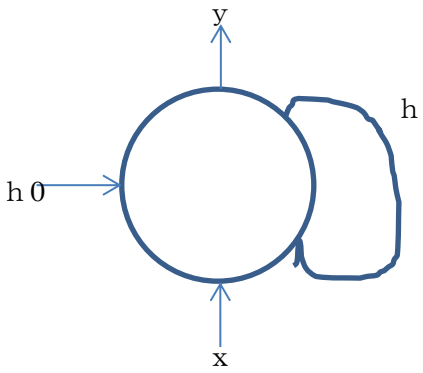
目　次

内容

1．　section1_再起型ニューラルネットワークの概念

再起型ニューラルネットワークの構成




```python
import numpy as np

Class RNN:
    def __init__(self,Wx,Wh,h0):
        """
        Wx : 入力xに係る重み(1,隠れ層のノード数)
        Wh : 1時間前のhにかかる重み(隠れ層のノード数,隠れ層のノード数)
        h0 : 隠れ層の初期値(データ数,隠れ層のノード数)
        """
        # パラメータのリスト
        self.params = {Wx, Wh}
        # 隠れ層の初期値を設定
        self.h_prev = h0

    def forward(self,X):
        """
        順伝播計算
        X : 入力データ数
        """
        Wx,Wh = self.params
        h_prev = self.h_prev

        t = np.dot(h_prev,Wh) + np.dot(x,Wx)
        # 活性化関数は恒等写像関数とする
        h_next = t
        # 隠れ層の状態の保存
        self.h_prev = h_next

        return h_next

```

2. section2_LSTM

```python
import numpy as np

def forward(X,h_prev,c_prev,Wx,Wh,b):
    """
    順伝播計算
    X: 入力(データ数,隠れ層のノード数)
    h_prev: 前時刻の隠れ層の出力(データ数,隠れ層のノード数)
    c_prev: 前時刻のメモリーの状態(データ数,隠れ層のノード数)
    Wx: 入力x用の重みパラメータ(特徴量の数,4X隠れ層のノード数)
    Wh: 隠れ状態h用の重みパラメータ(隠れ層のノード数,4X隠れ層のノード数)
    b: バイアス(4X隠れ層のノード数)
    """
    N,H = h_prev.shape
    A = np.dot(x,Wx) + np.dot(h_prev,Wh) + b

    f = A[:,:H]
    g = A[:,H:2*H]
    i = A[:,2*H:3*H]
    o = A[:,3*H:]

    f = sigmoid(f)
    g = np.tanh(g)
    i = sigmoid(i)
    o = sigmoid(o)

    print(f.shape,c_prev.shape,g.shape,i.shape)
    c_next = f*c_prev+g*i
    h_next = o*np.tanh(c_next)

    return h_next,c_next

