復習

<確認テスト 1>

Q. サイズ 5×5 の入力画像を、サイズ 3×3 のフィルタで畳み込んだ時の出力画像のサイズを答えよ。

なお、ストライドは 2、パディングは 1 とする。

A. 3×3

下記の計算式より、(5+(2×1)-3)/2 + 1 = 3 となるため、3×3 となる。

$$OH = \frac{H + 2P - FH}{S} + 1$$

$$OW = \frac{W + 2P - FW}{S} + 1$$

Section1：再帰的ニューラルネットワークの概念

RNN

時系列データに対応可能なニューラルネットワーク。

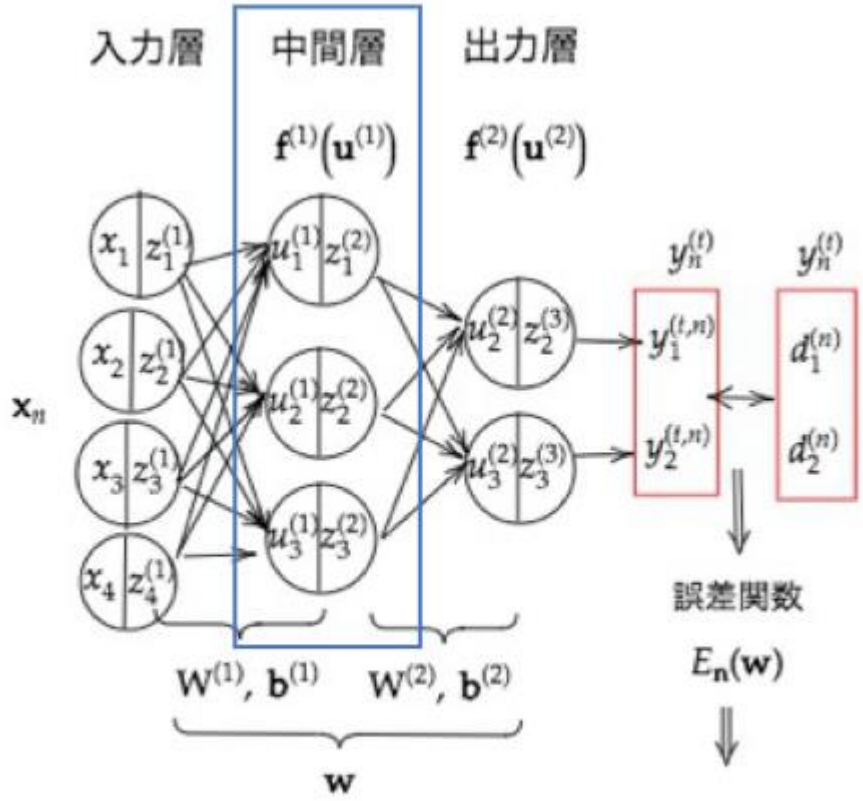
時系列データとは、時間的順序を追って一定間隔ごとに観察され、しかも相互に統計的依存関係が認められるようなデータ系列のこと。

時系列データの具体的な例

音声データ

テキストデータ

構造

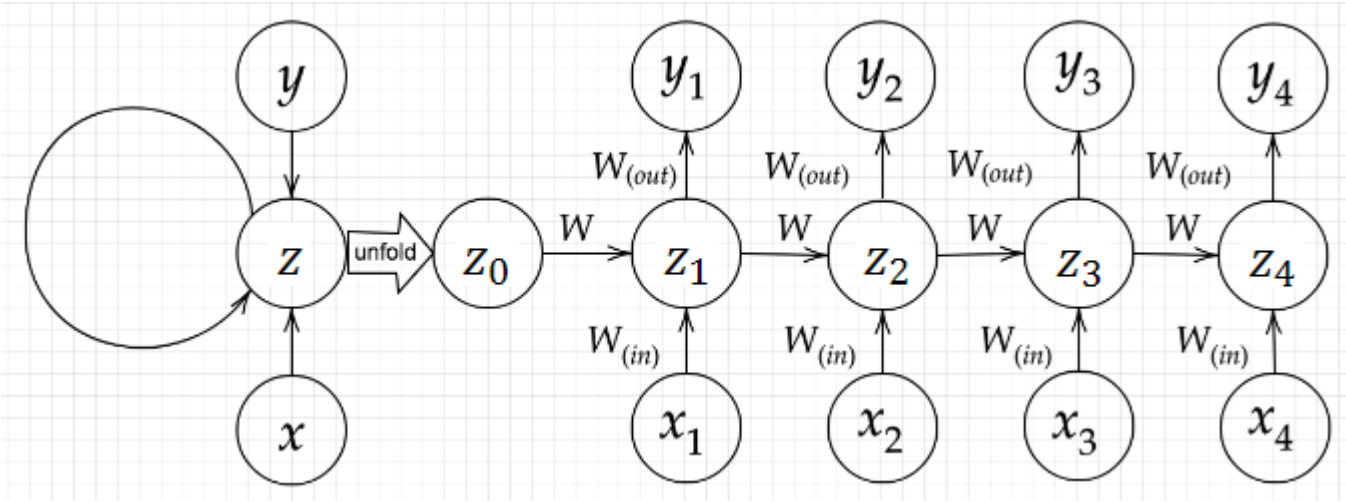


$$\mathbf{w}^{(t+1)} = \mathbf{w}^{(t)} - \varepsilon \nabla \mathbf{E}_n(\mathbf{w}) \Leftarrow \quad \nabla E_n(\mathbf{w}) = \frac{\partial E}{\partial \mathbf{w}}$$

