ラピッドチャレンジ「課題レポート (深層学習 Day3)」

佐藤晴一

2021 年 6 月 6 日

目次

第 1 章

再帰型ニューラルネットワークの概念

1.1 再帰型ニューラルネットワーク (RNN) 全体像

1.1.1 RNN(Reccurent Neural Network) 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。

1.1.2 時系列データ

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

具体的な時系列データとの一例としては、・音声データ　・株価変動データ　・テキストデータ...etc

1.1.3 RNN について

ニューラルネットや深層学習などの機械学習モデルは基本的にこの入力層～中間層（データの特徴量を抽出）～出力層の構造を持つ。特に RNN では、これまでのデータ（計算）フローと異なる点は、中間層からの出力を出力層に送ると同時に、次の（時間の）入力層に加えるという点にある。下図で示した 3 種類の重みを optimize していくことになる。これまでになかった（前の時間の）中間層からの出力を重み W で次の時間の中間層のインプットへつないでいる点がポイントである。

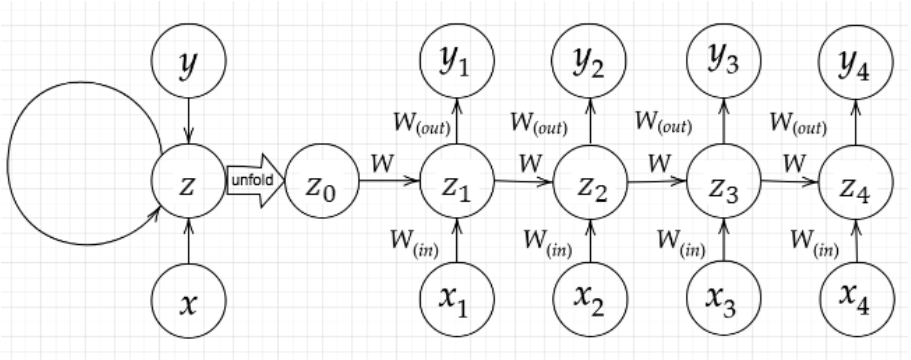


図 1.1　RNN 全体像

■RNN の数学的記述

$$
\begin{aligned}
u^t &= W_{(\mathrm{in})}x^t + Wz^{t-1} + b && \text{(入力層〜中間層へのインプット)} \\
z^t &= f(u^t) = f(W_{(\mathrm{in})}x^t + Wz^{t-1} + b) && \text{(活性化関数を通じた中間層からの出力)} \\
v^t &= W_{(\mathrm{out})}z^t + c && \text{(中間層〜出力層へのインプット)} \\
y^t &= g(v^t) = g(W_{(\mathrm{out})}x^t + c) && \text{(活性化関数を通じた出力層からの出力)}
\end{aligned}
$$

Listing 1.1　RNN -discription for math

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

■確認テスト　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。

■答え　前の中間層から次の中間層へいたる重み W のこと。RNN の骨幹をなす重要な重みである。

■RNN の特徴　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。

■実装演習成果

Listing 1.2　RNN(バイナリ加算)

```
# グラフの日本語表記化
!pip install japanize-matplotlib

import numpy as np
from common import functions
import matplotlib.pyplot as plt
import japanize_matplotlib

# def d_tanh(x):
# データを用意
# 進数の桁数 2
binary_dim = 8
# 最大値+ 1
largest_number = pow(2, binary_dim)
# まで進数を用意 largest_number2
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
```

```
21
22 weight_init_std = 1
23 learning_rate = 0.1
24
25 iters_num = 10000
26 plot_interval = 100
27
28 # ウェイト初期化バイアスは簡単のため省略()
29 W_in = weight_init_std * np.random.randn(input_layer_size, hidden_layer_size)
30 W_out = weight_init_std * np.random.randn(hidden_layer_size, output_layer_size)
31 W = weight_init_std * np.random.randn(hidden_layer_size, hidden_layer_size)
32
33 # Xavier
34
35 # He
36
37 # 勾配
38 W_in_grad = np.zeros_like(W_in)
39 W_out_grad = np.zeros_like(W_out)
40 W_grad = np.zeros_like(W)
41
42 u = np.zeros((hidden_layer_size, binary_dim + 1))
43 z = np.zeros((hidden_layer_size, binary_dim + 1))
44 y = np.zeros((output_layer_size, binary_dim))
45
46 delta_out = np.zeros((output_layer_size, binary_dim))
47 delta = np.zeros((hidden_layer_size, binary_dim + 1))
48
49 all_losses = []
50
51 for i in range(iters_num):
52
53     # A, 初期化 B (a + b = d)
54     a_int = np.random.randint(largest_number/2)
55     a_bin = binary[a_int] # binary encoding
56     b_int = np.random.randint(largest_number/2)
57     b_bin = binary[b_int] # binary encoding
58
59     # 正解データ
60     d_int = a_int + b_int
61     d_bin = binary[d_int]
62
63     # 出力バイナリ
64     out_bin = np.zeros_like(d_bin)
65
66     # 時系列全体の誤差
```

```
67      all_loss = 0
68
69      # 時系列ループ
70      for t in range(binary_dim):
71          # 入力値
72          X = np.array([a_bin[ - t - 1], b_bin[ - t - 1]]).reshape(1, -1)
73          # 時刻における正解データ t
74          dd = np.array([d_bin[binary_dim - t - 1]])
75
76          u[:,t+1] = np.dot(X, W_in) + np.dot(z[:,t].reshape(1, -1), W)
77          z[:,t+1] = functions.sigmoid(u[:,t+1])
78
79          y[:,t] = functions.sigmoid(np.dot(z[:,t+1].reshape(1, -1), W_out))誤差
80
81          #
82          loss = functions.mean_squared_error(dd, y[:,t])
83
84          delta_out[:,t] = functions.d_mean_squared_error(dd, y[:,t]) * functions.
                d_sigmoid(y[:,t])
85
86          all_loss += loss
87
88          out_bin[binary_dim - t - 1] = np.round(y[:,t])
89
90      for t in range(binary_dim)[::-1]:
91          X = np.array([a_bin[-t-1],b_bin[-t-1]]).reshape(1, -1)
92
93          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])
94
95          # 勾配更新
96          W_out_grad += np.dot(z[:,t+1].reshape(-1,1), delta_out[:,t].reshape(-1,1))
97          W_grad += np.dot(z[:,t].reshape(-1,1), delta[:,t].reshape(1,-1))
98          W_in_grad += np.dot(X.T, delta[:,t].reshape(1,-1))
99
100     # 勾配適用
101     W_in -= learning_rate * W_in_grad
102     W_out -= learning_rate * W_out_grad
103     W -= learning_rate * W_grad
104
105     W_in_grad *= 0
106     W_out_grad *= 0
107     W_grad *= 0
108
109     if(i % plot_interval == 0):
110         all_losses.append(all_loss)
```

```
111         print("iters:" + str(i))
112         print("Loss:" + str(all_loss))
113         print("Pred:" + str(out_bin))
114         print("True:" + str(d_bin))
115         out_int = 0
116         for index,x in enumerate(reversed(out_bin)):
117             out_int += x * pow(2, index)
118         print(str(a_int) + " + " + str(b_int) + " = " + str(out_int))
119         print("------------")
120
121 lists = range(0, iters_num, plot_interval)
122 plt.plot(lists, all_losses, label="loss")
123 plt.title時系列全体の誤差 ("")
124 plt.xlabel("iteration")
125 plt.ylabel("all_loss")
126 plt.show()
```

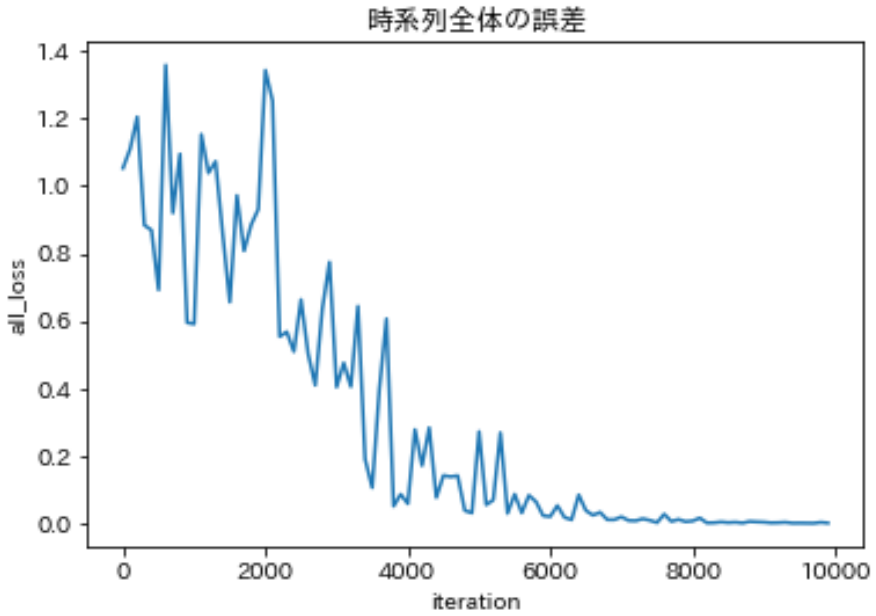


図 1.2　バイナリ加算

■演習チャレンジ　再帰型ニューラルネットワークにおいて構文木を入力として再帰的に文全体の表現ベクトルを得るプログラムである。ただし、ニューラルネットワークの重みパラメータはグローバル変数として定義してあるものとし、_activation 関数はなんらかの活性化関数であるとする。木構造は再帰的な辞書で定義してあり、root が最も外側の辞書であると仮定する。() にあてはまるのはどれか

