深層学習 day3 レポート

1　再帰型ニューラルネットワークの概念

1.1　要点のまとめ

・再帰型ニューラルネットワーク(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である

・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。具体的には、音声データ・テキストデータ等

・時系列モデルを扱うには、初期の状態と過去の時間 t-1 の状態を保持し、そこから次の時間での t を再帰的に求める再帰構造が必要になる。

・BPTT とは、RNN においてのパラメータ調整方法の一種で誤差逆伝播の一種

1.2　演習結果

▾ simple RNN

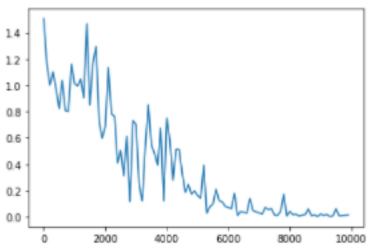
バイナリ加算

```
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 = 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


# 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_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:9300
Loss:0.0017941907521142946
Pred:[1 0 0 1 1 1 1 0]
True:[1 0 0 1 1 1 1 0]
77 + 81 = 158
------------
iters:9400
Loss:0.005859867517844166
Pred:[1 0 0 0 0 1 1 0]
True:[1 0 0 0 0 1 1 0]
123 + 11 = 134
------------
iters:9500
Loss:0.06080931869875537
Pred:[1 0 1 1 1 0 1 1]
True:[1 0 1 1 1 0 1 1]
127 + 60 = 187
------------
iters:9600
Loss:0.007068457801294877
Pred:[1 0 1 1 0 0 1 0]
True:[1 0 1 1 0 0 1 0]
104 + 74 = 178
------------
iters:9700
Loss:0.008937013135616581
Pred:[1 0 1 0 1 1 0 0]
True:[1 0 1 0 1 1 0 0]
84 + 88 = 172
------------
iters:9800
Loss:0.012170269605238948
Pred:[0 1 0 1 0 1 0 1]
True:[0 1 0 1 0 1 0 1]
78 + 7 = 85
------------
iters:9900
Loss:0.013864374582548198
Pred:[0 0 0 1 1 0 0 1]
True:[0 0 0 1 1 0 0 1]
15 + 10 = 25
------------
```



1.3　考察

・RNN のネットワークには大きくわけて下記の 3 つの重みがある。

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

残りは中間層から中間層の重み

2 LSTM

2.1 要点のまとめ

・時系列を遡れば遡るほど、勾配が消失していくことが RNN の課題。構造自体を変えて解決したものが LSTM。

・CEC は、入力データについて、時間依存度に関係なく重みが一律である。勾配消失および勾配爆発の解決方法であるが、ニューラルネットワークの学習特性が無いことは課題。

・CEC は、過去の情報が全て保管されているのに対し、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 LSTM

2.2 演習結果

```python
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_size - 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

    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_filename)

        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.unknown_word_symbol]))
        return self.dictionary.get(word, self.dictionary[self.unknown_word_symbol])
```

[4]

```
        # 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.ckpt"
            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_layer_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_layer_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, initial_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_logits(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.learning_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(actual_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).strftime("%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/" + dirname, 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: batch_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 out of", self.epochs, "-- 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_data: trX,
                                                           actual_labels: trY,
                                                           initial_state: np.zeros(
                                                               [trX.shape[0],
                                                                self.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 v[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")
```

```
cofactor : 1.4113519e-14
equatorial : 1.41412455e-14
tensors : 1.4906079e-14
epr : 1.419034e-14
vo : 1.32667205e-14
soluble : 1.5729129e-14
atp : 1.3559849e-14
adp : 1.5086707e-14
purified : 1.3827846e-14
nucleotidase : 1.4714473e-14
sod : 1.39041475e-14
adpnp : 1.306515e-14
spectra : 1.4082702e-14
mhz : 1.5197553e-14
carboxyl : 1.5074166e-14
oxygens : 1.4567642e-14
hydroxyl : 1.40896875e-14
hydrolysis : 1.4591056e-14
phosphates : 1.5364852e-14
conformations : 1.4613365e-14
conformation : 1.3528383e-14
annan : 1.4627056e-14
unesco : 1.3770205e-14
sci : 1.37950225e-14
ricyt : 1.450122e-14
averaged : 5.898562e-14
ecological : 1.4461912e-14
statins : 1.4468122e-14
coronary : 1.242908e-14
myocardial : 1.3684015e-14
infarction : 1.4459073e-14
revascularization : 1.410722e-14
lipid-lowering : 1.5685598e-14
incremental : 1.3963625e-14
ischemia : 1.4152389e-14
miracl : 1.510721e-14
atorvastatin : 1.3983241e-14
mgdl : 1.5000311e-14
rehospitalization : 1.456842e-14
worsening : 1.3476926e-14
endpoints : 1.4479218e-14
lipid : 1.5310498e-14
lipoprotein : 1.3668963e-14
ldl : 1.4424475e-14
statin : 1.3786473e-14
a--z : 1.3740822e-14
simvastatin : 1.4133777e-14
bmi : 3.1721086e-07
covariates : 2.834505e-06
yhl : 2.2330451e-07
yol : 9.9930106e-11
obesity : 1.3501609e-09
evgfp : 6.1830234e-09
unintended : 4.67851e-09
sizes : 2.5699424e-07
obese : 1.9368164e-07
<???> : 2.919565e-05
Prediction: some of them looks like et
```