前の中間層からの重み W が存在しているニューラルネットワーク。

現時点の $W_{(in)}$ と前の中間層からの W の 2 つが存在している。

W は前の中間層から受け取るが、前の中間層時点での W は前の前の中間層から受け取っているため、今までの中間層の影響が反映された値になっている。



数式

$u^t = W_{(in)}x^t + Wz^{t-1} + b$

$z^t = f(W_{(in)}x^t + Wz^{t-1} + b)$

$v^t = W_{(out)}z^t + c$

$y^t = g(W_{(out)}z^t + c)$

Python コード

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

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 t での状態を再帰的に求める再帰構造が必要となる。

BPTT

RNN におけるパラメータ調整方法の一種であり、誤差逆伝播の一種である。

数式 1

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

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

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

$$\frac{\partial E}{\partial b} = \frac{\partial E}{\partial u^t}\frac{\partial u^t}{\partial b} = \delta^t$$

$$\frac{\partial E}{\partial c} = \frac{\partial E}{\partial v^t}\frac{\partial v^t}{\partial b} = \delta^{out,t}$$

Python コード 1

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

コード記載は無し(simple RNN コードでは簡略化のため、$\frac{\partial E}{\partial b}$は省略。

コード記載は無し(simple RNN コードでは簡略化のため、$\frac{\partial E}{\partial c}$は省略。

数式 2

$$u^t = W_{(in)}x^t + Wz^{t-1} + b$$

$$z^t = f(W_{(in)}x^t + Wz^{t-1} + b)$$

$$v^t = W_{(out)}z^t + c$$

$$y^t = g(W_{(out)}z^t + c)$$

Python コード 2

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

数式 3

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

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

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

Python コード 3

```
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])
```

コード記載は無し。

コード記載は無し。

数式 4

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

$$W_{(out)}^{t+1} = W_{(out)}^t - \varepsilon\frac{\partial E}{\partial W_{(out)}} = W_{(out)}^t - \varepsilon\delta^{out,t}[z^t]^T$$

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

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

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

Python コード 4

W_in = learning_rate * W_in_grad

W_out = learning_rate * W_out_grad

W -= learning_rate * W_grad

コード記載は無し。

コード記載は無し。

全体像

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

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

<確認テスト 1>

Q. RNN のネットワークには大きく分けて 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。

残りの 1 つの重みについて説明せよ。

A. 前の中間層からの重み。

<確認テスト 2>

Q. 連鎖律の原理を使い、$\frac{dz}{dx}$を求めよ。

$z = t^2$

$t = x + y$

A. $2(x + y)$

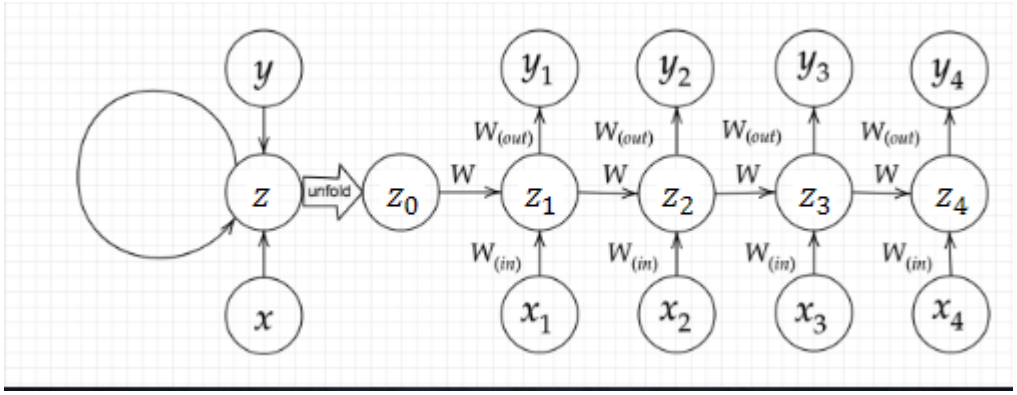
$\frac{dz}{dt} = 2t$、$\frac{dt}{dx} = 1$、$\frac{dz}{dx} = \frac{dz}{dt}\frac{dt}{dx}$より、$\frac{dz}{dx} = 2t \times 1 = 2t = 2(x + y)$