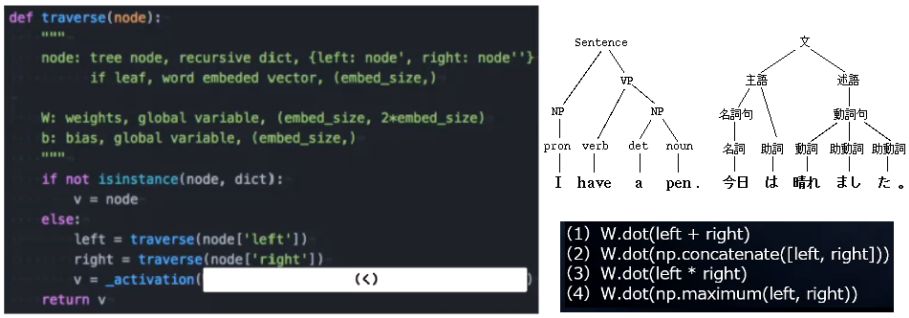


図 1.3　構文木

■答え　もともとの構文のデータの特徴を残しておくことが重要であるため、隣接単語（表現ベクトル）から表現ベクトルを作るという処理は、隣接している表現 left と right を合わせたものを特徴量としてそこに重みを掛けることで実現する。つまり、正解は (2) の W.dot(np.concatenate([left, right])) である。

ここで、表現ベクトルの特徴量を残したまま構文木の学習を進めていくと、どんどんデータが増大するような心配があるが、実際は重み W の成分を適切に設定することで、重みを掛け合わせたあとの表現ベクトルの成分数を抑制することが可能である。

1.2　BPTT

1.2.1　BPTT(Backpropagation through time) とは

BPTT は、RNN においてのパラメータ調整方法の一種（誤差逆伝播法[*1]の一種）である。この BPTT も最急降下法の一種であり、最急降下法は、関数の極小値を探し出すための一次の反復的最適化アルゴリズムである。ニューラルネットワークでは、非線形活性化関数が可微分であるという条件で、重みに関する誤差の微分係数に比例して個々の重みを変化させることによって誤差項を最小化するために使うことができる。標準的手法は「通時的誤差逆伝播法（Backpropagation through time、BPTT）」と呼ばれ、順伝播型ネットワークのための誤差逆伝播法の一般化である。誤差逆伝播法と同様に、BPTT はポントリャーギンの最小値原理（英語版）の後ろ向き連鎖（reverse accumulation）モードにおける自動微分の実例である。

■確認テスト　連鎖律の原理を使い、dz/dx を求めよ。($z = t^2$, $t = x + y$)

■答え

$$\frac{dz}{dx} = \frac{dz}{dt}\frac{dt}{dx} = 2t \cdot (1) = 2(x + y)$$

[*1] 計算結果（=誤差）から微分を逆算することで、不要な再帰的計算を避けて微分を算出できる。

1.2.2 BPIT の数学的記述

RNN では、3 種類の重みが存在する。このため誤差逆伝播法で更新（最適化）するために、誤差関数 E をそれぞれ 3 種類の重い W について偏微分する必要がある。

■RNN モデル

$$
\begin{aligned}
u^t &= W_{(\mathrm{in})}x^t + Wz^{t-1} + b && \text{(入力層〜中間層へのインプット)} \\
z^t &= f(u^t) = f(W_{(\mathrm{in})}x^t + Wz^{t-1} + b) && \text{(活性化関数を通じた中間層からの出力)} \\
v^t &= W_{(\mathrm{out})}z^t + c && \text{(中間層〜出力層へのインプット)} \\
y^t &= g(v^t) = g(W_{(\mathrm{out})}x^t + c) && \text{(活性化関数を通じた出力層からの出力)}
\end{aligned}
$$

■更新量の計算

$$
\begin{aligned}
\frac{\partial E}{\partial W_{(\mathrm{in})}} &= \frac{\partial E}{\partial u^t}\left[\frac{\partial u^t}{\partial W_{(\mathrm{in})}}\right]^T = \delta^t \left[x^t\right]^T \\
\frac{\partial E}{\partial W_{(\mathrm{out})}} &= \frac{\partial E}{\partial v^t}\left[\frac{\partial v^t}{\partial W_{(\mathrm{out})}}\right]^T = \delta^{\mathrm{out},t} \left[z^t\right]^T \\
\frac{\partial E}{\partial W} &= \frac{\partial E}{\partial u^t}\left[\frac{\partial u^t}{\partial W}\right]^T = \delta^t \left[z^{t-1}\right]^T \\
\frac{\partial E}{\partial b} &= \frac{\partial E}{\partial u^t}\frac{\partial u^t}{\partial b} = \delta^t \\
\frac{\partial E}{\partial c} &= \frac{\partial E}{\partial v^t}\frac{\partial v^t}{\partial c} = \delta^{\mathrm{out},t}
\end{aligned}
$$

ここで、大文字の T は「時間的に遡る微分」という意味である。※　行列の転置という意味ではないので注意

■数式とコード

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

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

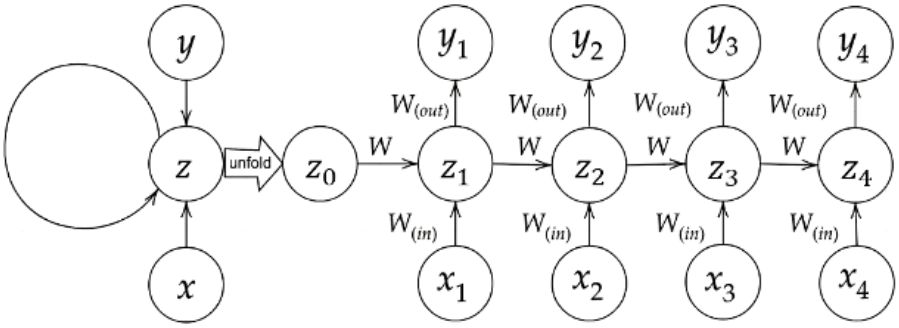
$$
\frac{\partial E}{\partial W_{(\mathrm{out})}} = \frac{\partial E}{\partial v^t}\left[\frac{\partial v^t}{\partial W_{(\mathrm{out})}}\right]^T = \delta^{\mathrm{out},t} \left[z^t\right]^T
$$

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

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

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

■確認テスト　下図の y1 を x1・z0・z1・win・w・wout 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。



■答え

$$y_1 = g(W_{out} z_1 + c), \quad \text{where} \quad z_1 = f(W_{in} x_1 + W z_0 + b) \rightarrow$$

ここで、b, c はバイアス、f, g はシグモイド関数を表している。

■BPTT の数学的記述（続き）

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

$$\delta^{t-1} = \frac{\partial E}{\partial u^{t-1}} = \frac{\partial E}{\partial u^t}\frac{\partial u^t}{\partial u^{t-1}} = \delta^t \left\{ \frac{\partial u^t}{\partial z^{t-1}}\frac{\partial z^{t-1}}{\partial u^{t-1}} \right\} = \delta^t \left\{ W f'\left(u^{t-1}\right) \right\}$$

$$\delta^{t-z-1} = \delta^{t-z} \left\{ W f'\left(u^{t-z-1}\right) \right\}$$

```
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])
```

ここで、$t-z$ の z は、中間層の出力 z を意味しているのではないことに注意。単純に z 時間前に遡ることを意味している。

■BPTT におけるパラメータ更新量　ここまでの数式をまとめると、入力層から中間層、中間層から中間層に至る重み更新には、前の時間における（過去の）情報が必要であり、式では $\sum$ で表されている。一方、中間層から出力層に至る重み更新には、前の時間における情報が加わっていないことに注意。この理由は出力層への入力となる「中間層からの活性化関数を通る」前までに、既に前の時間における情報が考慮されているからである。また ϵ は学習率であり、ハイパーパラメータである。

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

$$W_{(out)}^{t+1} = W_{(out)}^{t} - \epsilon \frac{\partial E}{\partial W_{(out)}} = \mathrm{W}_{(out)}^{t} - \epsilon \delta^{out,t} \left[z^{t} \right]^T$$

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

$$b^{t+1} = b^{t} - \epsilon \frac{\partial E}{\partial b} = b^{t} - \epsilon \sum_{z=0}^{T_t} \delta^{t-z}$$

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

Listing 1.3　パラメータ更新

```
# W_{in}
W_in -= learning_rate * W_in_grad
# W_{out}
W_out -= learning_rate * W_out_gradW -= learning_rate * W_grad
# W_{out}
W -= learning_rate * W_grad
```

1.2.3　BPIT の全体像

次式のように中間層の出力 z を通じて、再帰的に時間が前に遡って繰り返し計算 $(z \to z-1 \to z-2 \cdots)$ されていることが理解できる。

