0][0].cpu().numpy()

     print('src: {}'.format(' '.join(ids_to_sentence(vocab_X, src_ids[1:-1]))))
     print('tgt: {}'.format(' '.join(ids_to_sentence(vocab_Y, tgt_ids[1:-1]))))

     preds, enc_slf_attns, dec_slf_attns, dec_enc_attns = test(model, src)
     pred_ids = preds[0].data.cpu().numpy().tolist()
     print('out: {}'.format(' '.join(ids_to_sentence(vocab_Y, trim_eos(pred_ids)))))

     src: he lived a hard life .
     tgt: 彼 は つら い 人生 を 送 っ た 。
     out: 彼 は <UNK> に 住 ん で い た 。
```

参照コード：

```
! wget https://www.dropbox.com/s/9narw5x4uizmehh/utils.py
! mkdir images data

# data 取得
! wget https://www.dropbox.com/s/o4kyc52a8we25wy/dev.en -P data/
! wget https://www.dropbox.com/s/kdgskm5hzg6znuc/dev.ja -P data/
! wget https://www.dropbox.com/s/gyyx4gohv9v65uh/test.en -P data/
! wget https://www.dropbox.com/s/hotxwbgoe2n013k/test.ja -P data/
! wget https://www.dropbox.com/s/5lsftkmb20ay9e1/train.en -P data/
! wget https://www.dropbox.com/s/ak53qirssci6f1j/train.ja -P data/
import time
import numpy as np
from sklearn.utils import shuffle
from sklearn.model_selection import train_test_split
import matplotlib
import matplotlib.pyplot as plt
import seaborn as sns
%matplotlib inline

from nltk import bleu_score

import torch
import torch.nn as nn
```

```
import torch.optim as optim
import torch.nn.functional as F

from utils import Vocab

device = torch.device("cuda" if torch.cuda.is_available() else "cpu
")

torch.manual_seed(1)
random_state = 42
PAD = 0
UNK = 1
BOS = 2
EOS = 3

PAD_TOKEN = '<PAD>'
UNK_TOKEN = '<UNK>'
BOS_TOKEN = '<S>'
EOS_TOKEN = '</S>'
def load_data(file_path):
    """
    テキストファイルからデータを読み込む
    :param file_path: str, テキストファイルのパス
    :return data: list, 文章(単語のリスト)のリスト
    """
    data = []
    for line in open(file_path, encoding='utf-8'):
        words = line.strip().split()  # スペースで単語を分割
        data.append(words)
    return data
train_X = load_data('./data/train.en')
train_Y = load_data('./data/train.ja')
# 訓練データと検証データに分割
train_X, valid_X, train_Y, valid_Y = train_test_split(train_X, trai
n_Y, test_size=0.2, random_state=random_state)
# データセットの中身を確認
```

```
print('train_X:', train_X[:5])
print('train_Y:', train_Y[:5])
MIN_COUNT = 2  # 語彙に含める単語の最低出現回数

word2id = {
    PAD_TOKEN: PAD,
    BOS_TOKEN: BOS,
    EOS_TOKEN: EOS,
    UNK_TOKEN: UNK,
    }

vocab_X = Vocab(word2id=word2id)
vocab_Y = Vocab(word2id=word2id)
vocab_X.build_vocab(train_X, min_count=MIN_COUNT)
vocab_Y.build_vocab(train_Y, min_count=MIN_COUNT)

vocab_size_X = len(vocab_X.id2word)
vocab_size_Y = len(vocab_Y.id2word)
def sentence_to_ids(vocab, sentence):
    """
    単語のリストをインデックスのリストに変換する
    :param vocab: Vocab のインスタンス
    :param sentence: list of str
    :return indices: list of int
    """
    ids = [vocab.word2id.get(word, UNK) for word in sentence]
    ids = [BOS] + ids + [EOS]  # EOS を末尾に加える
    return ids
train_X = [sentence_to_ids(vocab_X, sentence) for sentence in train
_X]
train_Y = [sentence_to_ids(vocab_Y, sentence) for sentence in train
_Y]
valid_X = [sentence_to_ids(vocab_X, sentence) for sentence in valid
_X]
valid_Y = [sentence_to_ids(vocab_Y, sentence) for sentence in valid
_Y]
```