<確認テスト 3>

Q. 下図のy_1を$x \cdot z_0 \cdot z_1 \cdot w_{in} \cdot w \cdot w_{out}$を用いて数式で表せ。

※バイアスは任意の文字で定義せよ。

※また中間層の出力にシグモイド関数$g(x)$を作用させよ。



A. $y_1 = \mathrm{g}(w_{out}z_1 + \mathrm{c})$、$z_1 = f(w_{in}x_1 + b + wz_0)$

<練習問題 1>

Q. 以下は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。
(く)にあてはまるのはどれか。

```
def traverse(node):
  “””
  node: tree node, recursive dict, {left: node’, right: node’’}
          if leaf, word embeded vector, (emved_size,)

  w: weights, global variable, (embed_size, 2*embed_size)
  b: bias, global variable, (embed_size,)
  “””
  if not isinstance(node, dict):
     v = node
  else:
     left = traverse(node[‘left’])
     right = traverse(node[‘right’])
     v = _activation((く))
  return v
```

(1) W.dot(left + right)
(2) W.dot(np.concatenate([left, right])
(3) W.dot(left * right)
(4) W.dot(np.maximum(left, right))

A. (2) W.dot(np.concatenate([left, right])
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

現するため、W.dot(np.concatenate([left, right]))が正解となる。

<練習問題 2>

Q. 以下は BPTT を行うプログラムである。なお簡単化のため活性化関数は恒等関数であるとする。
また、calculate_do 関数は損失関数を出力に関して偏微分した値を返す関数であるとする。
(お)にあてはまるのはどれか。

```
def bptt(xs, ys, W, U, V):
  """
  ys: labels, (batch_size, n_seq, utput_size)
  """
  # 順伝播
  Hiddens, outputs = rnn_net(xs, W, U, V)
  # 損失関数の W, Y, V に関する偏微分値
  dW = np.zeros_like(W)     # dL/dW
  dU = np.zeros_like(U)     # dL/dU
  dV = np.zeros_like(V)     # dL/dV
  # 損失関数の出力値に関する偏微分値
  do = _calculate_do(outputs, ys)     # dL/do, (batch_size, n_seq, output_size)

  batch_size, n_seq = ys.shape[:2]
  # 時間を逆方向にたどり、パラメータの偏微分値を計算(バックプロパゲーション)
  # dL/dV = do/dV・dL/do = h・dL/do
  # dL/dW = do/dW・dL/do
  # dL/dU = do/dU・dL/do
  # do/dW = (dh_{t}/dW・d/dh_{t}+dh_{t-1}/dW・d/dh_{t-1}+・・)o_{t}
  #           = (x_{t} + x{t-1}・U+x_{t-2}・U^2+・・)・V
  # do/dU = (dh_{t}/dU・d/dh_{t}+dh_{t-1}/dU・d/dh_{t-1}+・・)o_{t}
```

```
#           = (h_{t-1} + h{t-2}・U+h_{t-3}・U^2+・・)・V
for t in reversed(range(n_seq)):
    dV += np.dot(do[:, t].T, hiddens[:, t]) / batch_size
    delta_t = do[:, t].dot(V)
    #  時間 t の出力は時間 t 以前の中間層すべてに依存するため
    # W, U はさらに遡って計算
    for bptt_step in reversed(range(t+1)):
        dW += np.dot(delta_t.T, xs[:, bptt_step]) / batch_size
        dU += np.dot(delta_t.T, hiddens[:, bptt_step-1]) / batch_size
        delta_t = (お)
return
```