2.3　考察

・忘却は人間的な考え方だと思っていたため、AI の世界でも適用できることは非常に面白いと思った。

3 GRU

3.1 要点のまとめ

・従来の LSTM では、パラメータが多数存在していたため、計算負荷が大きかった。
・GRU では、そのパラメータを大幅に削減し、精度は同等またはそれ以上が望める様になった構造。
・計算負荷が低いため、計算資源を有効に利用でき、処理時間の短縮にもつながる。

3.2 考察

・パラメータを大幅に削減しても、精度を同等に保てるということは、パラメータのうち、かなりの部分は成果に影響を及ぼしていないということだと認識した。

4　双方向 RNN

4.1　要点のまとめ

・双方向 RNN とは過去の情報だけでなく、未来の情報を加味することで、精度を向上させるためのモデル

・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。

・文書の遂行や機械翻訳等で使用される。

・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

4.2　演習結果

predict sin

[try]

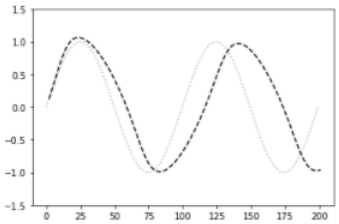
- iters_numを100にしよう
- maxlenを5, iters_numを500, 3000(※時間がかかる)にしよう

```
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 = 500

# 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]:

            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)
    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: 4.2320445876570847e-07    d: [0.99583607]    y: [0.99675608]
loss: 3.787447800805401e-07    d: [0.44358222]    y: [0.44445256]
loss: 1.8816335271670955e-06    d: [-0.94789551]    y: [-0.94983543]
loss: 1.2190012629927552e-06    d: [-0.97512765]    y: [-0.97668906]
loss: 1.996710038100934e-06    d: [-0.694759]    y: [-0.69675736]
loss: 1.8849134192597632e-06    d: [-0.2430756]    y: [-0.24501721]
loss: 1.874812344256188e-06    d: [0.22471249]    y: [0.22664888]
loss: 4.2620357543521325e-07    d: [-0.10056216]    y: [-0.0996389]
loss: 4.325800120879655e-09    d: [0.68564773]    y: [0.68555477]
loss: 1.8306176723616166e-06    d: [0.29761864]    y: [0.29953208]
loss: 2.332044298261599e-09    d: [-0.99583607]    y: [-0.99576778]
loss: 6.585761031381603e-07    d: [-0.99085292]    y: [-0.99200059]
loss: 7.651581590766538e-08    d: [0.35120641]    y: [0.3515976]
loss: 7.290859282130002e-07    d: [-0.88033969]    y: [-0.87913214]
loss: 1.668581403709776e-09    d: [-0.68105132]    y: [-0.68099355]
loss: 2.691838713149576e-06    d: [0.85534252]    y: [0.85766279]
loss: 7.327100876997763e-07    d: [-0.98907524]    y: [-0.99028578]
loss: 5.490892948262962e-08    d: [-0.71705202]    y: [-0.71672063]
loss: 6.837326417290091e-07    d: [-0.86179776]    y: [-0.86062837]
loss: 1.6566551435476584e-06    d: [0.13806466]    y: [0.13988491]
loss: 5.681550927765138e-07    d: [-0.48263615]    y: [-0.48370213]
loss: 2.258902803175786e-07    d: [-0.15052435]    y: [-0.1498522]
loss: 1.745212320460355e-06    d: [0.16296018]    y: [0.16482845]
loss: 2.091550344328814e-06    d: [0.70821885]    y: [0.71026411]
loss: 1.7631822152189272e-06    d: [0.95374324]    y: [0.9556211]
loss: 2.4114224765198005e-07    d: [0.97512765]    y: [0.97443319]
loss: 2.067490890144045e-06    d: [0.93739898]    y: [0.93943244]
loss: 8.485394806796522e-07    d: [0.98611478]    y: [0.9874175]
loss: 1.729863259082727e-07    d: [0.98039856]    y: [0.97981136]
loss: 3.9139136578777065e-07    d: [-0.55262221]    y: [-0.55350696]
loss: 5.144245939052398e-07    d: [0.08175375]    y: [0.08073943]
loss: 1.5434925122992598e-06    d: [-0.37467145]    y: [-0.37642843]
loss: 8.646011169609011e-08    d: [0.98714074]    y: [0.9867249]
loss: 1.7821870821636891e-06    d: [0.17537017]    y: [0.17725812]
loss: 1.7227716039720648e-06    d: [-0.95561698]    y: [-0.9574732]
loss: 1.7032584357934278e-06    d: [0.15052435]    y: [0.15237003]
loss: 2.2961399277295135e-06    d: [-0.92114593]    y: [-0.92328889]
loss: 1.4861572178320616e-07    d: [-0.38050117]    y: [-0.38104635]
loss: 8.149646046185881e-07    d: [-0.48814053]    y: [-0.48941721]
loss: 9.154701847836336e-07    d: [0.54208448]    y: [0.5434376]
loss: 8.191872200209376e-09    d: [-0.69021707]    y: [-0.69008907]
loss: 1.1946819048405905e-07    d: [0.60896952]    y: [0.60945833]
```