```
class DataLoader(object):
    def __init__(self, src_insts, tgt_insts, batch_size, shuffle=Tr
ue):
        """
        :param src_insts: list, 入力言語の文章(単語 ID のリスト)のリスト
        :param tgt_insts: list, 出力言語の文章(単語 ID のリスト)のリスト
        :param batch_size: int, バッチサイズ
        :param shuffle: bool, サンプルの順番をシャッフルするか否か
        """
        self.data = list(zip(src_insts, tgt_insts))

        self.batch_size = batch_size
        self.shuffle = shuffle
        self.start_index = 0

        self.reset()

    def reset(self):
        if self.shuffle:
            self.data = shuffle(self.data, random_state=random_stat
e)
        self.start_index = 0

    def __iter__(self):
        return self

    def __next__(self):

        def preprocess_seqs(seqs):
            # パディング
            max_length = max([len(s) for s in seqs])
            data = [s + [PAD] * (max_length - len(s)) for s in seqs
]
            # 単語の位置を表現するベクトルを作成
            positions = [[pos+1 if w != PAD else 0 for pos, w in en
umerate(seq)] for seq in data]
```

```
            # テンソルに変換
            data_tensor = torch.tensor(data, dtype=torch.long, device=device)
            position_tensor = torch.tensor(positions, dtype=torch.long, device=device)
            return data_tensor, position_tensor

        # ポインタが最後まで到達したら初期化する
        if self.start_index >= len(self.data):
            self.reset()
            raise StopIteration()

        # バッチを取得して前処理
        src_seqs, tgt_seqs = zip(*self.data[self.start_index:self.start_index+self.batch_size])
        src_data, src_pos = preprocess_seqs(src_seqs)
        tgt_data, tgt_pos = preprocess_seqs(tgt_seqs)

        # ポインタを更新する
        self.start_index += self.batch_size

        return (src_data, src_pos), (tgt_data, tgt_pos)
def position_encoding_init(n_position, d_pos_vec):
    """
    Positional Encoding のための行列の初期化を行う
    :param n_position: int, 系列長
    :param d_pos_vec: int, 隠れ層の次元数
    :return torch.tensor, size=(n_position, d_pos_vec)
    """
    # PAD がある単語の位置は pos=0 にしておき、position_enc も 0 にする
    position_enc = np.array([
        [pos / np.power(10000, 2 * (j // 2) / d_pos_vec) for j in range(d_pos_vec)]
        if pos != 0 else np.zeros(d_pos_vec) for pos in range(n_position)])
```

```
    position_enc[1:, 0::2] = np.sin(position_enc[1:, 0::2])  # dim
2i
    position_enc[1:, 1::2] = np.cos(position_enc[1:, 1::2])  # dim
2i+1
    return torch.tensor(position_enc, dtype=torch.float)
pe = position_encoding_init(50, 256).numpy()
plt.figure(figsize=(16,8))
sns.heatmap(pe, cmap='Blues')
plt.show()
def compute_loss(batch_X, batch_Y, model, criterion, optimizer=None
, is_train=True):
    # バッチの損失を計算
    model.train(is_train)

    pred_Y = model(batch_X, batch_Y)
    gold = batch_Y[0][:, 1:].contiguous()
#     gold = batch_Y[0].contiguous()
    loss = criterion(pred_Y.view(-1, pred_Y.size(2)), gold.view(-
1))

    if is_train:  # 訓練時はパラメータを更新
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()

    gold = gold.data.cpu().numpy().tolist()
    pred = pred_Y.max(dim=-1)[1].data.cpu().numpy().tolist()

    return loss.item(), gold, pred
MAX_LENGTH = 20
batch_size = 64
num_epochs = 15
lr = 0.001
ckpt_path = 'transformer.pth'
max_length = MAX_LENGTH + 2
model_args = {
```