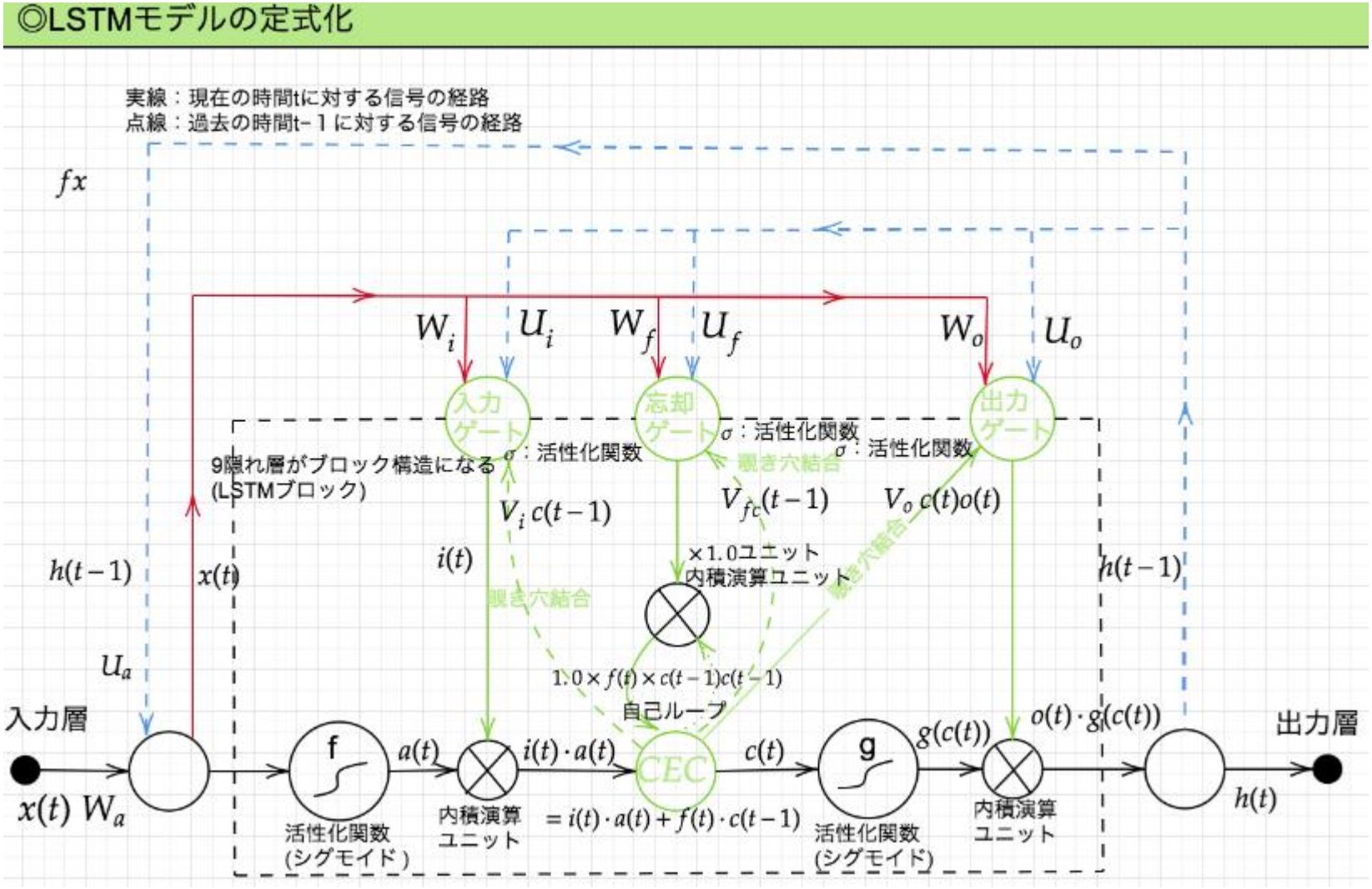
(1) delta_t.dot(W)
(2) delta_t.dot(U)
(3) delta_t.dot(V)
(4) delta_t * V

A. (2) delta_t.dot(U)
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 であるため、過去に遡るたびに U が掛けられる。そのため、delta_t = delta_t.dot(U)となるので(2)が正解となる。

<実装演習>
3_1_simple_RNN.ipynb
3_1_simple_RNN_after.ipynb

Section2：LSTM
RNN は時系列を遡れば遡るほど、勾配が消失していく課題があるため、長い時系列の学習が困難である。
この勾配が消失するという問題を構造自体を変えて解決する手法が LSTM である。

構造



CEC

勾配消失および勾配爆発の解決方法として、勾配が 1 であれば解決できる。

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

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

課題

入力データについては、時間依存度に関係なく重みが一律であるため、ニューラルネットワークの学習特性が無い。

過去の情報が全て保管されており、情報が不要となった場合でも削除されず保管され続けている。

CEC 自身の値は、ゲート制御に影響を与えていないため、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への伝播、忘却することができない。

入力ゲート・出力ゲート

それぞれのゲートへの入力値の重みを重み行列 W, U で可変可能とすることで、CEC のニューラルネットワークの学習特性が無いという課題を解決する役割を持つ。

忘却ゲート

過去の情報が要らなくなった場合に、そのタイミングで情報を忘却する(削除する)ことで、CEC の過去の情報が保管され続けるという課題を解決する役割を持つ。

覗き穴結合

CEC 自身の値に、重み行列を介して伝播可能にした構造であり、CEC の任意のタイミングで他ノードへの伝播や忘却をすることができないという課題を解決する役割を持つ。