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

$$\downarrow$$

$$\begin{aligned} & W_{(in)} x^t + W\, z^{t-1} + b \\ & W_{(in)} x^t + W\, f\left(u^{t-1}\right) + b \\ & W_{(in)} x^t + W\, f\left(W_{(in)} x^{t-1} + W\, z^{t-2} + b\right) + b \end{aligned}$$

■実装演習成果

Listing 1.4 BPTT

```
# グラフの日本語表記化
!pip install japanize-matplotlib

import numpy as np
from common import functions
import matplotlib.pyplot as plt
import japanize_matplotlib

def d_tanh(x):
    return 1/(np.cosh(x) ** 2)

# データを用意
# 進数の桁数 2
binary_dim = 8
# 最大値+ 1
largest_number = pow(2, binary_dim)
# まで進数を用意 largest_number2
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)
```

```
40 # W_out = np.random.randn(hidden_layer_size, output_layer_size) / (np.sqrt(
       hidden_layer_size)) * np.sqrt(2)
41 # W = np.random.randn(hidden_layer_size, hidden_layer_size) / (np.sqrt(hidden_layer_size
       )) * np.sqrt(2)
42
43 # 勾配
44 W_in_grad = np.zeros_like(W_in)
45 W_out_grad = np.zeros_like(W_out)
46 W_grad = np.zeros_like(W)
47
48 u = np.zeros((hidden_layer_size, binary_dim + 1))
49 z = np.zeros((hidden_layer_size, binary_dim + 1))
50 y = np.zeros((output_layer_size, binary_dim))
51
52 delta_out = np.zeros((output_layer_size, binary_dim))
53 delta = np.zeros((hidden_layer_size, binary_dim + 1))
54
55 all_losses = []
56
57 for i in range(iters_num):
58
59     # A, 初期化 B (a + b = d)
60     a_int = np.random.randint(largest_number/2)
61     a_bin = binary[a_int] # binary encoding
62     b_int = np.random.randint(largest_number/2)
63     b_bin = binary[b_int] # binary encoding
64
65     # 正解データ
66     d_int = a_int + b_int
67     d_bin = binary[d_int]
68
69     # 出力バイナリ
70     out_bin = np.zeros_like(d_bin)
71
72     # 時系列全体の誤差
73     all_loss = 0
74
75     # 時系列ループ
76     for t in range(binary_dim):
77         # 入力値
78         X = np.array([a_bin[ - t - 1], b_bin[ - t - 1]]).reshape(1, -1)
79         # 時刻における正解データ t
80         dd = np.array([d_bin[binary_dim - t - 1]])
81
82         u[:,t+1] = np.dot(X, W_in) + np.dot(z[:,t].reshape(1, -1), W)
83         z[:,t+1] = functions.sigmoid(u[:,t+1])
```

```
84 # z[:,t+1] = functions.relu(u[:,t+1])
85 # z[:,t+1] = np.tanh(u[:,t+1])
86         y[:,t] = functions.sigmoid(np.dot(z[:,t+1].reshape(1, -1), W_out))誤差
87 
88         #
89         loss = functions.mean_squared_error(dd, y[:,t])
90 
91         delta_out[:,t] = functions.d_mean_squared_error(dd, y[:,t]) * functions.
               d_sigmoid(y[:,t])
92 
93         all_loss += loss
94 
95         out_bin[binary_dim - t - 1] = np.round(y[:,t])
96 
97     for t in range(binary_dim)[::-1]:
98         X = np.array([a_bin[-t-1],b_bin[-t-1]]).reshape(1, -1)
99 
100        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])
101 # 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])
102 # delta[:,t] = (np.dot(delta[:,t+1].T, W.T) + np.dot(delta_out[:,t].T, W_out.T)) *
        d_tanh(u[:,t+1])
103 
104         # 勾配更新
105         W_out_grad += np.dot(z[:,t+1].reshape(-1,1), delta_out[:,t].reshape(-1,1))
106         W_grad += np.dot(z[:,t].reshape(-1,1), delta[:,t].reshape(1,-1))
107         W_in_grad += np.dot(X.T, delta[:,t].reshape(1,-1))
108 
109     # 勾配適用
110     W_in -= learning_rate * W_in_grad
111     W_out -= learning_rate * W_out_grad
112     W -= learning_rate * W_grad
113 
114     W_in_grad *= 0
115     W_out_grad *= 0
116     W_grad *= 0
117 
118     if(i % plot_interval == 0):
119         all_losses.append(all_loss)
120         print("iters:" + str(i))
121         print("Loss:" + str(all_loss))
122         print("Pred:" + str(out_bin))
123         print("True:" + str(d_bin))
124         out_int = 0
125         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.title時系列全体の誤差("")
plt.xlabel("iteration")
plt.ylabel("all_loss")
plt.show()
```

```
iters:0
Loss:1.6047152098915025
Pred:[0 0 0 0 0 0 0 0]
True:[0 1 0 1 0 1 1 0]
57 + 29 = 0
------------
iters:100
Loss:0.8674160764983969
Pred:[0 1 0 0 0 0 1 1]
True:[0 1 1 0 1 1 1 1]
33 + 78 = 67
------------
iters:9800
Loss:0.003595309574136548
Pred:[0 1 0 1 0 1 1 1]
True:[0 1 0 1 0 1 1 1]
18 + 69 = 87
------------
iters:9900
Loss:0.010934892011818569
Pred:[1 0 1 1 0 0 1 0]
True:[1 0 1 1 0 0 1 0]
124 + 54 = 178
------------
```

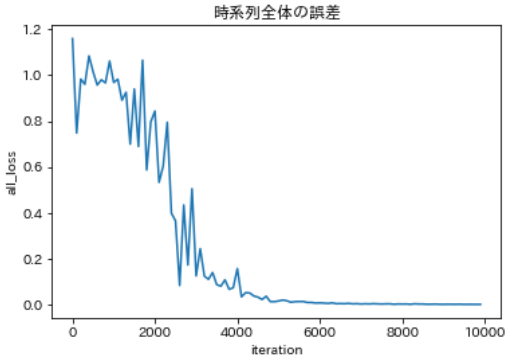
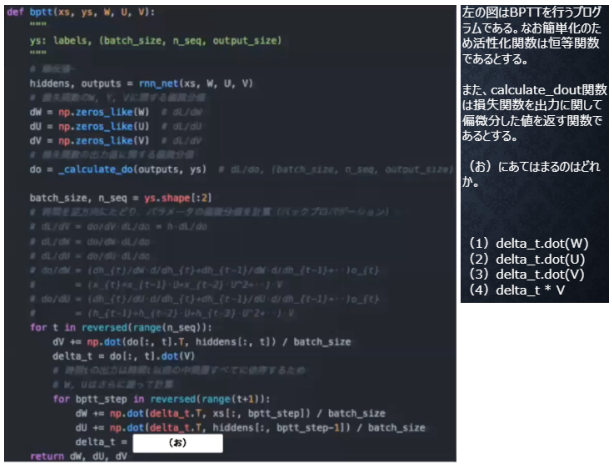


図 1.4　BPTT を用いたバイナリ加算

■演習チャレンジ



■答え　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り、$dh_t/dh_{t-1} = U$ であることに注意すると、過去に遡るたびに U が掛けられる。つまり、delta_t= delta_t.dot(U) の (2) なる。ここで、コーディング上で h_t と表記されているのは、ここまで数式で記述してきた際の中間層出力 z_t に相当する。

第 2 章

ＬＳＴＭ

■RNN の課題　時系列を遡れば遡るほど、勾配が消失していく。つまりこれは、長い時系列の学習が困難であることを意味している。解決策としては、以前学んだ勾配消失問題での解決方法とは別で、ネットワークの構造自体を変えて解決する LSTM（長・短期記憶, Long Short Term Memory）を用いる。

■勾配消失問題　誤差逆伝播法が下位層に進んでいくに連れて、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、訓練は最適値に収束しなくなる。

■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こと。前に学習した Adam optimizer などでは、ハイパーパラメータである学習率の推奨値が研究者などによって与えられている。一般にこの推奨値とは異なる値（例えば推奨値の 1/10 や 10 倍など）を学習率として設定した場合にこの勾配爆発事象が生起する。

■演習チャレンジ　RNN や深いモデルでは勾配の消失または爆発が起こる傾向がある。勾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。具体的には勾配のノルムがしきい値を超えたら、勾配のノルムをしきい値に正規化するというものである。以下は勾配のクリッピングを行う関数である。（さ）にあてはまるのはどれか。

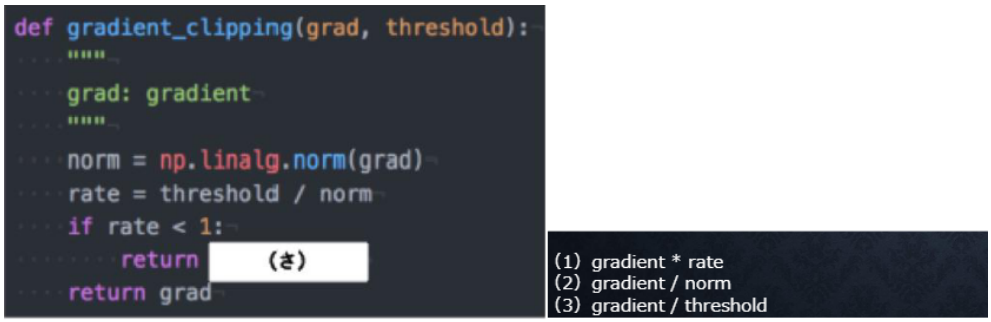


図 2.1　勾配爆発

■答え　勾配のノルムがしきい値より大きいときは、勾配のノルムをしきい値に正規化するので、クリッピングした勾配は、勾配×(しきい値/勾配のノルム) と計算される。つまり、(1) の gradient * rate である。