```
    'n_src_vocab': vocab_size_X,
    'n_tgt_vocab': vocab_size_Y,
    'max_length': max_length,
    'proj_share_weight': True,
    'd_k': 32,
    'd_v': 32,
    'd_model': 128,
    'd_word_vec': 128,
    'd_inner_hid': 256,
    'n_layers': 3,
    'n_head': 6,
    'dropout': 0.1,
}
# DataLoader やモデルを定義
train_dataloader = DataLoader(
    train_X, train_Y, batch_size
    )
valid_dataloader = DataLoader(
    valid_X, valid_Y, batch_size,
    shuffle=False
    )

model = Transformer(**model_args).to(device)

optimizer = optim.Adam(model.get_trainable_parameters(), lr=lr)

criterion = nn.CrossEntropyLoss(ignore_index=PAD, size_average=Fals
e).to(device)
def calc_bleu(refs, hyps):
    """
    BLEU スコアを計算する関数
    :param refs: list, 参照訳。単語のリストのリス
ト (例: [['I', 'have', 'a', 'pen'], ...])
    :param hyps: list, モデルの生成した訳。単語のリストのリス
ト (例: [['I', 'have', 'a', 'pen'], ...])
    :return: float, BLEU スコア(0~100)
```

```
    """
    refs = [[ref[:ref.index(EOS)]] for ref in refs]
    hyps = [hyp[:hyp.index(EOS)] if EOS in hyp else hyp for hyp in
hyps]
    return 100 * bleu_score.corpus_bleu(refs, hyps)
# 訓練
best_valid_bleu = 0.

for epoch in range(1, num_epochs+1):
    start = time.time()
    train_loss = 0.
    train_refs = []
    train_hyps = []
    valid_loss = 0.
    valid_refs = []
    valid_hyps = []
    # train
    for batch in train_dataloader:
        batch_X, batch_Y = batch
        loss, gold, pred = compute_loss(
            batch_X, batch_Y, model, criterion, optimizer, is_train
=True
            )
        train_loss += loss
        train_refs += gold
        train_hyps += pred
    # valid
    for batch in valid_dataloader:
        batch_X, batch_Y = batch
        loss, gold, pred = compute_loss(
            batch_X, batch_Y, model, criterion, is_train=False
            )
        valid_loss += loss
        valid_refs += gold
        valid_hyps += pred
    # 損失をサンプル数で割って正規化
```

```
    train_loss /= len(train_dataloader.data)
    valid_loss /= len(valid_dataloader.data)
    # BLEU を計算
    train_bleu = calc_bleu(train_refs, train_hyps)
    valid_bleu = calc_bleu(valid_refs, valid_hyps)

    # validation データで BLEU が改善した場合にはモデルを保存
    if valid_bleu > best_valid_bleu:
        ckpt = model.state_dict()
        torch.save(ckpt, ckpt_path)
        best_valid_bleu = valid_bleu

    elapsed_time = (time.time()-start) / 60
    print('Epoch {} [{:.1f}min]: train_loss: {:5.2f}  train_bleu: {
:2.2f}  valid_loss: {:5.2f}  valid_bleu: {:2.2f}'.format(
            epoch, elapsed_time, train_loss, train_bleu, valid_loss
, valid_bleu))
    print('-'*80)
def test(model, src, max_length=20):
    # 学習済みモデルで系列を生成する
    model.eval()

    src_seq, src_pos = src
    batch_size = src_seq.size(0)
    enc_output, enc_slf_attns = model.encoder(src_seq, src_pos)

    tgt_seq = torch.full([batch_size, 1], BOS, dtype=torch.long, de
vice=device)
    tgt_pos = torch.arange(1, dtype=torch.long, device=device)
    tgt_pos = tgt_pos.unsqueeze(0).repeat(batch_size, 1)

    # 時刻ごとに処理
    for t in range(1, max_length+1):
        dec_output, dec_slf_attns, dec_enc_attns = model.decoder(
            tgt_seq, tgt_pos, src_seq, enc_output)
        dec_output = model.tgt_word_proj(dec_output)
```