<確認テスト 1>

Q. シグモイド関数を微分した時、入力値が 0 の時に最大値をとる。その値として正しいものを選択肢から選べ。

(1) 0.15

(2) 0.25

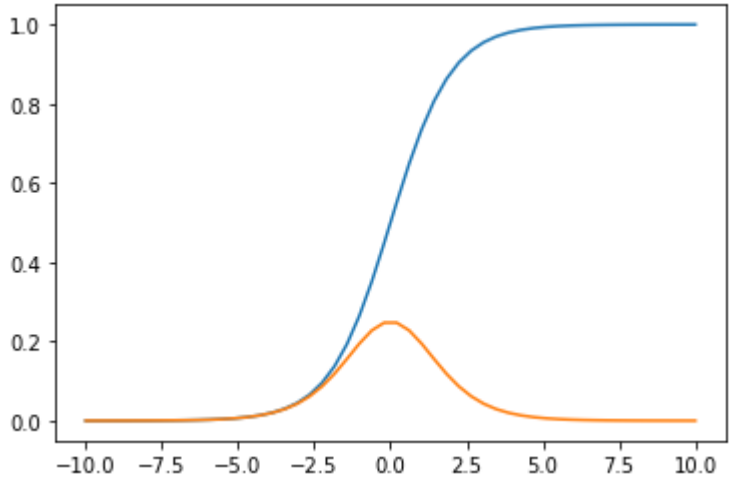
(3) 0.35

(4) 0.45

A. (2)

深層学習 Day2 の Section1：勾配消失問題の確認テストでも同様の問題が出題された。

シグモイド関数を微分すると、$f'(u) = (1 - \frac{1}{1+e^{-u}})\frac{1}{1+e^{-u}}$であり、これをプロットすると最大値が 0.25 であることがわかる。



青線　　：シグモイド関数
オレンジ線　　：シグモイド関数の微分値

<確認テスト 2>

Q. 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___。」

A. 忘却ゲート
入力ゲート、出力ゲート、忘却ゲートの 3 つのゲートが LSTM には存在するが、情報の忘却を行うゲートは忘却ゲートであるため。

<練習問題 1>

Q. 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。

```
def gradient_clipping(grad, threshold):
  “””
  grad: gradient
  “””
  norm = np.linalg.norm(grad)
  rate = threshold / norm
  if rate < 1:
     return (さ)
  return grad
```

(1) gradient * rate
(2) gradient / norm
(3) gradient /threshold
(4) np.maximum(gradient, threshold)

A. (1) gradient * rate
threshold がしきい値が設定されている変数、norm が勾配のノルムが設定されている変数、grad が勾配が設定されている変数、rate がしきい値/勾配のノルムの計算結果が設定されている変数である。
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)と計算するので、gradient * rate が正解となる。

<練習問題 2>
Q. 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。

```
def lstm(x, prev_h, prev_c, W, U, b)
```

```
    “””
    x: inputs, (batch_size, input_size)
    prev_h: outputs at the previous time step, (batch_size, state_size)
    prev_c: cell states at the previous time step, (batch_size, state_size)
    W: upward weights, (4*state_size, input_size)
    U: lateral weights, (4*state_size, state_size)
    B: bias, (4*state_size,)
    “””
    # セルへの入力やゲートをまとめて計算し、分割
    lstm_in = _activation(x.dot(W.T) + prev_h.dot(U.T) + b)
    a, i, f, o = np.hsplit(lstm_in, 4)

    # 値を変換、セルへの入力(-1, 1), ゲート(0, 1)
    a = np.tanh(a)
    input_gate = _sigmoid(i)
    forget_gate = _sigmoid(f)
    output_gate = _sigmoid(o)

    # セルの状態を更新し、中間層の出力 j を計算
    c = (け)
    h = output_gate = np.tanh(c)
    return c, h
```