```

3． section3_GRU

```python
import keras
from keras.models import Sequetial
from keras.layers import Dense
from keras.layers.recurrent import GRU
import numpy as np
import pandas as pd
import matplotlib
matprotlib.use('Agg')
import matplotlib.pyplot as plt
import sys
import os
import time

#時系列
n_rnn = 7
#バッチサイズ設定
n_bs = 4
#中間層ニューロン数設定
n_units = 20
#出力層ニューロン数設定
n_out = 1
#ドロップアウト率
r_dropout = 0.0
#エポック数
nb_epochs = 1000
#学習データセットcsvパス設定
csvfile = 'dataset/temp/train.csv'
def main():
    #環境設定（ディスプレイの出力先をlocalhostにする）
    os.environ['DISPLAY'] = '0'
    #時間計測開始
    start = time.time()
    #コマンド引数確認
    if len(sys.argv) != 2:
        print('使用法: python3 本ファイル名.py モデルファイル名.h5')
        sys.exit()
    #学習モデルファイルパス取得
    savefile = sys.arg[1]
    #データセットファイル読み込み
    data = len(data) - n_rnn
    #入力データ定義
    x = np.zeros((n_samples,n_rnn))
    #正解データ定義
    t = np.zeros((n_samples,))
    for i in range(0, n_samples):
        x[i] = data[i:i+n_rnn]
        #正解データは1単位時刻後の値
        t[i] = data[i+n_rnn]
    #print(x)
    #print(t)
    #Keras向けにxとtを整形
    x = x.reshape(n_samples,n_rnn,1)
    t = t.reshape(n_samples,1)
    #print(x.shape)
    #print(t.shape)
    #データシャッフル
    p = np.random.permutation(n_samples)
    x = x[p]
    t = t[p]
    #モデル作成（既存モデルがある場合は読み込んで再学習。なければ新規作成）
    if os.path.exist(savefile):
        print('モデル再学習')
        rnn_model = keras.models.load_model(savefile)
    else:
        rnn_model = rnn_model_maker(n_samples,n_out)
    #モデル構造の確認
    rnn_model.summary()
    #モデルの学習
    history = rnn.model.fit(x,t,epochs=nb_epochs,validation_split=0.1,batch_size=n_bs,verbos=2)
    #学習結果を保存
```

```
    #学習結果を保存
    rnn_model.save(savefile)
    #学習所要時間の計算、表示
    process_time = (time.time()-start)/60
    print('process_time=',process_time,'[min]')
    #損失関数の時系列変化をグラフ表示
    plot_loss(history)
def load_csv(csvfile):
    #csvをロードし変数に格納
    df = pd.read_csv(csvfile)
    dfv = df.values.astype(np.float64)
    n_dfv = dfv.shape[1]
    data = dfv[:,np.array((n_dfv-1))]
    #print(data)
    #データの標準化
    data = (data-data.mean())/data.std()
    #print(data)
    return data
def rnn_model_maker(n_samples,n_out):
    #3層RNN(リカレントネットワーク)を定義
    model = Sequential()
    #中間層(RNN)を定義
    model.add(GRU(units=n_units,input_shape=(n_rnn,1),dropout=r_dropout,return_sequences=False))
    #出力層を定義(ニューロン数は1個)
    model.add(Dense(units=n_out,activation='lenear'))
    #回帰学習モデル作成
    model.compile(loss='mean_squared_error',optimizer='rmsprop')
    #モデルを返す
    return model
```

```
def plot_loss(history):
    #損失関数のグラフの軸ラベルを設定
    plot.xlabel('time step')
    plot.ylabel('loss')
    #グラフ縦軸の範囲を0以上と定める
    plot.ylim(0,max(np.r_[history.history['val_loss'],history.history['loss']]))
    #損失関数の時間変化を描画
    val_loss = plt.plot(history.history['val_loss'],c='#56B4E9')
    loss, = plt.plot(history.history['loss'],c='#E69F00')
    #グラフの(はんれい)を追加
    plt.legend([loss,val_loss],['loss','val_loss'])
    #描画したグラフを表示
    #plt.show()
    #グラフを保存
    plt.savefig('train_figure.png')
if __name__ == '__main__':
    main()
```

4． section4_双方向 RNN

```python
class TimeBiLSTM:
    def __init__(self, Wx1, Wh1, b1,
                 Wx2, Wh2, b2, stateful=False):
        self.forward_lstm = TimeLSTM(Wx1, Wh1, b1, stateful)
        self.backward_lstm = TimeLSTM(Wx2, Wh2, b2, stateful)
        self.params = self.forward_lstm.params + self.backward_lstm.params
        self.grads = self.forward_lstm.grads + self.backward_lstm.grads

    def forward(self, xs):
        o1 = self.forward_lstm.forward(xs)
        o2 = self.backward_lstm.forward(xs[:, ::-1])
        o2 = o2[:, ::-1]
        out = np.concatenate((o1, o2), axis=2)
        return out

    def backward(self, dhs):
        H = dhs.shape[2] // 2
        do1 = dhs[:, :, :H]
        do2 = dhs[:, :, H:]

        dxs1 = self.forward_lstm.backward(do1)
        do2 = do2[:, ::-1]
        dxs2 = self.backward_lstm.backward(do2)
        dxs2 = dxs2[:, ::-1]
        dxs = dxs1 + dxs2
        return dxs