```
            out = dec_output[:, -1, :].max(dim=-1)[1].unsqueeze(1)
            # 自身の出力を次の時刻の入力にする
            tgt_seq = torch.cat([tgt_seq, out], dim=-1)
            tgt_pos = torch.arange(t+1, dtype=torch.long, device=device
)
            tgt_pos = tgt_pos.unsqueeze(0).repeat(batch_size, 1)

    return tgt_seq[:, 1:], enc_slf_attns, dec_slf_attns, dec_enc_at
tns
def ids_to_sentence(vocab, ids):
    # IDのリストを単語のリストに変換する
    return [vocab.id2word[_id] for _id in ids]

def trim_eos(ids):
    # IDのリストからEOS以降の単語を除外する
    if EOS in ids:
        return ids[:ids.index(EOS)]
    else:
        return ids
# 学習済みモデルの読み込み
model = Transformer(**model_args).to(device)
ckpt = torch.load(ckpt_path)
model.load_state_dict(ckpt)
# テストデータの読み込み
test_X = load_data('./data/dev.en')
test_Y = load_data('./data/dev.ja')
test_X = [sentence_to_ids(vocab_X, sentence) for sentence in test_X
]
test_Y = [sentence_to_ids(vocab_Y, sentence) for sentence in test_Y
]
test_dataloader = DataLoader(
    test_X, test_Y, 1,
    shuffle=False
    )
src, tgt = next(test_dataloader)
```

```
src_ids = src[0][0].cpu().numpy()
tgt_ids = tgt[0][0].cpu().numpy()

print('src: {}'.format(' '.join(ids_to_sentence(vocab_X, src_ids[1:
-1]))))
print('tgt: {}'.format(' '.join(ids_to_sentence(vocab_Y, tgt_ids[1:
-1]))))

preds, enc_slf_attns, dec_slf_attns, dec_enc_attns = test(model, sr
c)
pred_ids = preds[0].data.cpu().numpy().tolist()
print('out: {}'.format(' '.join(ids_to_sentence(vocab_Y, trim_eos(p
red_ids)))))
# BLEUの評価
test_dataloader = DataLoader(
    test_X, test_Y, 128,
    shuffle=False
    )
refs_list = []
hyp_list = []