2.1　LSTM の全体像

LSTM は 1997 年にゼップ・ホッフライター（英語版）とユルゲン・シュミットフーバー（英語版）によって提唱された。

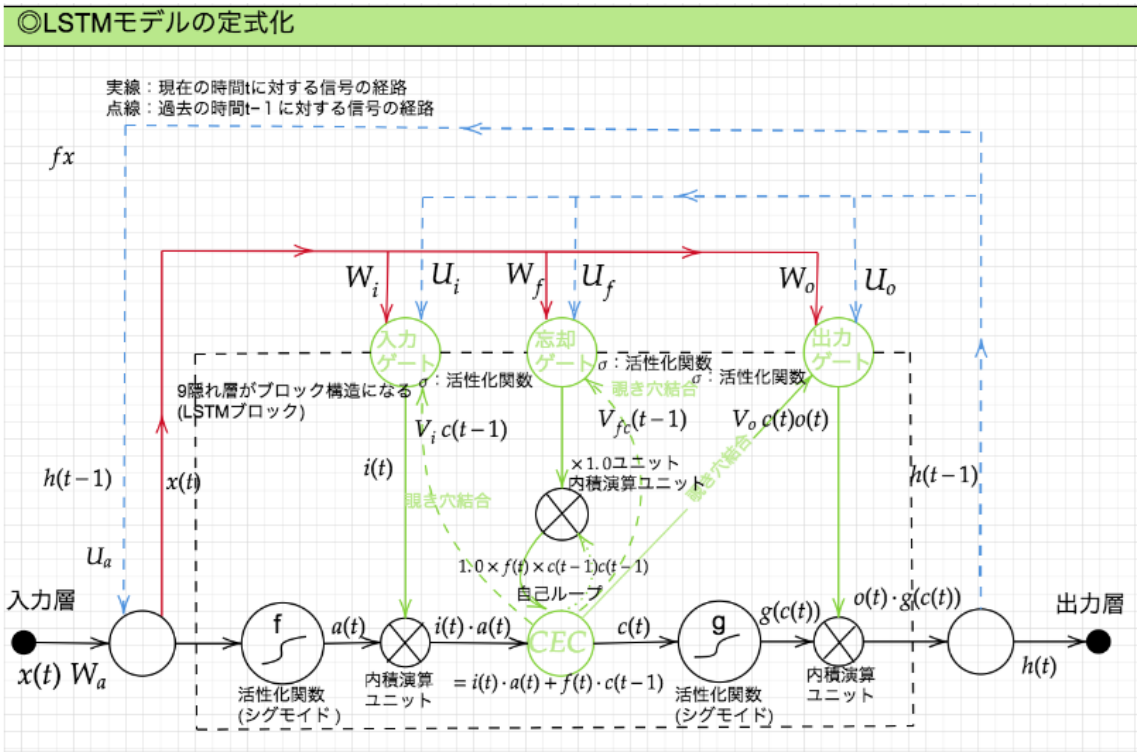


図 2.2　LSTM

2.2　CEC

Constant Error Carousel（定誤差カルーセル、CEC）ユニットの導入によって、LSTM は勾配爆発および消失問題を解決しようとする。CEC は記憶（入力層や中間層の情報）のみに特化したユニット[*1]。勾配消失および勾配爆発の解決方法として、時間を幾ら遡っても勾配が 1 であるようにすれば解決できるだろうという考え出されたもの。

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

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

図 2.3　CEC

[*1] 逆に学習機能が無い。CEC に何を記憶させるか、CEC から何の情報を引っ張り出すかという取り巻きを設定することになる (入力ゲートや出力ゲートのこと)。

■課題　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。このため、ニューラルネットワークの学習特性が無いということになる。入力層→隠れ層への重み入力重み衝突。隠れ層→出力層への重み出力重み衝突。

2.3　入力ゲートと出力ゲート

入力ゲートは、CEC にこんな風に覚えてくださいねと調整するもの。出力ゲートは、CEC にこんな風に出力させると誤差小さくなるね（良い感じの出力が得られる）と調整するもの。入力ゲートでは、 W は今回の入力をどの程度覚えさせるか、U は前回の出力をどの程度反映して覚えさせるかと調整する。

■サマリ　入力・出力ゲートの役割とは何か。これは、入力・出力ゲートを追加することで、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。結果として、CEC の課題を解決することができる。

2.4　忘却ゲート

■LSTM ブロックの課題

1. LSTM の現状 · · · CEC は、過去の情報が全て保管されている。
2. 課題 · · · 過去の情報が要らなくなった場合、削除することはできず、保管され続ける。
3. 解決策 · · · 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。

こうした要求から「忘却ゲート」の概念が誕生した。

■確認テスト　以下の文章を LSTM に入力し空欄に当てはまる単語を予測したいとする。文中の「とても」という言葉は空欄の予測においてなくなっても影響を及ぼさないと考えられる。このような場合、どのゲートが作用すると考えられるか。「映画おもしろかったね。ところで、とてもお腹が空いたから何か ____。」

■答え　忘却ゲート

■演習チャレンジ　以下のプログラムは LSTM の順伝播を行うプログラムである。ただし _sigmoid 関数は要素ごとにシグモイド関数を作用させる関数である。（け）にあてはまるのはどれか。

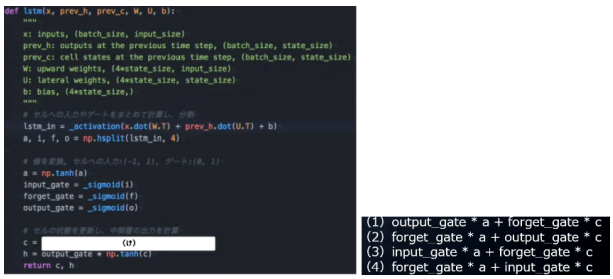


図 2.4　CEC

■答え　新しいセルの状態は、計算されたセルへの入力と 1 ステップ前のセルの状態に入力ゲート、忘却ゲートを掛けて足し合わせたものと表現される。つまり、input_gate* a + forget_gate* c である。

2.5　覗き穴結合

■課題　CEC の保存されている過去の情報を、任意のタイミングで他のノードに伝播させたり、あるいは任意のタイミングで忘却させたい。CEC 自身の値は、ゲート制御に影響を与えていない。

■覗き穴結合とは　CEC 自身の値に、重み行列を介して伝播可能にした構造。一方で、あまり効果は見られない・・・

第 3 章

ＧＲＵ

ゲート付き回帰型ユニット（Gated recurrent unit, GRU）は、回帰型ニューラルネットワーク（RNN）におけるゲート機構である。2014 年に Kyunghyun Cho らによって発表された。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。これを解決するために考え出されたのが「GRU」である。

■GRU とは　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。しかし、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。メリットは計算負荷が低いことにある。

■LSTM との比較　GRU は忘却ゲートを持つ長・短期記憶（long short-term memory、LSTM）に似ているが [3]、出力ゲートを欠くため LSTM よりもパラメータが少ない。多声音楽モデリングおよび音声シグナルモデリングの特定の課題における GRU の性能は、LSTM の性能と類似していることが明らかにされている。GRU は特定のより小さなデータセットではもっと良い性能を示すことが明らかにされている。

しかしながら、Gail Weiss、Yoav Goldberg、および Eran Yahav によって示されているように、LSTM は無制限の計数を容易に実行できるが GRU はできないため、LSTM は GRU よりも「厳密に強力」である。これが、LSTM によって学習可能な単純な言語の学習を GRU が失敗する理由である。