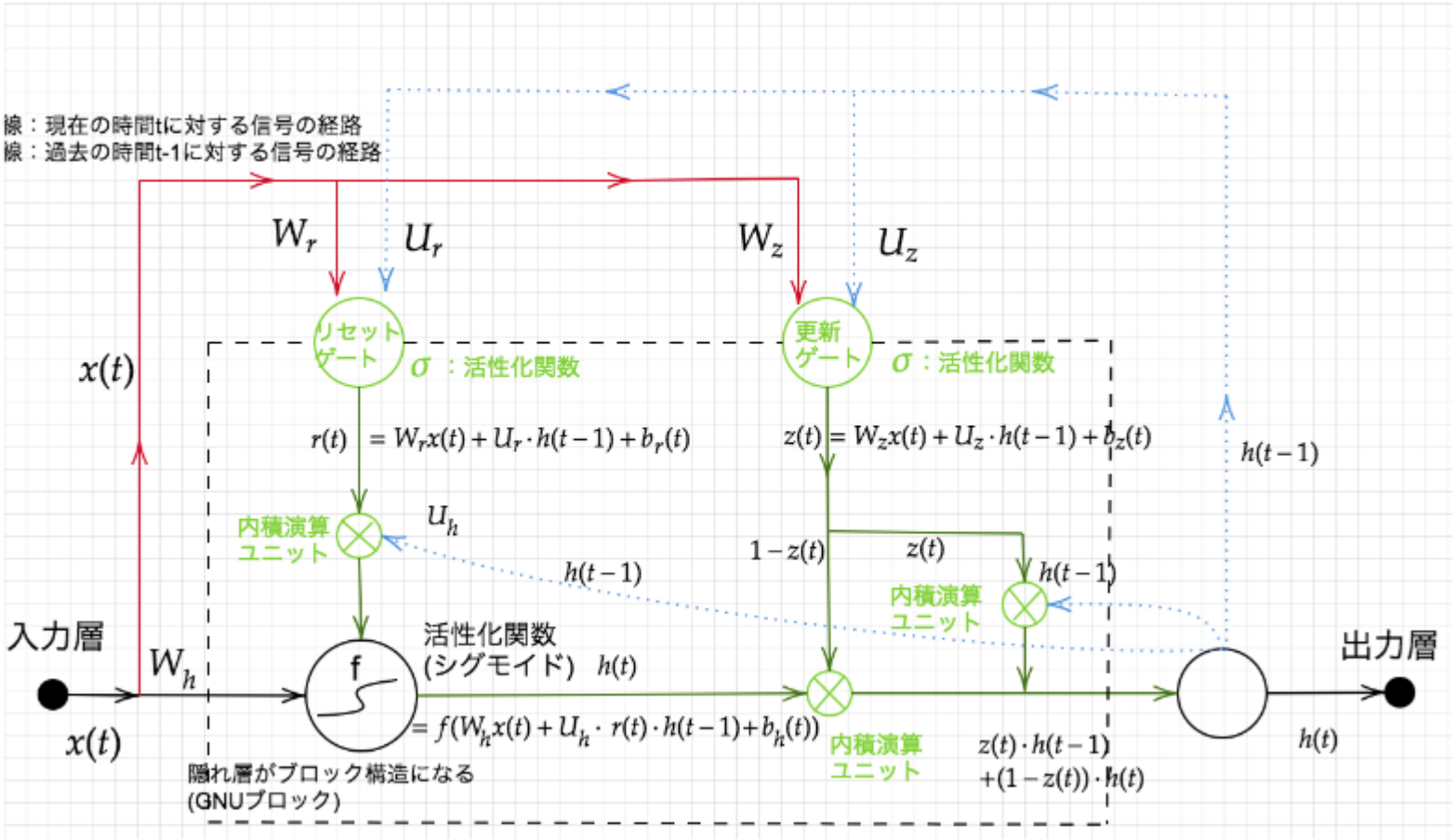
(1) output_gate * a + forget_gate * c
(2) forget_gate * a + output_gate * c
(3) input_gate * a + forget_gate * c
(4) forget_gate * a + input_gate * c

A. (3) input_gate * a + forget_gate * c

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。
a は入力値を活性化関数(np.tanh())で処理した値が設定されている変数であるため、input_gate と掛けるのが正しい。
forget_gate は 1 ステップ前のセルの値と掛けるので、正しくは forget_gate * prev_c だと思うが選択肢には存在しないので、一番近い(3)が正解。

Section3：GRU

LSTM では、パラメータ数が多いため計算負荷が高くなる問題があるため、この課題を解決するための手法。
GRU では、パラメータ数を大幅に削減し、精度は LSTM と同等またはそれ以上が望める構造となっている。
構造



<確認テスト 1>

Q. LSTM と CEC が抱える課題について、それぞれ簡潔に述べよ。

A. LSTM は入力ゲート、出力ゲート、忘却ゲート、CEC の 4 つの部品を持つことで構成されていた。
そのため、LSTM ではパラメータ数が多く、計算負荷が高くなるという課題がある。
CEC は、学習能力がないため、学習機能を持たせるために周りにゲートを付ける必要があるという課題がある。

<確認テスト 2>

Q. LSTM と GRU の違いを簡潔に述べよ。

A. LSTM はパラメータが多いが、GRU はパラメータが少ない。
パラメータ数の違いにより、GRU の方が計算量が少ない。
LSTM は 4 つの部品(入力ゲート、出力ゲート、忘却ゲート、CEC)で構成されているが、GRU は 2 つの部品(更新ゲート、リセットゲート)のみ。

<練習問題>

Q. 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を解決し、長期的な依存関係を学習することができる。LSTM に比べ変数の数やゲート数が少なく、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
```

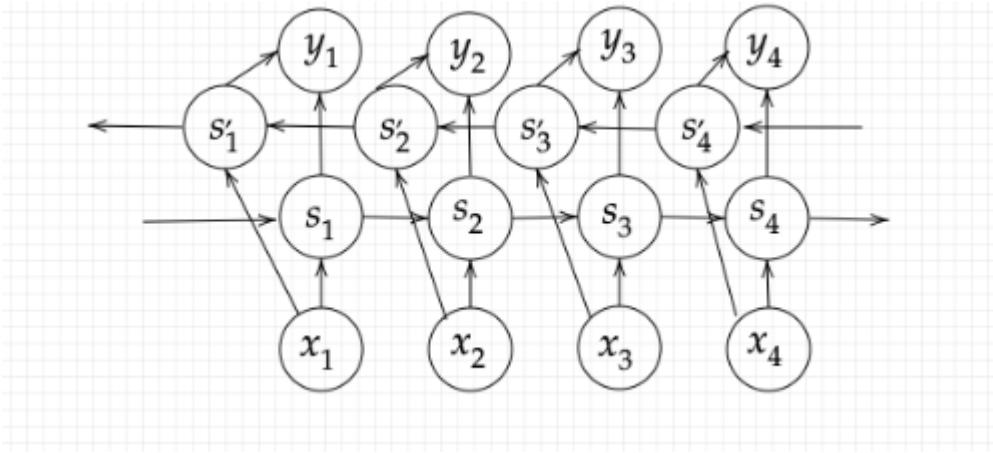
B. z * h_bar
C. (1-z) * h_bar
D. z * h * h_bar
E. (1-z) * h + z * h_bar

B. (4) (1-z) * h + z * h_bar
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ため、更新ゲート z を用いて、(1-z) * h + z * h_bar という計算となるため(4)が正解となる。

<実装演習>
predict_word.ipynb

Section4：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で、文書の推敲や機械翻訳に利用されている。

構造



<練習問題>

Q. 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
```