for batch in test_dataloader:
    batch_X, batch_Y = batch
    preds, *_ = test(model, batch_X)
    preds = preds.data.cpu().numpy().tolist()
    refs = batch_Y[0].data.cpu().numpy()[:, 1:].tolist()
    refs_list += refs
    hyp_list += preds
bleu = calc_bleu(refs_list, hyp_list)
print(bleu)
```

Section 6 ：物体検知・セグメンテーション

・物体認識タスク

分類：(画像（カラー・モノクロは問わない）に対し単一または複数の）クラスラベル

物体検知：Bounding Box [bbox/BB]

セマンティックセグメンテーション（意味領域分割)：(各ピクセルに対し単一の） クラスラベル

インスタンスセグメンテーション（個体領域分割)：(各ピクセルに対し単一の） クラスラベル

・代表的なデータセット(DATASET)

目的に応じた「BOX/画像」の選択を！

小：アイコン的な映り。日常感とはかけ離れやすい。

大：部分的な重なり等も見られる。日常生活のコンテクストに近い。

目的に合ったデータセットを選ぶ事が大切（クラス数あるいは BOX／画像を 1 つの基準として）

クラス数が大きいことは嬉しいのか？⇒目的に応じた選択

VOC12（Visual Object Classes）⇒20 クラス

ILSVRC12: ImageNet のサブセット⇒200 クラス

MS COCO18 (Microsoft Common Object in Context）⇒80 クラス

OICOD18 (Open Images Challenges Object Detection）⇒500 クラス

・評価指標

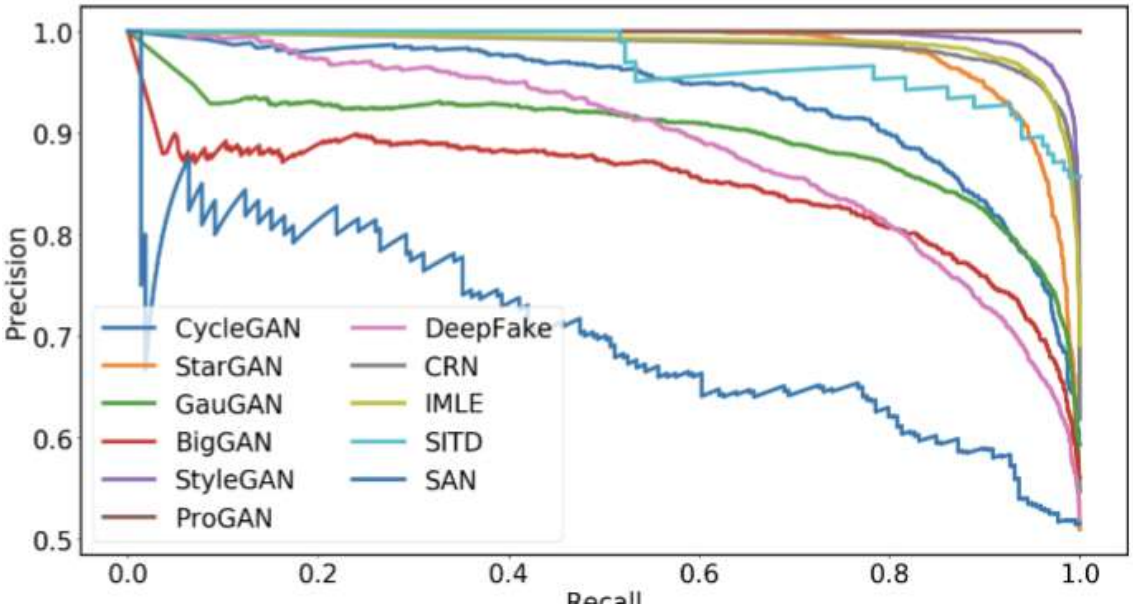
分類問題における評価指標の復習

Confusion Matrix

Precision = TP/(TP + FP)

Recall = TP/(TP + FN)

pdfconfidence の閾値を変化させる事で Precision-Recall curve が描ける。



クラス分類の場合、閾値を変えても混同行列に含まれる件数(サンプル数)が変わらない。

物体検出の場合はクラス分類と異なり、件数(サンプル数)が変わりうる。

IoU（Intersection over Union）

Motivation 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！

IoU = Area of Overlap / Area of Union

IoU は値の直観的解釈が難しい。

（入力 1 枚で見る）Precision/Recall

conf.の閾値：0.5　IoU の閾値：0.5

Precision/Recall の計算例（クラス単位）