5 Seq2Seq

5.1 要点のまとめ

・Seq2seq とは、Encoder-Decoder モデルの一種。機械対話や、機械翻訳などに使用されている。

・Encoder RNN は、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

・Decoder RNN はシステムがアウトプットデータを、単語等のトークンごとに生成する構造。

・Seq2seq の課題である一問一答しかできないに対応したものが HRED である。

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

・オートエンコーダとは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。

・通常のオートエンコーダの場合、何かしら潜在変数 z にデータを押し込めているものの、その構造がどのような状態かわからない。VAE は、データを潜在変数 z の確率分布という構造に押し込めることを可能にする。

6 Word2vec

6.1 要点のまとめ

・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ため、固定長形式で単語を表す必要がある。

・Word2vec では学習データからボキャブラリを作成

・辞書の単語数だけ one-hot ベクトルができあがる。

・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。

7 Attention Mechanism

7.1 要点のまとめ

・seq2seq の問題は長い文章への対応が難しい。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。

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

・Attention Mechanism は「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

深層学習 day4 レポート

1　強化学習

1.1　要点のまとめ

・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

・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

・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

・探索と利用はトレードオフの関係にある。

・関数近似法と、Q 学習を組み合わせる手法の登場

・Q 学習とは行動価値関数を、行動する毎に更新することにより学習を進める方法

・関数近似法とは価値関数や方策関数を関数近似する手法のこと

・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、行動価値関数を利用する。

・方策関数とは方策ベースの強化学習手法において、ある状態でどのような行動を採るのかの確率を与える関数のこと。

・方策反復法とは、方策をモデル化して最適化する手法。方策の良さを示す関数は定義しなければならない。定義方法としては、平均報酬、割引報酬和がある。

1.2　考察

・他の深層学習（教師あり、教師なし学習）とは異なり優れた方策(報酬が大きい方策)の探求が目標である。

・強化学習は計算資源を多く使うため停滞していた時期があった。近年の進展はコンピュータ性能が向上したことによるものが大きい。

2 AlphaGo

2.1 要点のまとめ

・AlphaGo の学習は以下のステップで行われる。

① 教師あり学習による RollOutPolicy と PolicyNet の学習

② 強化学習による PolicyNet の学習

③ 強化学習による ValueNet の学習

・PolicyNet の教師あり学習については、KGS Go Server（ネット囲碁対局サイト）の棋譜データから 3000 万局面分の教師を用意し、教師と同じ着手を予測できるよう学習を行った。

・現状の PolicyNet と PolicyPool からランダムに選択された PolicyNet と対局シミュレーションを行い、その結果を用いて方策勾配法で学習を行った。

