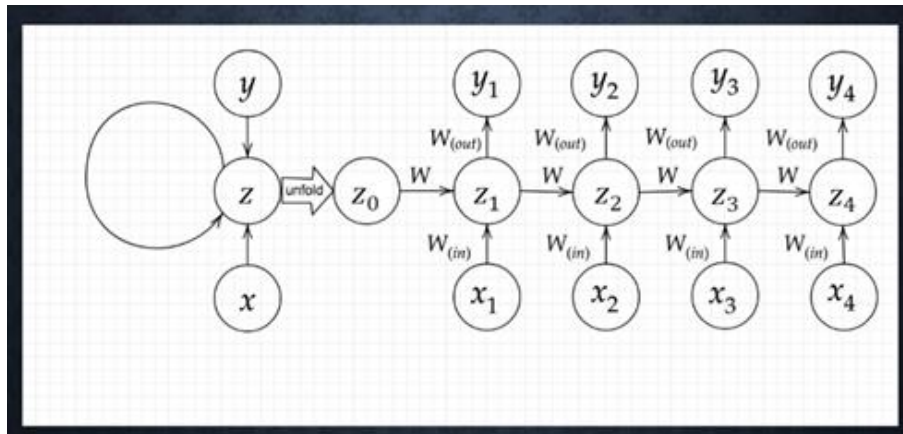


Stage4『深層学習 Day3』 レポート

●Section1：再帰型（Recurrent）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モデル。

- ・時系列データ…時間的順序を追って一定間隔毎に観測され、しかも相互に統計的依存関係が認められる様なデータの系列（音声データ、テキストデータなど）



⇒時刻 t の隠れ層の内容が、次の時刻 $t+1$ の時の入力として扱われます。 $t+1$ の隠れ層が $t+2$ の・・・と続く。つまり、前回の隠れ層が次の隠れ層の学習にも使用される。

$$u^t = \mathcal{W}_{(in)}x^t + \mathcal{W}z^{t-1} + b$$

`u[:, t+1] = np.dot(x, W_in) + np.dot(z[:, t], reshape(1, -1), W)`

$$z^t = f(u^t) = f(\mathcal{W}_{(in)}x^t + \mathcal{W}z^{t-1} + b)$$

`z[:, t+1] = functions.sigmoid(u[:, t+1])`

$$v^t = \mathcal{W}_{(out)}z^t + c = \mathcal{W}_{(out)}f(u^t) + c$$

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

$$y^t = g(v^t) = g(\mathcal{W}_{(out)}z^t + c) = g(\mathcal{W}_{(out)}f(u^t) + c)$$

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

- ・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型構造が必要になる。

補足(YouTube): <https://youtu.be/NJdrYvYgaPM>、<https://youtu.be/IcCIu5Gx6uA>

演習チャレンジ

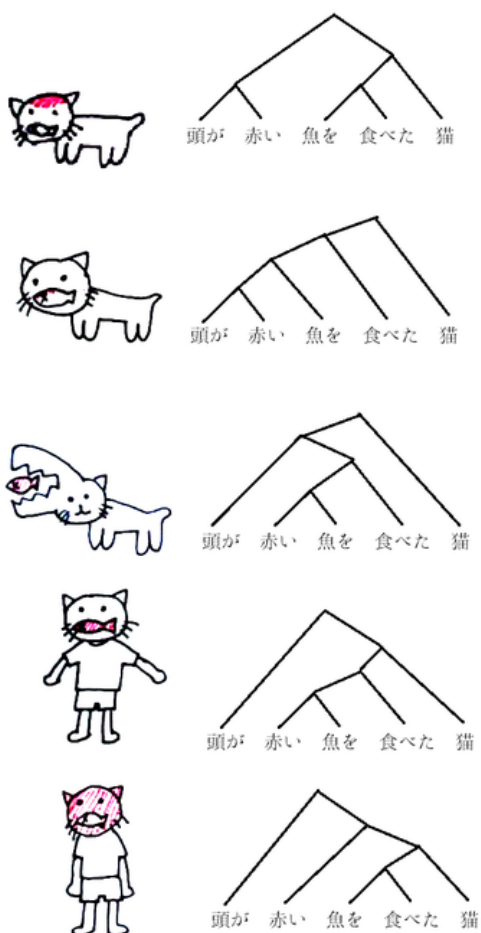
以下は再帰型ニューラルネットワークにおいて構文木を入力として再帰的に文全体の表現ベクトルを得るプログラムである。ただし、ニューラルネットワークの重みパラメータはグローバル変数として定義してあるものとし、activation関数はなんらかの活性化関数であるとする。本構造は再帰的な辞書で定義しており、rootが最も外側の辞書であると仮定する。
(\square) にあてはまるのはどれか。

```
def traverse(node):  
    """  
    node: tree node, recursive dict, {left: node', right: node''}  
    if leaf, word embed vector, (embed_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(  
            return v
```

(1) $W \cdot \text{dot}(\text{left} + \text{right})$
(2) $W \cdot \text{dot}(\text{np.concatenate}([\text{left}, \text{right}]))$
(3) $W \cdot \text{dot}(\text{left} * \text{right})$
(4) $W \cdot \text{dot}(\text{np.maximum}(\text{left}, \text{right}))$