この Precision や Recall は Conf.の閾値を変えれば変わるはずでした。

そこから出て来るのが物体検知で用いられる指標：Average Precision (AP)。

AP: Average Precision

Average Precision の定義

$$\mathrm{AP} = \int_0^1 P(R)\mathrm{d}R$$ (和訳：PR曲線の下側面積)

PR 曲線を描くために Conf.を 0.05 から変化させていくと、conf.の閾値の変化に伴って Recall, Precision が変わり、それによって PR 曲線が描ける。

物体検出の精度評価の指標：AP = PR 曲線の下側面積

AP が大きい方が良い精度指標になっている。

厳密には各 Recall のレベルに対して最大の Precision にスムージング。

積分は Interpolated AP として有限点で計算される。

mAP: mean Average Precision

AP の平均（AP はクラスごとに計算される。）

mAP COCO: MS COCO で導入された指標

IoU 閾値は 0.5 で固定→0.5 から 0.95 まで 0.05 刻みで AP＆mAP を計算し算術平均を計算⇒mAPCOCO=(mAP0.5+ mAP0.55+…+ mAP0.9510)/10

FPS：Flames per Second

物体検知応用上の要請から，検出精度に加え検出速度も問題となる。

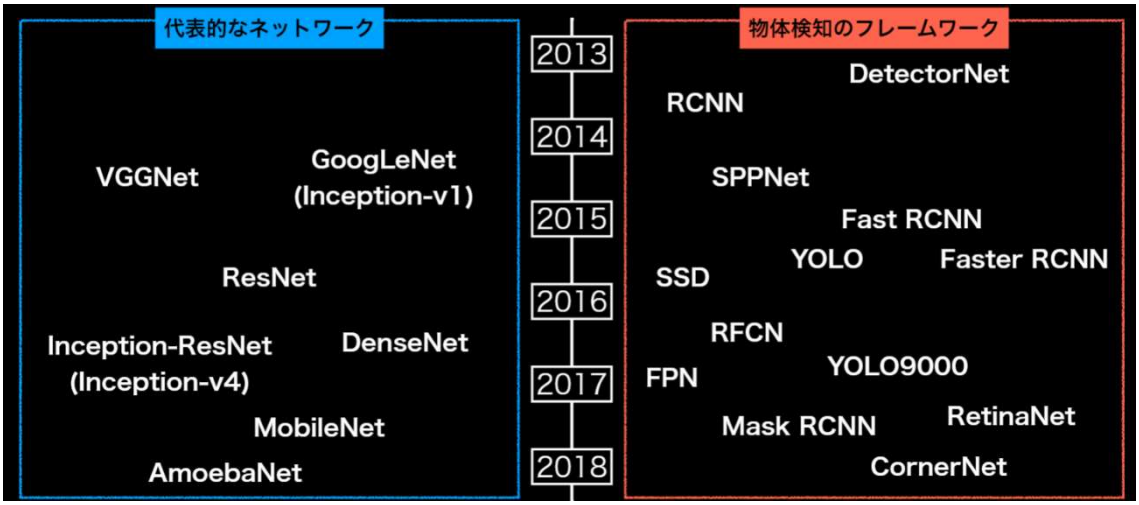
利用したデータセットが大事です。

・物体検知の大枠

マイルストーン：深層学習以降の物体検知

2012 AlexNet の登場を皮切りに，時代は SIFT から DCNN へ

ネットワークイメージ図



物体検知のフレームワーク

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行う

相対的に精度が低い傾向

相対的に計算量が小さく推論も早い傾向

2 段階検出器（Two-stage detector）

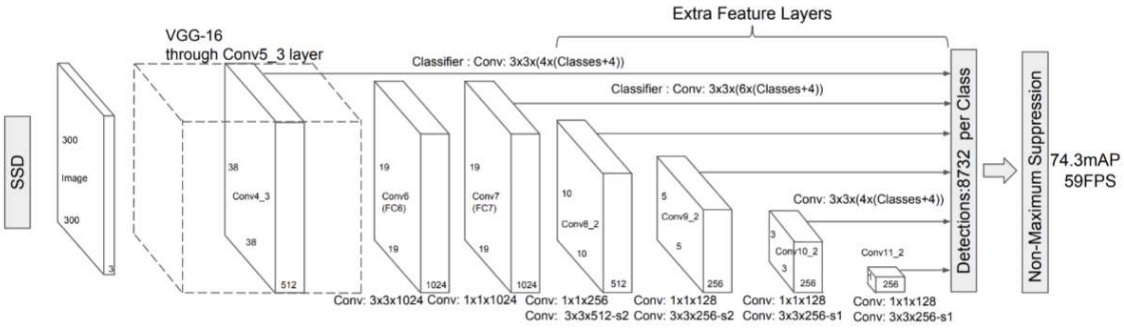
候補領域の検出とクラス推定を別々に行う

相対的に精度が高い傾向

相対的に計算量が大きく推論も遅い傾向

・SSD: Single Shot Detector

SSD のネットワークアーキテクチャ



YOLO と並んで有名なモデル。

１段階検出器モデルの１つです。

VGG16 をベースとしたアーキテクチャ。

（VGG16 は「ImageNet」と呼ばれる大規模画像データセットで学習された CNN モデルです。16 層から成り立っており、入力画像を 1000 のクラスに分類することができます。）

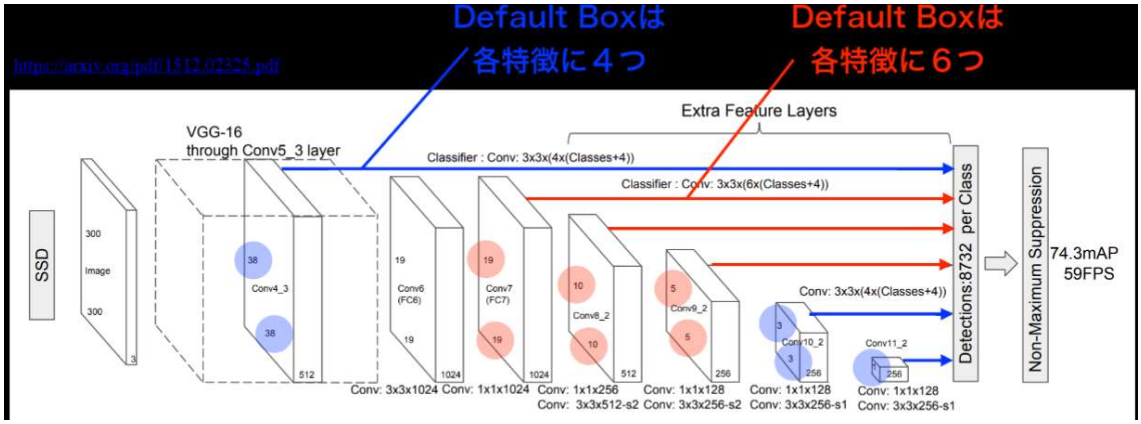
特徴マップからの出力

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は「#Class+4」。