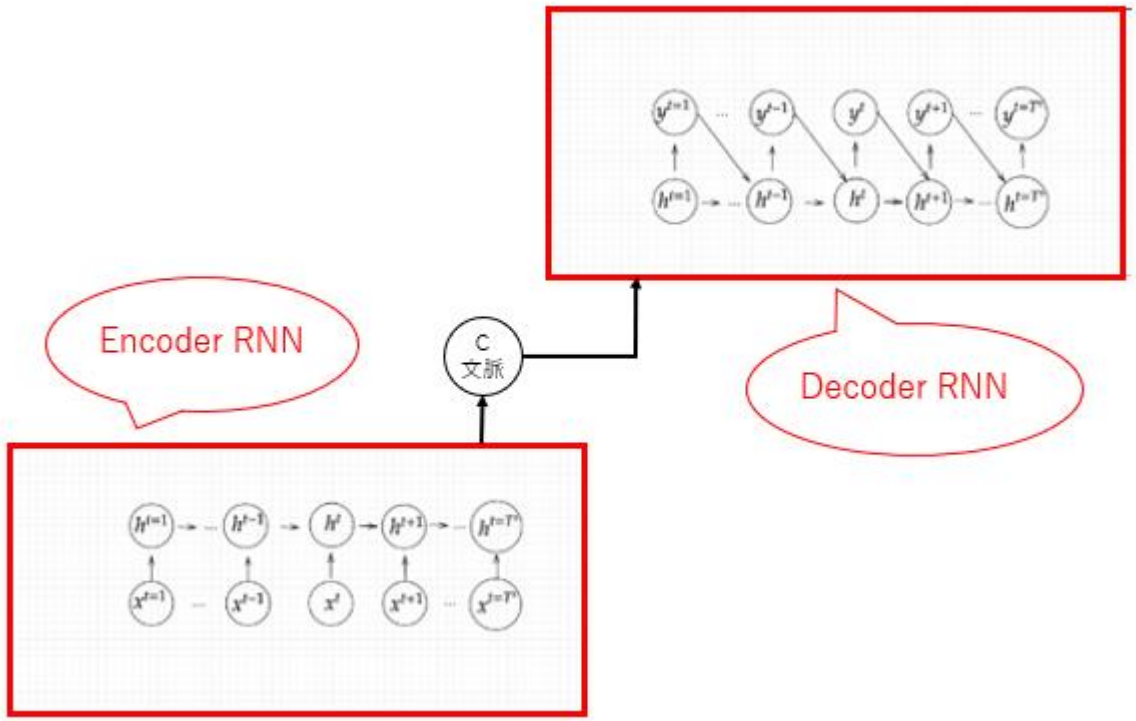
(1) h_f + h_b[::-1]
(2) h_f * h_b[::-1]
(3) np.concatenate([h_f, h_b[::-1]], axis=0)
(4) np.concatenate([h_f, h_b[::-1]], axis=1)

A. (4) np.concatenate([h_f, h_b[::-1]], axis=1)
順方向と逆方向に伝播したときの中間層表現を合わせたものが双方向RNNの特徴量となるため、np.concatenate()の演算をしているのが正解の対象となる。
また、列を軸として連結する必要があるため、axis=1 を指定している(4)が正解となる。

Section5：Seq2Seq

Seq2Seq とは Encoder-Decoder モデルの一種。
機械対話や機械翻訳などで使用されている。

Seq2Seq の全体図の例



Encoder RNN

ユーザーがインプットしたテキストデータを単語などのトークンに区切って渡す構造。

Taking：文章を単語などのトークン毎に分割し、トークン毎の ID に分割する。

Embedding：ID からそのトークンを表す分散表現ベクトルに変換する。

Encoder RNN：ベクトルを順番に RNN に入力していく。

処理手順

1. vec1 を RNN に入力し、hidden state を出力。
   この hidden state 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 hidden state を final state としてとっておく。
   この final state が thought vector と呼ばれ、入力した文の意味を表すベクトルとなる。