```

5. section5_Seq2Seq

```python
import random
import torch
import torch.nn as nn
from torch import optim
import torch.nn.functional as F
SOS_token = 0
EOS_token = 1
device = "cuda" # torch.device("cuda" if torch.cuda.is_available() else "cpu")
class Lang:
    def __init__( self, filename ):
        self.filename = filename
        self.word2index = {}
        self.word2count = {}
        self.sentences = []
        self.index2word = { 0: "SOS", 1: "EOS" }
        self.n_words = 2  # Count SOS and EOS
        with open( self.filename ) as fd:
            for i, line in enumerate( fd.readlines() ):
                line = line.strip()
                self.sentences.append( line )
        self.allow_list = [ True ] * len( self.sentences )
        self.target_sentences = self.sentences[ :: ]
    def get_sentences( self ):
        return self.sentences[ :: ]
    def get_sentence( self, index ):
        return self.sentences[ index ]
    def choice( self ):
        while True:
            index = random.randint( 0, len( self.allow_list ) - 1 )
            if self.allow_list[ index ]:
                break
        return self.sentences[ index ], index
```

```python
    def get_allow_list( self, max_length ):
        allow_list = []
        for sentence in self.sentences:
            if len( sentence.split() ) < max_length:
                allow_list.append( True )
            else:
                allow_list.append( False )
        return allow_list
    def load_file( self, allow_list = [] ):
        if allow_list:
            self.allow_list = [x and y for (x,y) in zip( self.allow_list, allow_list ) ]
        self.target_sentences = []
        for i, sentence in enumerate( self.sentences ):
            if self.allow_list[ i ]:
                self.addSentence( sentence )
                self.target_sentences.append( sentence )
    def addSentence( self, sentence ):
        for word in sentence.split():
            self.addWord(word)
    def addWord( self, word ):
        if word not in self.word2index:
            self.word2index[ word ] = self.n_words
            self.word2count[ word ] = 1
            self.index2word[ self.n_words ] = word
            self.n_words += 1
        else:
            self.word2count[word] += 1
```

```
60  def tensorFromSentence( lang, sentence ):
61      indexes = [ lang.word2index[ word ] for word in sentence.split(' ') ]
62      indexes.append( EOS_token )
63      return torch.tensor( indexes, dtype=torch.long ).to( device ).view(-1, 1)
64  def tensorsFromPair( input_lang, output_lang ):
65      input_sentence, index = input_lang.choice()
66      output_sentence       = output_lang.get_sentence( index )
67      input_tensor  = tensorFromSentence( input_lang, input_sentence )
68      output_tensor = tensorFromSentence( output_lang, output_sentence )
69      return (input_tensor, output_tensor)
70
```

```
71  class Encoder( nn.Module ):
72      def __init__( self, input_size, embedding_size, hidden_size ):
73          super().__init__()
74          self.hidden_size = hidden_size
75          # 単語をベクトル化する。1単語はembedding_sie次元のベクトルとなる
76          self.embedding   = nn.Embedding( input_size, embedding_size )
77          # GRUに依る実装.
78          self.gru         = nn.GRU( embedding_size, hidden_size )
79      def initHidden( self ):
80          return torch.zeros( 1, 1, self.hidden_size ).to( device )
81      def forward( self, _input, hidden ):
82          # 単語のベクトル化
83          embedded        = self.embedding( _input ).view( 1, 1, -1 )
84          # ベクトル化したデータをGRUに噛ませる。通常のSeq2Seqでは出力outは使われることはない。
85          # ただしSeq2Seq + Attentionをする場合にはoutの値を使うことになるので、リターンする
86          out, new_hidden = self.gru( embedded, hidden )
87          return out, new_hidden
88
```

```
89  class Decoder( nn.Module ):
90      def __init__( self, hidden_size, embedding_size, output_size ):
91          super().__init__()
92          self.hidden_size = hidden_size
93          # 単語をベクトル化する。1単語はembedding_sie次元のベクトルとなる
94          self.embedding   = nn.Embedding( output_size, embedding_size )
95          # GRUによる実装(RNN素子の一種)
96          self.gru         = nn.GRU( embedding_size, hidden_size )
97          # 全結合して1層のネットワークにする
98          self.linear          = nn.Linear( hidden_size, output_size )
99          # softmaxのLogバージョン。dim=1で行方向を確率変換する(dim=0で列方向となる)
100         self.softmax     = nn.LogSoftmax( dim = 1 )
101     def forward( self, _input, hidden ):
102         # 単語のベクトル化。GRUの入力に合わせ三次元テンソルにして渡す。
103         embedded            = self.embedding( _input ).view( 1, 1, -1 )
104         # relu活性化関数に突っ込む( 3次元のテンソル)
105         relu_embedded       = F.relu( embedded )
106         # GRU関数( 入力は3次元のテンソル )
107         gru_output, hidden = self.gru( relu_embedded, hidden )
108         # softmax関数の適用。outputは3次元のテンソルなので2次元のテンソルを渡す
109         result              = self.softmax( self.linear( gru_output[ 0 ] ) )
110         return result, hidden
111     def initHidden( self ):
112         return torch.zeros( 1, 1, self.hidden_size ).to( device )
```