マップ中の各特徴量に k 個の Default Box を用意するとき，出力サイズは「k（#Class + 4）」。4⇒オフセット項 △x、△y、△w、△h

更に，特徴マップのサイズが m×n であるとすれば，出力サイズは「k（#Class + 4）mn」。

SSD のデフォルトボックス数



多数の Default Box を用意したことで生ずる問題への対処

Non-Maximum Suppression

Default Box を複数用意したことにより１つの物体しか映っていなくとも、そこに複数の Bounding Box が存在する事になってしまいます。特徴マップごとに Default Box 数を用意する。

対策：IoU を計算し、IoU が 1 定以上かぶっているのであれば、最も conf.が高いもののみを残す。

Hard Negative Mining

物体として映っているものよりも、背景と判断されるネガティヴクラスに属するような Bounding Box は多く存在するはず。背景（ネガティヴ）と非背景（ポジティヴ）のバランスが非常に不均衡になる事が用意に想像される。

対策：最大でも 1:3 となるように、ネガティヴの方が大きくても 3 倍でというような、制約をかける（ネガティヴの Bounding Box を削っている。）

損失関数

損失関数が confidence に対する損失と検出位置に対する損失を確認。

・Semantic Segmentation の概略

Up-sampling

Convolution や Pooling を繰り返す事により、入力画像の解像度が落ちていくという問題がある。セマンティックセグメンテーションでは各ピクセルに対してクラス分類を行う事となるため、落ちた解像度をどのようにして元に戻すのか、が問題となる。この解像度をもとに戻す事を Up-sampling と言う。

Deconvolution/Transposed convolution

通常の Conv.層と同様 kernel size, padding, stride を指定

処理手順

特徴マップの pixel 間隔を stride だけ空ける

特徴マップのまわりに(kernel size - 1) - padding だけ余白を作る

畳み込み演算を行う

※逆畳み込みと呼ばれることも多いが畳み込みの逆演算ではないことに注意　→当然, pooling で失われた情報が復元されるわけではない。

輪郭情報の補完

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補完してから Up-sampling

Unpooling

Unpooling 層は Convolutional AutoEncoder の Encoder 側で MaxPooling 層を用いている際に Decoder 側に対応するように使われる層です。

Dilated Convolution

Convolution の段階で受容野を広げる工夫。

実施結果⇒該当なし

確認テスト⇒該当なし