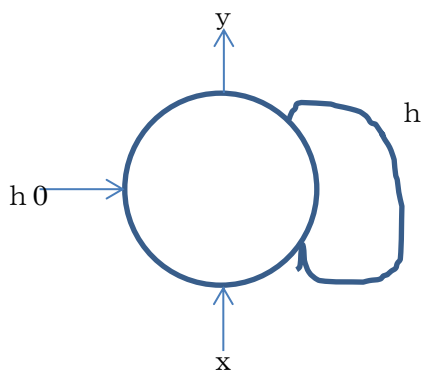


1. section1_再起型ニューラルネットワークの概念

再起型ニューラルネットワークの構成



```
1  import numpy as np
2
3  class RNN:
4      def __init__(self, Wx, Wh, h0):
5          """
6          Wx : 入力xに係る重み(1, 隠れ層のノード数)
7          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)
8          h0 : 隠れ層の初期値(データ数, 隠れ層のノード数)
9          """
10         # パラメータのリスト
11         self.params = {Wx, Wh}
12         # 隠れ層の初期値を設定
13         self.h_prev = h0
14
15     def forward(self, x):
16         """
17         順伝播計算
18         x : 入力データ数
19         """
20         Wx, Wh = self.params
21         h_prev = self.h_prev
22
23         t = np.dot(h_prev, Wh) + np.dot(x, Wx)
24         # 活性化関数は恒等写像関数とする
25         h_next = t
26         # 隠れ層の状態の保存
27         self.h_prev = h_next
28
29         return h_next
30
```

2. section2_LSTM

```
1  import numpy as np
2
3  def forward(X,h_prev,c_prev,Wx,Wh,b):
4      """
5      順伝播計算
6      X: 入力(データ数,隠れ層のノード数)
7      h_prev: 前時刻の隠れ層の出力(データ数, 隠れ層のノード数)
8      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)
9      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)
10     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)
11     b: バイアス(4x隠れ層のノード数)
12     """
13     N,H = h_prev.shape
14     A = np.dot(x,Wx) + np.dot(h_prev,Wh) + b
15
16     f = A[:, :H]
17     g = A[:, H:2*H]
18     i = A[:, 2*H:3*H]
19     o = A[:, 3*H:]
20
21     f = sigmoid(f)
22     g = np.tanh(g)
23     i = sigmoid(i)
24     o = sigmoid(o)
25
26     print(f.shape,c_prev.shape,g.shape,i.shape)
27     c_next = f*c_prev+g*i
28     h_next = o*np.tanh(c_next)
29
30     return h_next,c_next
31
```

3. section3_GRU

```
1 import keras
2 from keras.models import Sequential
3 from keras.layers import Dense
4 from keras.layers.recurrent import GRU
5 import numpy as np
6 import pandas as pd
7 import matplotlib
8 matplotlib.use('Agg')
9 import matplotlib.pyplot as plt
10 import sys
11 import os
12 import time
13
14 #時系列
15 n_rnn = 7
16 #バッチサイズ設定
17 n_bs = 4
18 #中間層ニューロン数設定
19 n_units = 20
20 #出力層ニューロン数設定
21 n_out = 1
22 #ドロップアウト率
23 r_dropout = 0.0
24 #エポック数
25 nb_epochs = 1000
26 #学習データセットcsvパス設定
27 csvfile = 'dataset/temp/train.csv'
28 def main():
29     #環境設定(ディスプレイの出力先をlocalhostにする)
30     os.environ['DISPLAY'] = '0'
31     #時間計測開始
32     start = time.time()
33     #コマンド引数確認
34     if len(sys.argv) != 2:
35         print('使用方法: python3 本ファイル名.py モデルファイル名.h5')
36         sys.exit()
37     #学習モデルファイルパス取得
```

```
37 #学習モデルファイルパス取得
38 savefile = sys.argv[1]
39 #データセットファイル読み込み
40 data = len(data) - n_rnn
41 #入力データ定義
42 x = np.zeros((n_samples, n_rnn))
43 #正解データ定義
44 t = np.zeros((n_samples,))
45 for i in range(0, n_samples):
46     x[i] = data[i:i+n_rnn]
47     #正解データは1単位時刻後の値
48     t[i] = data[i+n_rnn]
49 #print(x)
50 #print(t)
51 #Keras向けにxとtを整形
52 x = x.reshape(n_samples, n_rnn, 1)
53 t = t.reshape(n_samples, 1)
54 #print(x.shape)
55 #print(t.shape)
56 #データシャッフル
57 p = np.random.permutation(n_samples)
58 x = x[p]
59 t = t[p]
60 #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)
61 if os.path.exists(savefile):
62     print('モデル再学習')
63     rnn_model = keras.models.load_model(savefile)
64 else:
65     rnn_model = rnn_model_maker(n_samples, n_out)
66 #モデル構造の確認
67 rnn_model.summary()
68 #モデルの学習
69 history = rnn_model.fit(x, t, epochs=nb_epochs, validation_split=0.1, batch_size=n_bs, verbose=2)
70 #学習結果を保存
```

```

70 #学習結果を保存
71 rnn_model.save(savefile)
72 #学習所要時間の計算、表示
73 process_time = (time.time()-start)/60
74 print('process time=',process_time,'[min]')
75 #損失関数の時系列変化をグラフ表示
76 plot_loss(history)
77 def load_csv(csvfile):
78     #csvをロードし変数に格納
79     df = pd.read_csv(csvfile)
80     dfv = df.values.astype(np.float64)
81     n_dfv = dfv.shape[1]
82     data = dfv[:,np.array((n_dfv-1))]
83     #print(data)
84     #データの標準化
85     data = (data-data.mean())/data.std()
86     #print(data)
87     return data
88 def rnn_model_maker(n_samples,n_out):
89     #3層RNN(リカレントネットワーク)を定義
90     model = Sequential()
91     #中間層(RNN)を定義
92     model.add(GRU(units=n_units,input_shape=(n_rnn,1),dropout=r_dropout,return_sequences=False))
93     #出力層を定義(ニューロン数は1個)
94     model.add(Dense(units=n_out,activation='lencar'))
95     #回帰学習モデル作成
96     model.compile(loss='mean_squared_error',optimizer='rmsprop')
97     #モデルを返す
98     return model