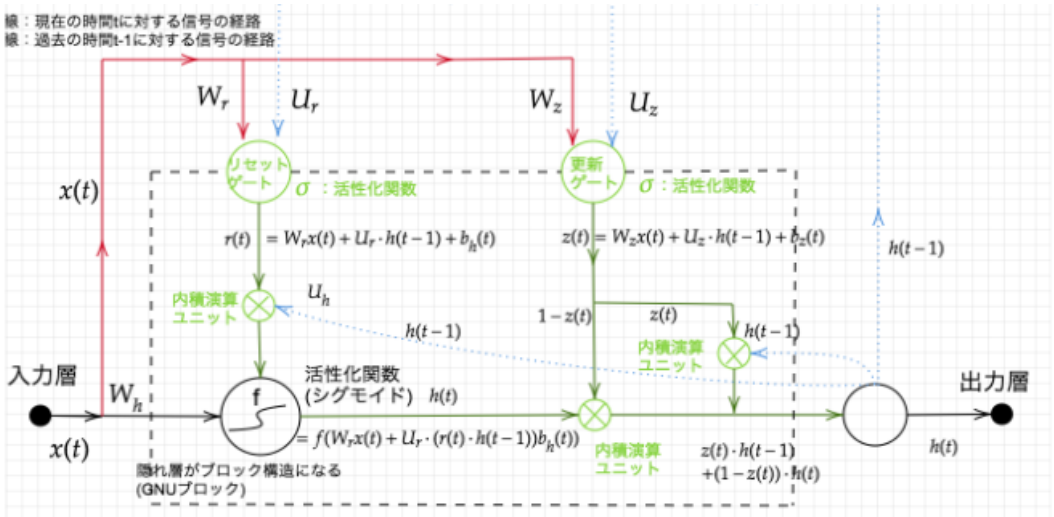


図 3.1　CEC

■確認テスト　LSTM と CEC が抱える課題について、それぞれ簡潔に述べよ。

■答え　LSTM：パラメータ数が多くなって計算負荷が大きくなった。
CEC：学習能力がなくなり、ニューラルネットワークの特性を失った。

■実装演習成果　tensorflow は numpy よりも機械学習に特化したライブラリ。RNN なども 1 行で実行可能となる。

Listing 3.1　predict word for RNN

```
# tensorflow library install
%tensorflow_version 1.x

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 を変更 level
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
```

```
37                text = re.sub(r"[^a-z '\-]", "", text)
38                # 複数のスペースはスペース一文字に変換
39                text = re.sub(r"[ ]+", " ", text)
40
41                # 前処理：'-' で始まる単語は無視する
42                words = [ word for word in text.split() if not word.startswith("-")]
43
44
45        self.data_n = len(words) - self.chunk_size
46        self.data = self.seq_to_matrix(words)
47
48    def prepare_data(self):
49        """訓練データとテストデータを準備する。
50
51        data_n = ( text データの総単語数) - chunk_size
52        input: (data_n, chunk_size, vocabulary_size)
53        output: (data_n, vocabulary_size)
54        """
55
56        # 入力と出力の次元テンソルを準備
57        all_input = np.zeros([self.chunk_size, self.vocabulary_size, self.data_n])
58        all_output = np.zeros([self.vocabulary_size, self.data_n])
59
60        # 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
61        # 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
62        # このときの出力は( i + chunk_size ) 番目の単語
63        for i in range(self.data_n):
64            all_output[:, i] = self.data[:, i + self.chunk_size] # (i + chunk_size) 番
                目の単語のone-hot ベクトル
65            for j in range(self.chunk_size):
66                all_input[j, :, i] = self.data[:, i + self.chunk_size - j - 1]
67
68        # 後に使うデータ形式に合わせるために転置を取る
69        all_input = all_input.transpose([2, 0, 1])
70        all_output = all_output.transpose()
71
72        # 訓練データ：テストデータを4 : 1 に分割する
73        training_num = ( self.data_n * 4 ) // 5
74        return all_input[:training_num], all_output[:training_num], all_input[
              training_num:], all_output[training_num:]
75
76
77    def build_dict(self):
78        # コーパス全体を見て、単語の出現回数をカウントする
79        counter = collections.Counter()
80        for filename in glob.glob(self.corpus_file, recursive=True):
```

```
81              with open(filename, "r", encoding=self.corpus_encoding) as f:
82
83                  # word breaking
84                  text = f.read()
85                  # 全ての文字を小文字に統一し、改行をスペースに変換
86                  text = text.lower().replace("\n", " ")
87                  # 特定の文字以外の文字を空文字に置換する
88                  text = re.sub(r"[^a-z '\-]", "", text)
89                  # 複数のスペースはスペース一文字に変換
90                  text = re.sub(r"[ ]+", " ", text)
91
92                  # 前処理：'-' で始まる単語は無視する
93                  words = [word for word in text.split() if not word.startswith("-")]
94
95                  counter.update(words)
96
97          # 出現頻度の低い単語を一つの記号にまとめる
98          word_id = 0
99          dictionary = {}
100         for word, count in counter.items():
101             if count <= self.unknown_word_threshold:
102                 continue
103
104             dictionary[word] = word_id
105             word_id += 1
106         dictionary[self.unknown_word_symbol] = word_id
107
108         print総単語数：("", len(dictionary))
109
110         # 辞書をpickle を使って保存しておく
111         with open(self.dictionary_filename, "wb") as f:
112             pickle.dump(dictionary, f)
113             print("Dictionary is saved to", self.dictionary_filename)
114
115         self.dictionary = dictionary
116
117         print(self.dictionary)
118
119     def load_dict(self):
120         with open(self.dictionary_filename, "rb") as f:
121             self.dictionary = pickle.load(f)
122             self.vocabulary_size = len(self.dictionary)
123             self.input_layer_size = len(self.dictionary)
124             self.output_layer_size = len(self.dictionary)
125             print総単語数 (": ", self.input_layer_size)
126
```

```
127     def get_word_id(self, word):
128         # print(word)
129         # print(self.dictionary)
130         # print(self.unknown_word_symbol)
131         # print(self.dictionary[self.unknown_word_symbol])
132         # print(self.dictionary.get(word, self.dictionary[self.unknown_word_symbol]))
133         return self.dictionary.get(word, self.dictionary[self.unknown_word_symbol])
134 
135     # 入力された単語をone-hot ベクトルにする
136     def to_one_hot(self, word):
137         index = self.get_word_id(word)
138         data = np.zeros(self.vocabulary_size)
139         data[index] = 1
140         return data
141 
142     def seq_to_matrix(self, seq):
143         print(seq)
144         data = np.array([self.to_one_hot(word) for word in seq]) # (data_n,
                vocabulary_size)
145         return data.transpose() # (vocabulary_size, data_n)
146 
147 class Language:
148     """
149     input layer: self.vocabulary_size
150     hidden layer: rnn_size = 30
151     output layer: self.vocabulary_size
152     """
153 
154     def __init__(self):
155         self.corpus = Corpus()
156         self.dictionary = self.corpus.dictionary
157         self.vocabulary_size = len(self.dictionary) # 単語数
158         self.input_layer_size = self.vocabulary_size # 入力層の数
159         self.hidden_layer_size = 30 # 隠れ層のRNN ユニットの数
160         self.output_layer_size = self.vocabulary_size # 出力層の数
161         self.batch_size = 128 # バッチサイズ
162         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。
163         self.learning_rate = 0.005 # 学習率
164         self.epochs = 1000 # 学習するエポック数
165         self.forget_bias = 1.0 # LSTM における忘却ゲートのバイアス
166         self.model_filename = "./data_for_predict/predict_model.ckpt"
167         self.unknown_word_symbol = self.corpus.unknown_word_symbol
168 
169     def inference(self, input_data, initial_state):
```

```
170         """
171         :param input_data: (batch_size, chunk_size, vocabulary_size) 次元のテンソル
172         :param initial_state: (batch_size, hidden_layer_size) 次元の行列
173         :return:
174         """
175         # 重みとバイアスの初期化
176         hidden_w = tf.Variable(tf.truncated_normal([self.input_layer_size, self.
                hidden_layer_size], stddev=0.01))
177         hidden_b = tf.Variable(tf.ones([self.hidden_layer_size]))
178         output_w = tf.Variable(tf.truncated_normal([self.hidden_layer_size, self.
                output_layer_size], stddev=0.01))
179         output_b = tf.Variable(tf.ones([self.output_layer_size]))
180 
181         # 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。
182         # 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
183         # 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。
184 
185         input_data = tf.transpose(input_data, [1, 0, 2]) # 転置。
                (chunk_size, batch_size, vocabulary_size)
186         input_data = tf.reshape(input_data, [-1, self.input_layer_size]) # 変形。
                (chunk_size * batch_size, input_layer_size)
187         input_data = tf.matmul(input_data, hidden_w) + hidden_b # 重みとバイアスを適用。
                WB (chunk_size, batch_size, hidden_layer_size)
188         input_data = tf.split(input_data, self.chunk_size, 0) # リストに分割。
                chunk_size * (batch_size, hidden_layer_size)
189 
190         # 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。
191         cell = tf.nn.rnn_cell.BasicRNNCell(self.hidden_layer_size)
192         outputs, states = tf.nn.static_rnn(cell, input_data, initial_state=initial_state
                )
193 
194         # 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
195         # 最終的にsoftmax 関数で処理し、確率として解釈される。
196         # softmax 関数はこの関数の外で定義する。
197         output = tf.matmul(outputs[-1], output_w) + output_b
198 
199         return output
200 
201     def loss(self, logits, labels):
202         cost = tf.reduce_mean(tf.nn.softmax_cross_entropy_with_logits(logits=logits,
                labels=labels))
203 
204         return cost
205 
206     def training(self, cost):
207         # 今回は最適化手法としてAdam を選択する。
```

```
208         # 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
209         optimizer = tf.train.AdamOptimizer(learning_rate=self.learning_rate).minimize(
                cost)
210
211         return optimizer
212
213     def train(self):
214         # 変数などの用意
215         input_data = tf.placeholder("float", [None, self.chunk_size, self.
                input_layer_size])
216         actual_labels = tf.placeholder("float", [None, self.output_layer_size])
217         initial_state = tf.placeholder("float", [None, self.hidden_layer_size])
218
219         prediction = self.inference(input_data, initial_state)
220         cost = self.loss(prediction, actual_labels)
221         optimizer = self.training(cost)
222         correct = tf.equal(tf.argmax(prediction, 1), tf.argmax(actual_labels, 1))
223         accuracy = tf.reduce_mean(tf.cast(correct, tf.float32))
224
225         # 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
226         tf.summary.scalar("Cross entropy: ", cost)
227         summary = tf.summary.merge_all()
228
229         # 訓練・テストデータの用意
230         # corpus = Corpus()
231         trX, trY, teX, teY = self.corpus.prepare_data()
232         training_num = trX.shape[0]
233
234         # ログを保存するためのディレクトリ
235         timestamp = time.time()
236         dirname = datetime.datetime.fromtimestamp(timestamp).strftime("%Y%m%d%H%M%S")
237
238         # ここから実際に学習を走らせる
239         with tf.Session() as sess:
240             sess.run(tf.global_variables_initializer())
241             summary_writer = tf.summary.FileWriter("./log/" + dirname, sess.graph)
242
243             # エポックを回す
244             for epoch in range(self.epochs):
245                 step = 0
246                 epoch_loss = 0
247                 epoch_acc = 0
248
249                 # 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)
250                 # 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
```

```
251                 while (step + 1) * self.batch_size < training_num:
252                     start_idx = step * self.batch_size
253                     end_idx = (step + 1) * self.batch_size
254
255                     batch_xs = trX[start_idx:end_idx, :, :]
256                     batch_ys = trY[start_idx:end_idx, :]
257
258                     _, c, a = sess.run([optimizer, cost, accuracy],
259                                        feed_dict={input_data: batch_xs,
260                                                   actual_labels: batch_ys,
261                                                   initial_state: np.zeros([self.
                                                       batch_size, self.hidden_layer_size
                                                       ])
262                                                   }
263                                        )
264                     epoch_loss += c
265                     epoch_acc += a
266                     step += 1
267
268                 # コンソールに損失関数の値や精度を出力しておく
269                 print("Epoch", epoch, "completed ouf of", self.epochs, "-- loss:",
                        epoch_loss, " -- accuracy:",
270                       epoch_acc / step)
271
272                 # が終わるごとに用に値を保存 EpochTensorBoard
273                 summary_str = sess.run(summary, feed_dict={input_data: trX,
274                                                            actual_labels: trY,
275                                                            initial_state: np.zeros(
276                                                                [trX.shape[0],
277                                                                 self.hidden_layer_size]
278                                                            )
279                                                            }
280                                        )
281                 summary_writer.add_summary(summary_str, epoch)
282                 summary_writer.flush()
283
284             # 学習したモデルも保存しておく
285             saver = tf.train.Saver()
286             saver.save(sess, self.model_filename)
287
288             # 最後にテストデータでの精度を計算して表示する
289             a = sess.run(accuracy, feed_dict={input_data: teX, actual_labels: teY,
290                                               initial_state: np.zeros([teX.shape[0],
                                                    self.hidden_layer_size])})
291             print("Accuracy on test:", a)
292
```

```
293
294     def predict(self, seq):
295         """文章を入力したときに次に来る単語を予測する
296
297         :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。
298         :return:
299         """
300
301         # 最初に復元したい変数をすべて定義してしまいます
302         tf.reset_default_graph()
303         input_data = tf.placeholder("float", [None, self.chunk_size, self.
                input_layer_size])
304         initial_state = tf.placeholder("float", [None, self.hidden_layer_size])
305         prediction = tf.nn.softmax(self.inference(input_data, initial_state))
306         predicted_labels = tf.argmax(prediction, 1)
307
308         # 入力データの作成
309         # seq をone-hot 表現に変換する。
310         words = [word for word in seq.split() if not word.startswith("-")]
311         x = np.zeros([1, self.chunk_size, self.input_layer_size])
312         for i in range(self.chunk_size):
313             word = seq[len(words) - self.chunk_size + i]
314             index = self.dictionary.get(word, self.dictionary[self.unknown_word_symbol])
315             x[0][i][index] = 1
316         feed_dict = {
317             input_data: x, # (1, chunk_size, vocabulary_size)
318             initial_state: np.zeros([1, self.hidden_layer_size])
319         }
320
321         # tf.Session()を用意
322         with tf.Session() as sess:
323             # 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。
324             saver = tf.train.Saver()
325             saver.restore(sess, self.model_filename)
326
327             # ロードしたモデルを使って予測結果を計算
328             u, v = sess.run([prediction, predicted_labels], feed_dict=feed_dict)
329
330             keys = list(self.dictionary.keys())
331
332
333             # コンソールに文字ごとの確率を表示
334             for i in range(self.vocabulary_size):
335                 c = self.unknown_word_symbol if i == (self.vocabulary_size - 1) else
                        keys[i]
336                 print(c, ":", u[0][i])
```

```
337
338             print("Prediction:", seq + " " + ("<???>" if v[0] == (self.vocabulary_size
                    - 1) else keys[v[0]]))
339
340         return u[0]
341
342
343 def build_dict():
344     cp = Corpus()
345     cp.build_dict()
346
347 if __name__ == "__main__":
348     #build_dict()
349
350     ln = Language()
351
352     # 学習するときに呼び出す
353     #ln.train()
354
355     # 保存したモデルを使って単語の予測をする
356     ln.predict("some of them looks like")
```

```
countrys : 1.3141076e-14
cemetery : 1.3612609e-14
arched : 1.4054285e-14
faithful : 1.3997865e-14
canton : 1.4607429e-14
pavilion : 1.4429179e-14
ashore : 1.39286725e-14
ornate : 1.5933007e-14
saint : 1.5465495e-14
barrier : 1.28282445e-14
fitted : 1.3812267e-14
dice : 1.7615722e-14
taipa : 1.4471156e-14
brazil : 1.288054e-14
elbow : 1.433364e-14
beautifully : 1.5469302e-14
lush : 1.5432405e-14
fertile : 1.4179788e-14
guangzhous : 1.4806676e-14
heroic : 1.2670136e-14
tile : 1.3839799e-14
rode : 1.5756664e-14
ci : 1.6474115e-14
carved : 1.4261538e-14
✓ 2分 28 秒　完了時間: 21:58
```