・ValueNet の学習については、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した。

・囲碁ソフトではモンテカルロ木探索が最も有効とされている。

・Alpha Go のモンテカルロ木探索は選択、評価、バックアップ、成長という 4 つのステップで構成

・AlphaGoZero は AlphaGo(Lee)と異なり、教師あり学習を一切行わず、強化学習のみで作成

・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

2.2 考察

・近年では囲碁や将棋でプロ棋士を上回るほどに進展している。

・Alpha Go は教師あり学習を利用していたが、強化学習のみで行う AlphaGoZero には、AI の新たな可能性があると認識した。

・ソフトによる評価点は、囲碁や将棋の解説でも利用されており世間の認知度も高まっている。

3 軽量化・高速化技術

3.1 要点のまとめ

・深層学習は多くのデータを使用したり、パラメータ調整のために多くの時間を使用したりするため、高速な計算が求められる。

・複数の計算資源(ワーカー)を使用し、並列的にニューラルネットを構成することで、効率の良い学習を行いたい。

・データ並列化、モデル並列化、GPU による高速技術は不可欠である。

・データ並列化には同期型、非同期型が存在する。

・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。
・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。
・モデル並列化については、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。
・GPU は CPU と比較し低性能なコアが多数で構成される。簡単な並列処理が得意であり、ニューラルネットの学習は単純な行列演算が多いので、高速化が可能となる。
・GPGPU (General-purpose on GPU)は元々の使用目的であるグラフィック以外の用途で使用される GPU の総称
・軽量化の手法として、量子化・蒸留・プルーニングがある。

3.2　考察

・モデルの軽量化は計算資源の節約につながり、モバイル機器や IoT 機器等の計算資源が比較的乏しいハードでは必須のチューニングである。
・AI 以外のプログラムと同様、ハードウエア、ソフトウエア、その組み合わせを考えてチューニングを進める必要がある。

4　応用技術

4.1　要点のまとめ

【MobileNet】
・ディープラーニングモデルは精度は良いが、その分ネットワークが深くなり計算量が増える。
・計算量が増えると、多くの計算リソースが必要で、お金がかかってしまう。
・ディープラーニングモデルの軽量化・高速化・高精度化を実現(その名の通りモバイルなネットワーク
・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。
【DenseNet】
・CNN アーキテクチャの一種
・ニューラルネットワークでは層が深くなるにつれて、学習が難しくなるという問題があったが、ResNet などの CNN アーキテクチャでは前方の層から後方の層へアイデンティティ接続を介してパスを作ることで問題を対処
・DenseBlock と呼ばれるモジュールを用いた、DenseNet もそのようなアーキテクチ

ャの一つである

【正規化】

・BatchNorm は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

・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

・BatchNorm 以外の成果手法として、Layer Norm、InstanceNorm 等がある。

【WaveNet】

・生の音声波形を生成する深層学習プログラムである。

・Pixel CNN を音声に応用したものである。

4.2 考察

・応用技術が進むことで、音声や映像判別にも AI が利用されるようになり、今まで人間でした判定できなかったことができるようになっており、ますます利用できる領域が広がっていくと認識した。

5 Transformer

5.1 要点のまとめ

・2017 年 6 月に登場。

・RNN を使わず、必要なのは Attention だけ

・当時の SOTA をはるかに少ない計算量で実現。英仏（3600 万文）の学習を 8GPU で 3.5 日で完了

・注意機構には「ソース・ターゲット注意機構」「自己注意機」の 2 種類ある

・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 が肝：入力を全て同じにして学習的に注意箇所を決めていく

・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

・Decoder：Encoder と同じく 6 層、自己注意機構、Encoder-Decoder attention

・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!18

・Position Encoding について、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。

5.2 演習結果

5. 評価

```
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, device=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_attns
```

```
[42] 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
```

```
[43] # 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)

     <All keys matched successfully>
```

```
[44] # 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]
```

```
[43] # 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)
```

```
<All keys matched successfully>
```

```
# 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]
```

生成

```
[47] test_dataloader = DataLoader(
         test_X, test_Y, 1,
         shuffle=False
         )
```

```
[48] 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: show your own business .
tgt: 自分 の 事 を しろ 。
out: 自分 の 仕事 に は 自分 の <UNK> し て い る 。
```

6 物体検知・セグメンテーション

6.1 要点のまとめ

・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

・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

・Semantic Segmentation

Up-sampling は受容野を広げるための処理

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

受容野を広げる典型①深い Conv.層 ②プーリング（ストライド）

6.2　考察

・受容野を広げることで、より広い視点から俯瞰的に判断することができるというのは、人間の物の見方につながるものがある。