```

```

99 def plot_loss(history):
100     #損失関数のグラフの軸ラベルを設定
101     plot.xlabel('time step')
102     plot.ylabel('Loss')
103     #グラフ縦軸の範囲を0以上と定める
104     plot.ylim(0,max(np.r_[history.history['val_loss'],history.history['loss']]))
105     #損失関数の時間変化を描画
106     val_loss = plt.plot(history.history['val_loss'],c='#56B4E9')
107     loss, = plt.plot(history.history['loss'],c='#E69F00')
108     #グラフの(はなれい)を追加
109     plt.legend([loss,val_loss],['loss','val_loss'])
110     #描画したグラフを表示
111     #plt.show()
112     #グラフを保存
113     plt.savefig('train_figure.png')
114     if __name__ == '__main__':
115         main()
116

```

4. section4_双向 RNN

```
1  class TimeBiLSTM:
2      def __init__(self, Wx1, Wh1, b1,
3                  Wx2, Wh2, b2, stateful=False):
4          self.forward_lstm = TimeLSTM(Wx1, Wh1, b1, stateful)
5          self.backward_lstm = TimeLSTM(Wx2, Wh2, b2, stateful)
6          self.params = self.forward_lstm.params + self.backward_lstm.params
7          self.grads = self.forward_lstm.grads + self.backward_lstm.grads
8
9      def forward(self, xs):
10         o1 = self.forward_lstm.forward(xs)
11         o2 = self.backward_lstm.forward(xs[:, ::-1])
12         o2 = o2[:, ::-1]
13         out = np.concatenate((o1, o2), axis=2)
14         return out
15
16     def backward(self, dhs):
17         H = dhs.shape[2] // 2
18         do1 = dhs[:, :, :H]
19         do2 = dhs[:, :, H:]
20
21         dxs1 = self.forward_lstm.backward(do1)
22         do2 = do2[:, ::-1]
23         dxs2 = self.backward_lstm.backward(do2)
24         dxs2 = dxs2[:, ::-1]
25         dxs = dxs1 + dxs2
26         return dxs
27
28
```

5. section5_Seq2Seq

```

1  import random
2  import torch
3  import torch.nn as nn
4  from torch import optim
5  import torch.nn.functional as F
6  SOS_token = 0
7  EOS_token = 1
8  device = "cuda" # torch.device("cuda" if torch.cuda.is_available() else "cpu")
9  class Lang:
10     def __init__( self, filename ):
11         self.filename = filename
12         self.word2index = {}
13         self.word2count = {}
14         self.sentences = []
15         self.index2word = { 0: "SOS", 1: "EOS" }
16         self.n_words = 2 # Count SOS and EOS
17         with open( self.filename ) as fd:
18             for i, line in enumerate( fd.readlines() ):
19                 line = line.strip()
20                 self.sentences.append( line )
21         self.allow_list = [ True ] * len( self.sentences )
22         self.target_sentences = self.sentences[ :: ]
23     def get_sentences( self ):
24         return self.sentences[ :: ]
25     def get_sentence( self, index ):
26         return self.sentences[ index ]
27     def choice( self ):
28         while True:
29             index = random.randint( 0, len( self.allow_list ) - 1 )
30             if self.allow_list[ index ]:
31                 break
32         return self.sentences[ index ], index