図 3.2　RNN を用いた自然言語処理

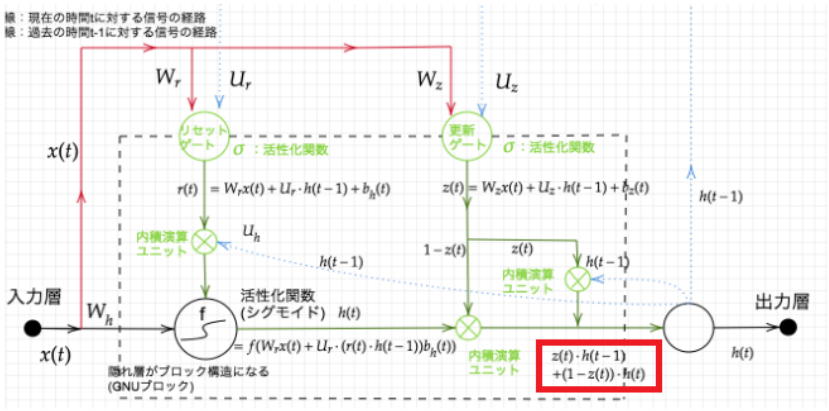
■演習チャレンジ　GRU(Gated Recurrent Unit) も LSTM と同様に RNN の一種であり、単純な RNN において問題となる勾配消失問題解決し、長期的な依存関係を学習することができる。LSTM に比べ変数の数やゲートの数が少なく、より単純なモデルであるが、タスクによっては LSTM より良い性能を発揮する。以下のプログラムは GRU の順伝播を行うプログラムである。ただし _sigmoid 関数は要素ごとにシグモイド関数を作用させる関数である。（こ）にあてはまるのはどれか。

```
def gru(x, h, W_r, U_r, W_z, U_z, W, U):
    """
    x: inputs, (batch_size, input_size)
    h: outputs at the previous time step, (batch_size, state_size)
    W_r, U_r: weights for reset gate
    W_z, U_z: weights for update gate
    U, W: weights for new state
    """
    # ゲートを計算
    r = _sigmoid(x.dot(W_r.T) + h.dot(U_r.T))
    z = _sigmoid(x.dot(W_z.T) + h.dot(U_z.T))

    # 次状態を計算
    h_bar = np.tanh(x.dot(W.T) + (r * h).dot(U.T))
    h_new = (こ)
    return h_new
```

(1) z * h_bar
(2) (1-z) * h_bar
(3) z * h * h_bar
(4) (1-z) * h + z * h_bar

■答え　新しい中間状態は、1 ステップ前の中間表現と計算された中間表現の線形和で表現される。つまり更新ゲート z を用いて、(1-z) * h + z * h_bar と書ける。



■確認テスト　LSTM と GRU の違いを簡潔に述べよ。

■答え　どちらも RNN の改良版という位置づけのモデルであるが、LSTM は、CEC を基幹とする入力ゲート、出力ゲート、忘却ゲートという記憶と学習機能を完全に分離して構成したモデルである。一方、GRU は LSTM のデメリットであったパラメータ数の多さ、計算負荷量の増大に対応するため、CEC を機能をなくして、リセットゲートと更新ゲートの機能に絞り、パラメータ数を減らして計算速度をあげたもの（タスクによっては、簡素化した影響で LSTM より精度は劣る）。