Decoder RNN

システムがアウトプットデータを単語などのトークンごとに生成する構造。

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、final state を Decoder RNN の initial state ととして設定して Embedding を入力する。
2. Sampling：生成確率にもとづいて token をランダムに選択する。
3. Embedding：上記の 2 で選ばれた token を Embedding して Decoder RNN への次の入力とする。
4. Detokenize：上記の 1〜3 を繰り返し、2 で得られた token を文字列に直す。

HRED

一問一答しかできない(問いに対して文脈など関係なく、ただ応答が行われ続ける)という Seq2Seq の課題を解決するために考えされた手法。

過去 n-1 個の発話から次の発話を生成する手法。

Seq2Seq は会話の文脈を無視して応答していたが、HRED では前の単語の流れに即して応答されるため、人間らしい文書が生成されやすい。

構造

Seq2Seq + Context RNN

Context RNN とは、Encoder のまとめた各文書の系列をまとめて、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応答ができるようになる。

課題

1. 確率的な多様性が字面のみのため、会話の流れのような多様性はない。
   同じコンテキストを与えられると同じ応答しか出せない。
2. 短く情報量の乏しい応答になりがちである。
   短いよくある答えを学ぶ傾向があるためこのようになる。

VHRED

HRED に、VAE の潜在変数の概念を追加した手法。

HRED の課題を VAE の潜在変数の概念を追加することで解説した構造となっている。

オートエンコーダー

教師なし学習の一つであり、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。

具体例

MNIST の場合、28×28 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。

構造

Encoder：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。

Decoder：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。

メリット

次元削減が行える。

※潜在変数 z の次元が入力データよりも小さい場合は、次元削減とみなすことができる。

VAE

通常のオートエンコーダーの場合は、何かしらの潜在変数 z にデータを押し込めているが、その構造がどのような状態かわからないため、
VAE はこの潜在変数 z に確率分布 z-N(0, 1)を仮定した手法。
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ができる。

<確認テスト 1>

Q. 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。

A. (2)
(1)は双方向 RNN、(3)は構文木、(4)は LSTM に関する説明のため、(2)が正解となる。

<確認テスト 2>
Q. Seq2Seq と HRED、HRED と VHRED の違いを簡潔に述べよ。

A. Seq2Seq は 1 問 1 答しかできない(問いに対して文脈も何もなく、ただ応答が行われ続ける)のに対して、
HRED は文書の過去の文脈というものを考慮できる。
HRED は確かに文脈を意識した文を作ることができるが、ありがちな答えしか出さなくなるのに対して、
VHRED は文脈を意識しながら文を作成するモデルにバリエーションを持たせられるように工夫をしてみようという仕組み。

<確認テスト 3>
Q. 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。

A. 確率分布
VAE はオートエンコーダーにおける潜在変数 z に確率分布を仮定したものであるため、空欄には確率分布が入る。

<練習問題>
Q. 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 embedding 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.shapre[0]
h = np.zeros(hidden_size)
for w in words:
  e = (き)
  h = _activation(W.dot€ + U.dot(h) + b)
return h
```

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

A. (1) E.dot(w)
Words から取り出された単語が w には設定されておりこれは one-hot ベクトル。
それを単語埋め込みにより別の特徴量に変換するため、埋め込み行列 E との積を計算する(1)が該当する。

Section6：word2vec

word2vec は単語の分散表現ベクトルを得る手法。
※「単語の分散表現」とは単語を固定長のベクトルで表現すること。

RNN では、単語のような可変長の文字列を NN に与えることができないため、固定長形式で単語を表す必要があるという課題があるため、word2vec という手法が必要となる。

メリット

大規模データの分散表現の学習が、現実的な計算速度とメモリ量で実現可能。

学習データからボキャブラリを作成し、各単語を one-hot ベクトルにして入力して単語分散表現する。
これにより、以下のようなボキャブラリ数×単語ベクトルの次元数の重み行列の計算を行うことで学習を行う。

$$[0,1,0,0,0,0,0,0,0,0,\cdots] * \begin{bmatrix} w_{11}^{(I)} & \cdots & w_{1I}^{(I)} \\ \vdots & \ddots & \vdots \\ w_{J1}^{(I)} & \cdots & w_{JI}^{(I)} \end{bmatrix} = [1, 13, 15]$$

Section7：Attention Mechanism

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いけないため、長い文書への対応が難しいという課題がある。
この課題を解決には、文書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となる。
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。

<確認テスト 1>

Q. RNN と word2vec、Seq2Seq と Attention の違いを簡潔に述べよ。

A. RNN は時系列データを処理するのに適したネットワーク。
word2vec は単語の分散表現ベクトルを得る手法。
Seq2Seq は 1 つの時系列データから別の時系列データを得るネットワーク。
Attention Mechanism は時系列データの中身に対して関連性に重みを付ける手法。

その他

<実装演習>

3_3_predict_sin.ipynb
data_augmentation_with_tf_shusei1031.ipynb
activation_functions.ipynb