6． section6_Word2vec

```
from janome.tokenizer import Tokenizer
tokenizer = Tokenizer()

sentences = text.split("。")
tokenizer = Tokenizer()

word_list = []
for sentence in sentences:
    tokens = tokenizer.tokenize(sentence)
    temp_word = []
    for token in tokens:
        if token.part_of_speech.split(",")[0] in ["名詞","動詞"]:
            temp_word.append(token.base_form)
    word_list.append(temp_word)

import gensim
model = gensim.models.word2vec.Word2Vec(word_list, size=100, min_count=1, window=5, iter=100)

# 単語の足し算、引き算
ret = model.wv.most_similar(
    positive = ['吾輩','猫'],
    negative = '猫'
)

for item in ret:
    print(item[0], item[1])

# 単語の類似度 cos類似度のため -1〜1
a = model.wv.similarity('吾輩','猫')
print(a)
```

7．　section7_Attention Mechanism

```
# Attention Decoderクラス
class AttentionDecoder(nn.Module):
    def __init__(self, vocab_size, embedding_dim, hidden_dim, batch_size):
        super(AttentionDecoder, self).__init__()
        self.hidden_dim = hidden_dim
        self.batch_size = batch_size
        self.word_embeddings = nn.Embedding(vocab_size, embedding_dim, padding_idx=char2id[" "])
        self.gru = nn.GRU(embedding_dim, hidden_dim, batch_first=True)
        # hidden_dim*2としているのは、各系列のGRUの隠れ層とAttention層で計算したコンテキストベクトルをtorch.catでつなぎ
        self.hidden2linear = nn.Linear(hidden_dim * 2, vocab_size)
        # 列方向を確率変換したいのでdim=1
        self.softmax = nn.Softmax(dim=1)

    def forward(self, sequence, hs, h):
        embedding = self.word_embeddings(sequence)
        output, state = self.gru(embedding, h)

        # Attention層
        # hs.size() = ([100, 29, 128])
        # output.size() = ([100, 10, 128])

        # bmmを使ってEncoder側の出力(hs)とDecoder側の出力(output)をbatchごとまとめて行列計算するために、Decoder側の
        t_output = torch.transpose(output, 1, 2) # t_output.size() = ([100, 128, 10])

        # bmmでバッチも考慮してまとめて行列計算
        s = torch.bmm(hs, t_output) # s.size() = ([100, 29, 10])

        # 列方向(dim=1)でsoftmaxをとって確率表現に変換
        # この値を後のAttentionの可視化などにも使うため、returnで返しておく
        attention_weight = self.softmax(s) # attention_weight.size() = ([100, 29, 10])

        # コンテキストベクトルをまとめるために入れ物を用意
        c = torch.zeros(self.batch_size, 1, self.hidden_dim, device=device) # c.size() = ([100, 1,

        # 各DecoderのGRU層に対するコンテキストベクトルをまとめて計算する方法がわからなかったので、
        # 各層(Decoder側のGRU層は生成文字列が10文字なので10個ある)におけるattention weightを取り出してforループ
        # バッチ方向はまとめて計算できたのでバッチはそのまま
        for i in range(attention_weight.size()[2]): # 10回ループ

            # attention_weight[:,:,i].size() = ([100, 29])
            # i番目のGRU層に対するattention weightを取り出すが、テンソルのサイズをhsと揃えるためにunsqueezeする
            unsq_weight = attention_weight[:,:,i].unsqueeze(2) # unsq_weight.size() = ([100, 29, 1])

            # hsの各ベクトルをattention weightで重み付けする
            weighted_hs = hs * unsq_weight # weighted_hs.size() = ([100, 29, 128])

            # attention weightで重み付けされた各hsのベクトルをすべて足し合わせてコンテキストベクトルを作成
            weight_sum = torch.sum(weighted_hs, axis=1).unsqueeze(1) # weight_sum.size() = ([100, 1,

            c = torch.cat([c, weight_sum], dim=1) # c.size() = ([100, i, 128])

        # 箱として用意したzero要素が残っているのでスライスして削除
        c = c[:,1:,:]

        output = torch.cat([output, c], dim=2) # output.size() = ([100, 10, 256])
        output = self.hidden2linear(output)
        return output, state, attention_weight
```