第 4 章

双方向ＲＮＮ

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。実用例としては、文章の推敲や、機械翻訳等があげられる。

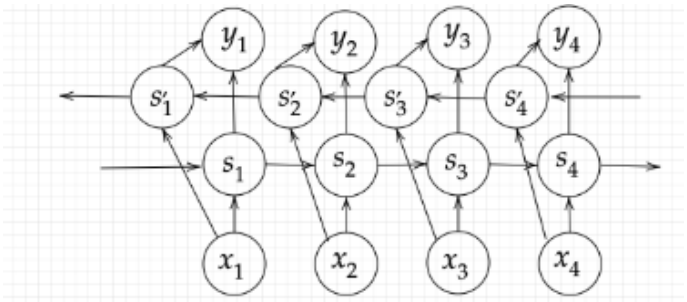
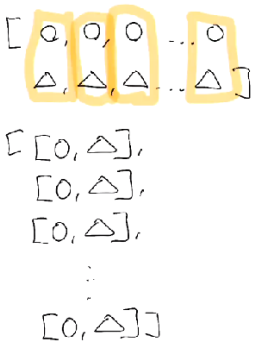


図 4.1　双方向 RNN

■演習チャレンジ　以下は双方向 RNN の順伝播を行うプログラムである。順方向については、入力から中間層への重み W_f, 一ステップ前の中間層出力から中間層への重みを U_f、逆方向に関しては同様にパラメータ W_b, U_b を持ち、両者の中間層表現を合わせた特徴から出力層への重みは V である。_rnn 関数は RNN の順伝播を表し中間層の系列を返す関数であるとする。(か) にあてはまるのはどれか

```
def bidirectional_rnn_net(xs, W_f, U_f, W_b, U_b, V):
    """
    W_f, U_f: forward rnn weights, (hidden_size, input_size)
    W_b, U_b: backward rnn weights, (hidden_size, input_size)
    V: output weights, (output_size, 2*hidden_size)
    """
    xs_f = np.zeros_like(xs)
    xs_b = np.zeros_like(xs)
    for i, x in enumerate(xs):
        xs_f[i] = x
        xs_b[i] = x[::-1]
    hs_f = _rnn(xs_f, W_f, U_f)
    hs_b = _rnn(xs_b, W_b, U_b)
    hs = [                    (か)                    for h_f, h_b in zip(hs_f, hs_b)]
    ys = hs.dot(V.T)
    return ys
```

(1) h_f + h_b[::-1]
(2) h_f * h_b[::-1]
(3) np.concatenate([h_f, h_b[::-1]], axis=0)
(4) np.concatenate([h_f, h_b[::-1]], axis=1)



■答え　双方向 RNN では、順方向と逆方向に伝播したときの中間層表現（同時刻）をあわせたものが特徴量となるので、np.concatenate([h_f, h_b[::-1]], axis=1) である。

第 5 章

Ｓｅｑ２Ｓｅｑ

Seq2Seq(sequence to sequence) は、以下で説明する Encoder と Decoder を備えた Encoder-Decoder モデルを使って、系列データを別の系列データに変換するモデルのことを指す[*1]。この機能を用いることにより、seq2seq で機械翻訳をしたり、対話モデルを作ったりすることが可能になる。

構成としては、2 つのニューラルネットワークがドッキングしたものである。1 つのニューラルネットワーク（エンコーダ）に（RNN、LSTM、GRU など）単語の列が順々に入力、そうすることで単語の情報（記憶）に関する文脈が保存され、隠れ層には、その文脈の意味が（ベクトルとして）保存されることになる。もう一方のニューラルネットワーク（デコーダ）は、入力として文の意味（文脈）を受け取り、出力として解釈を生成する。

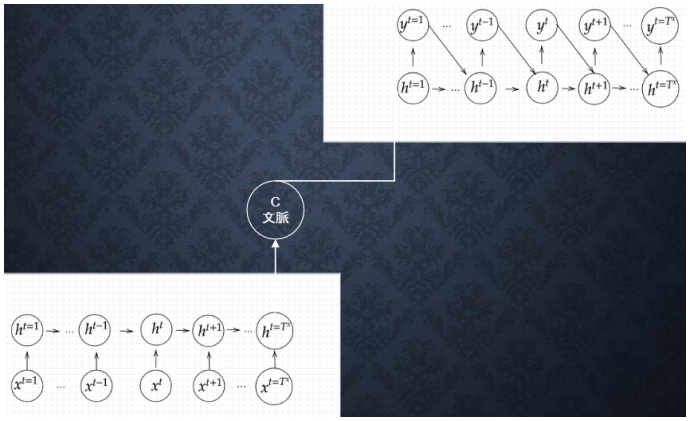
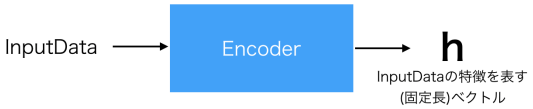


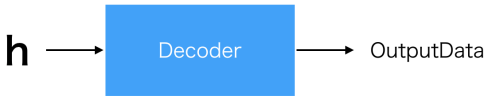
図 5.1　SEQ2SEQ

■Encorder　InputData(画像、テキスト、音声、動画 etc) を何かしらの (固定長) 特徴ベクトルに変換する機構のことをいう。下図は、InputData を抽象的なベクトルにエンコードしてるイメージ



[*1] https://qiita.com/m__k/items/b18756628575b177b545

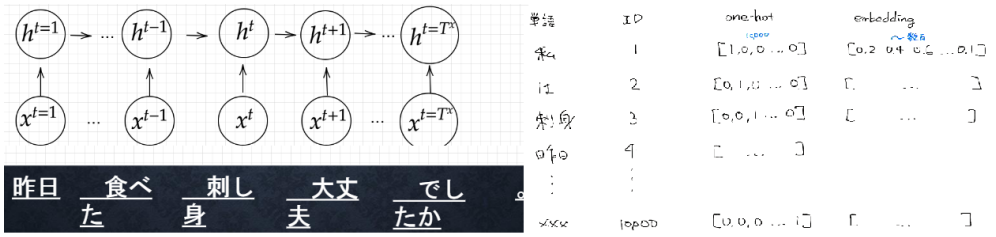
■Decorder　Encoder でエンコードされた特徴ベクトルをデコードして何か新しいデータを生む機構のことをいう。OutputData は、InputData と同じデータ形式である必要はなく、画像、テキスト、音声 etc･･･。



5.1　Encoder RNN

下図は、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。エンコーダでは、文章を受け取って、文脈を意味ベクトルとして生成することになる。

自然言語をいかにして数字（ベクトル）で表現できるのであろうか？ 1 つ 1 つの言葉をまずナンバリング（ID）する。そうすることで、文章を one–hot–vector で表現できるようになる。一方でこのままでは、ゼロ成分が多く無駄の多いベクトル表現であり、取り扱いが難であるため、実際の処理では、機械学習（深層学習）により、embedding 表現 (数百の成分) に変換する。※ embedding 表現で成分が近いということは、単語の意味が近いことを意味する。



1. Taking :文章を単語等のトークン毎に分割し、トークンごとの ID に分割する。
2. Embedding :ID から、そのトークンを表す分散表現ベクトルに変換。
3. Encoder RNN:ベクトルを順番に RNN に入力していく。

■Encorder RNN の処理手順

1. vec1 を RNN に入力し、hidden state を出力。この hiddenstate と次の入力 vec2 をまた RNN に入力してきた hidden state を出力という流れを繰り返す
2. 最後の vec を入れたときの hiddenstate を finalstate としてとっておく。この finalstate が thoughtvector と呼ばれ、入力した文の意味を表すベクトルとなる。

■最近の進展　Google が開発した BERT と呼ばれる自然言語処理がある[*2]。BERT（Bidirectional Encoder Representations from Transformers, Transformer) は、「双方向のエンコード表現」と訳され、2018 年 10 月に Google の Jacob Devlin らの論文で発表された自然言語処理モデル。翻訳、文書分類、質問応答など自然言語処理の仕事の分野のことを「(自然言語処理）タスク」と言いますが、BERT は、多様なタスクにおいて当

[*2] LINE Clova、Amazon Echo などのスマートスピーカーやウェブ上のカスタマーサービスに見られるチャットボットはどれも自然言語処理という AI 技術が用いられています。この自然言語処理においては、2018 年 10 月に Google が BERT という手法を発表し、「AI が人間を超えた」と言わしめるほどのブレイクスルーをもたらしました。

時の最高スコアを叩き出しました。これは、MLM(Masked Languiage Model) というものが使われている。

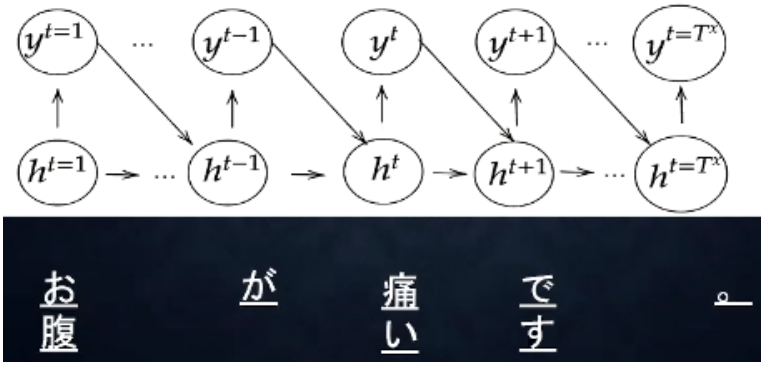
従来の自然言語処理モデルでは、文章を単一方向からでしか処理できませんでした。そのため、目的の単語の前の文章データから予測する必要がありました。しかし、先述の通り BERT は双方向の Transformer によって学習するため、従来の手法に比べ精度が向上しました。それを実現しているのが Masked Language Model です。以下で説明します。