```

```

33     def get_allow_list( self, max_length ):
34         allow_list = []
35         for sentence in self.sentences:
36             if len( sentence.split() ) < max_length:
37                 allow_list.append( True )
38             else:
39                 allow_list.append( False )
40         return allow_list
41     def load_file( self, allow_list = [] ):
42         if allow_list:
43             self.allow_list = [x and y for (x,y) in zip( self.allow_list, allow_list ) ]
44             self.target_sentences = []
45             for i, sentence in enumerate( self.sentences ):
46                 if self.allow_list[ i ]:
47                     self.addSentence( sentence )
48                     self.target_sentences.append( sentence )
49     def addSentence( self, sentence ):
50         for word in sentence.split():
51             self.addWord(word)
52     def addWord( self, word ):
53         if word not in self.word2index:
54             self.word2index[ word ] = self.n_words
55             self.word2count[ word ] = 1
56             self.index2word[ self.n_words ] = word
57             self.n_words += 1
58         else:
59             self.word2count[word] += 1
60     def tensorFromSentences( self, sentences ):

```

```

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_size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_size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

```
1  from janome.tokenizer import Tokenizer
2  tokenizer = Tokenizer()
3
4  sentences = text.split(".")
5  tokenizer = Tokenizer()
6
7  word_list = []
8  for sentence in sentences:
9      tokens = tokenizer.tokenize(sentence)
10     temp_word = []
11     for token in tokens:
12         if token.part_of_speech.split(",")[0] in ["名詞", "動詞"]:
13             temp_word.append(token.base_form)
14     word_list.append(temp_word)
15
16 import gensim
17 model = gensim.models.word2vec.Word2Vec(word_list, size=100, min_count=1, window=5, iter=100)
18
19 # 単語の足し算、引き算
20 ret = model.wv.most_similar(
21     positive = ['吾輩', '猫'],
22     negative = '猫'
23 )
24
25 for item in ret:
26     print(item[0], item[1])
27
28 # 単語の類似度 cos類似度のため -1~1
29 a = model.wv.similarity('吾輩', '猫')
30 print(a)
31
```


7. section7_Attention Mechanism

```

79 # Attention Decoderクラス
80 class AttentionDecoder(nn.Module):
81     def __init__(self, vocab_size, embedding_dim, hidden_dim, batch_size):
82         super(AttentionDecoder, self).__init__()
83         self.hidden_dim = hidden_dim
84         self.batch_size = batch_size
85         self.word_embeddings = nn.Embedding(vocab_size, embedding_dim, padding_idx=char2id[" "])
86         self.gru = nn.GRU(embedding_dim, hidden_dim, batch_first=True)
87         # 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
88         self.hidden2linear = nn.Linear(hidden_dim * 2, vocab_size)
89         # 列方向を確率変換したいのでdim=1
90         self.softmax = nn.Softmax(dim=1)
91
92     def forward(self, sequence, hs, h):
93         embedding = self.word_embeddings(sequence)
94         output, state = self.gru(embedding, h)
95
96         # Attention層
97         # hs.size() = ([100, 29, 128])
98         # output.size() = ([100, 10, 128])
99
100         # 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
101         t_output = torch.transpose(output, 1, 2) # t_output.size() = ([100, 128, 10])
102
103         # bmmでバッチも考慮してまとめて行列計算
104         s = torch.bmm(hs, t_output) # s.size() = ([100, 29, 10])
105
106         # 列方向(dim=1)でsoftmaxをとって確率表現に変換
107         # 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
108         attention_weight = self.softmax(s) # attention_weight.size() = ([100, 29, 10])
109
110         # コンテキストベクトルをまとめるために入れ物を用意
111         c = torch.zeros(self.batch_size, 1, self.hidden_dim, device=device) # c.size() = ([100, 1,
112
113         # 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ため、
114         # 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
115         # 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
116         for i in range(attention_weight.size()[2]): # 10回ループ
117
118             # attention_weight[:, :, i].size() = ([100, 29])
119             # 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
120             unsq_weight = attention_weight[:, :, i].unsqueeze(2) # unsq_weight.size() = ([100, 29, 1])
121
122             # hsの各ベクトルをattention_weightで重み付けする
123             weighted_hs = hs * unsq_weight # weighted_hs.size() = ([100, 29, 128])
124
125             # 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
126             weight_sum = torch.sum(weighted_hs, axis=1).unsqueeze(1) # weight_sum.size() = ([100, 1,
127
128             c = torch.cat([c, weight_sum], dim=1) # c.size() = ([100, i, 128])
129
130         # 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
131         c = c[:, 1:, :]
132
133         output = torch.cat([output, c], dim=2) # output.size() = ([100, 10, 256])
134         output = self.hidden2linear(output)
135         return output, state, attention_weight
136

```