■MLM の具体的な処理　入力文の 15% の単語を確率的に別の単語で置き換えし、文脈から置き換える前の単語を予測させます。具体的には、選択された 15% のうち、80% は [MASK] に置き換えるマスク変換、10% をランダムな別の単語に変換、残りの 10% はそのままの単語にする。

このように置換された単語を周りの文脈から当てるタスクを解くことで、単語に対応する文脈情報を学習する[*3]。

5.2　Decorder RNN

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。ベクトル表現から ID に変換して言葉に戻していく。



[*3] https://ledge.ai/bert/

■Decorder RNN の処理手順

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。

■確認テスト　下記の選択肢から、seq2seq について説明しているものを選べ。

1. 時刻に関して順方向と逆方向の RNN を構成し、それら 2 つの中間層表現を特徴量として利用するものである。
2. RNN を用いた Encoder-Decoder モデルの一種であり、機械翻訳などのモデルに使われる。
3. 構文木などの木構造に対して、隣接単語から表現ベクトル（フレーズ）を作るという演算を再帰的に行い（重みは共通）、文全体の表現ベクトルを得るニューラルネットワークである。
4. 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 CEC とゲートの概念を導入することで解決したものである。

■答え

1. (×)　双方向ＲＮＮ
2. (○)　SEQ2SEQ
3. (×)　構文木
4. (×)　LSTM

■演習チャレンジ　機械翻訳タスクにおいて、入力は複数の単語から成る文（文章）であり、それぞれの単語は one-hot ベクトルで表現されている。Encoder において、それらの単語は単語埋め込みにより特徴量に変換され、そこから RNN によって（一般には LSTM を使うことが多い）時系列の情報をもつ特徴へとエンコードされる。以下は、入力である文（文章）を時系列の情報をもつ特徴量へとエンコードする関数である。ただし _activation 関数はなんらかの活性化関数を表すとする。（き）にあてはまるのはどれか。

```
def encode(words, E, W, U, b):
    """
    words: sequence words (sentence), one-hot vector, (n_words, vocab_size)
    E: word embeding matrix, (embed_size, vocab_size)
    W: upward weights, (hidden_size, hidden_size)
    U: lateral weights, (hidden_size, embed_size)
    b: bias, (hidden_size,)
    """
    hidden_size = W.shape[0]
    h = np.zeros(hidden_size)
    for w in words:
        e = （き）
        h = _activation(W.dot(e) + U.dot(h) + b)
    return h
```

(1) E.dot(w)
(2) E.T.dot(w)
(3) w.dot(E.T)
(4) E * w

■答え　単語 w は one-hot ベクトルであり、それを単語埋め込みにより別の特徴量に変換する。これは埋め込み行列 E を用いて、(1) E.dot(w) と書ける。

5.3　HRED

■Seq2Seq の課題　一問一答しかできない。問に対して文脈も何もなく、ただ応答が行われる続ける。ということになる。

■HRED とは　過去 n-1 個の発話から次の発話を生成する。システム:インコかわいいよね。ユーザー:うんシステム:インコかわいいのわかる。Seq2seq では、会話の文脈無視で、応答がなされたが、HRED では、前の単語の流れに即して応答されるため、より人間らしい文章が生成される。

■HRED の構造　Seq2Seq+ Context RNN ここで、context RNN: Encoder とは、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キスト全体を表すベクトルに変換する構造。これにより、過去の発話の履歴を加味した返答をできる。

■Seq2Seq の課題

1. 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。
   → 同じコンテキスト（発話リスト）を与えられても、答えの内容が毎回会話の流れとしては同じものしか出せない。
2. HRED は短く情報量に乏しい答えをしがちである。
   → 短いよくある答えを学ぶ傾向がある。ex)「うん」「そうだね」「・・・」など。

5.4　VHRED

VHRED は、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である。

■確認テスト　(1)seq2seq と HRED、(2)HRED と VHRED の違いを簡潔に述べよ。

■答え

1. seq2seq は、1 文の 1 問 1 答に対して処理ができる。（ある時系列データから別の時系列データを作る）
   HRED は、seq2seq の機構にこれまでの文脈ベクトルの文脈の意味を汲み取った encorder-decorder（文の変換）ができるようにしたもの
2. HRED が当たり障りのない回答したできなくなったことに対して、課題を解決するために改良されたモデル

5.5 VAE

5.5.1 オートエンコーダー

オートエンコーダは、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。オートエンコーダの具体例である MNIST の場合、28x28 の数字の画像を入れて、同じ画像を出力するニューラルネットワークということになる。

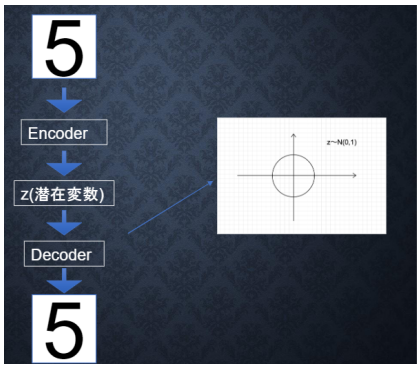
■オートエンコーダー構造　オートエンコーダ構造説明入力データから潜在変数 z に変換するニューラルネットワークを Encoder 逆に潜在変数 z をインプットとして元画像を復元するニューラルネットワークを Decoder。

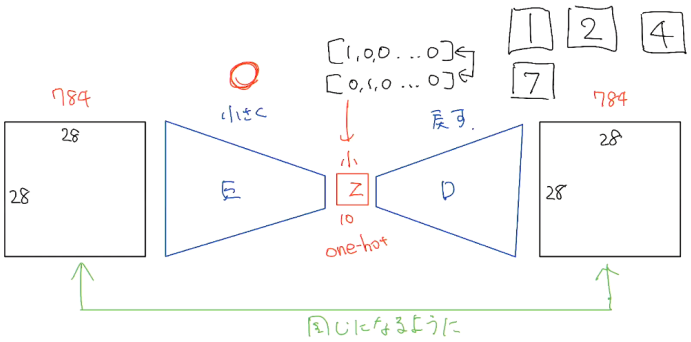
■メリット　次元削減が行えること。※ z の次元が入力データより小さい場合、次元削減とみなすことができる。

5.5.2 VAE

通常のオートエンコーダーの場合、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 はこの潜在変数 z に確率分布 $z \sim N(0,1)$ を仮定したもの[*4]。

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。





■確認テスト　VAE に関する下記の説明文中の空欄に当てはまる言葉を答えよ。自己符号化器の潜在変数に（）を導入したもの。

■答え　確率変数

[*4] L2 正則化のような一種の正規化に相当

第 6 章

Ｗｏｒｄ２ｖｅｃ

■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。つまり、固定長形式で単語を表す必要がある。

■解決策　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。

■word2vec とは　単語を固定長のベクトルで表現することを「単語の分散表現」と呼ぶます。単語をベクトルで表現することができれば単語の意味を定量的に把握することができるため、様々な処理に応用することができる。Word2Vec も単語の分散表現の獲得を目指した手法。

■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。ボキャブラリ×任意の単語ベクトル次元で重み行列が誕生する。

第 7 章

Attention Mechanism

■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。「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。

■解決策　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、仕組みが必要になる。

■Attention Mechanism とは　「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。

■確認テスト　RNN と word2vec、seq2seq と Attention の違いを簡潔に述べよ。

■答え

1. RNN･･･ 時系列データを処理するのに適したニューラルネットワーク
2. word2vec･･･ 単語の分散表現ベクトルを得る手法
3. seq2seq･･･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
4. Attention･･･ 時系列データの中身に対して関連性に応じて重みをつける手法

参考文献

[1] 奥村晴彦，黒木裕介『LaTeX 2_ε 美文書作成入門 第 7 版』（技術評論社，2017）

[2] 加藤公一，『機械学習のエッセンス』（SB クリエイティブ，2018）

[3] 大重美幸，『Python 3 入門ノート』（ソーテック，2018）

[4] 大関真之，『機械学習入門』（オーム，2018）

[5] Andreas C. Müller et al.,『Python ではじめる機械学習』（オライリージャパン，2018）

[6] 加藤公一（監修），『機械学習図鑑』（翔泳社，2019）

[7] 岡谷貴之，『深層学習』（講談社，2015）

[8] 竹内一郎，『サポートベクトルマシン』、講談社，2015）

[9] 斎藤康毅，『ゼロから作る Deep Learning』（オライリージャパン，2019）

[10] Guido van Rossum，『Python チュートリアル（第 4 版)』（オライリージャパン，2021）

[11] 中田亨，『多様性工学』（日科技連，2021）