演習チャレンジ

正解: 2
【解説】
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、 $W \cdot \text{dot}(\text{np.concatenate}([\text{left}, \text{right}]))$ である。



参考サイト: [構文木を書き表すためのシンプルな表記法](#)

◇BTTP : Back Propagation Through Time

RNN におけるパラメータ調整方法 (誤差逆伝播の一種)

【重み: $\mathcal{W}_{(in)}$, $\mathcal{W}_{(out)}$, \mathcal{W} の勾配】

$$\textcircled{1} \frac{\partial E}{\partial \mathcal{W}_{(in)}} = \frac{\partial E}{\partial \mathbf{u}^t} \left[\frac{\partial E}{\partial \mathcal{W}_{(in)}} \right]^T$$

ここで、E を \mathbf{u}^t で偏微分したものの $\frac{\partial E}{\partial \mathbf{u}^t}$ を δ^t とおく。

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

```
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])
```

$$\therefore \frac{\partial E}{\partial \mathcal{W}_{(in)}} = \frac{\partial E}{\partial \mathbf{u}^t} \left[\frac{\partial E}{\partial \mathcal{W}_{(in)}} \right]^T = \delta^t [\mathbf{x}^t]^T$$

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

$$\textcircled{2} \frac{\partial E}{\partial \mathcal{W}_{(out)}} = \frac{\partial E}{\partial \mathbf{u}^t} \left[\frac{\partial \mathbf{v}^t}{\partial \mathcal{W}_{(out)}} \right]^T = \delta^{out,t} [\mathbf{z}^t]^T$$

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

$$\textcircled{3} \frac{\partial E}{\partial \mathcal{W}} = \frac{\partial E}{\partial \mathbf{u}^t} \left[\frac{\partial E}{\partial \mathcal{W}} \right]^T = \delta^t [\mathbf{z}^{t-1}]^T$$

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

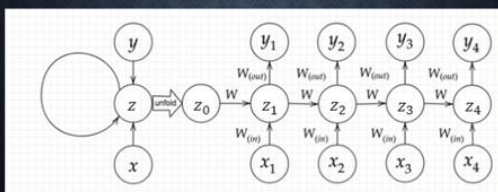
【バイアス:b,c の更新】

$$\textcircled{4} \frac{\partial E}{\partial \mathbf{b}} = \frac{\partial E}{\partial \mathbf{u}^t} \frac{\partial \mathbf{u}^t}{\partial \mathbf{b}} = \delta^t \left(\frac{\partial \mathbf{u}^t}{\partial \mathbf{b}} = \mathbf{1} \right)$$

$$\textcircled{5} \frac{\partial E}{\partial \mathbf{c}} = \frac{\partial E}{\partial \mathbf{v}^t} \frac{\partial \mathbf{v}^t}{\partial \mathbf{c}} = \delta^{out,t} \left(\frac{\partial \mathbf{v}^t}{\partial \mathbf{c}} = \mathbf{1} \right)$$

確認テスト

下図の y_1 を $\mathbf{x} \cdot \mathbf{s}_0 \cdot \mathbf{s}_1 \cdot \mathbf{W}_{in} \cdot \mathbf{W} \cdot \mathbf{W}_{out}$ 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。
 (7分)



回答：

$$\begin{aligned} y^1 &= g(v^1) = g(\mathcal{W}_{(out)}s^1 + c) = g(\mathcal{W}_{(out)}f(u^1) + c) \\ &= g(\mathcal{W}_{(out)}f(\mathcal{W}_{(in)}x^1 + \mathcal{W}s^0 + b) + c) \end{aligned}$$

$$\ast g, f = \text{sigmoid}$$

◇ δ^t の時間における関係式

$$\begin{aligned} \cdot \delta^t &= \frac{\partial E}{\partial v^t} \frac{\partial v^t}{\partial u^t} = \frac{\partial E}{\partial v^t} \frac{\partial (\mathcal{W}_{(out)}f(u^t) + c)}{\partial u^t} = f'(u^t) \mathcal{W}_{(out)}^T \delta^{out,t} \\ \cdot \delta^{t-1} &= \frac{\partial E}{\partial u^{t-1}} = \frac{\partial E}{\partial v^t} \frac{\partial v^t}{\partial u^{t-1}} = \delta^t \left\{ \frac{\partial u^t}{\partial z^{t-1}} \frac{\partial z^{t-1}}{\partial u^{t-1}} \right\} = \delta^t \{ \mathcal{W} f'(u^{t-1}) \} \\ \cdot \delta^{t-z-1} &= \delta^{t-z} \{ \mathcal{W} f'(u^{t-z-1}) \} \end{aligned}$$

◇パラメータの更新

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

$$W_in := \text{learning_rate} * W_in_grad$$

$$\textcircled{2} \mathbf{w}_{(out)}^{t+1} = \mathbf{w}_{(out)}^t - \varepsilon \frac{\partial E}{\partial \mathbf{w}_{(out)}^t} = \mathbf{w}_{(out)}^t - \varepsilon \delta^{out,t} [z^t]^T$$

$$W_out := \text{learning_rate} * W_out_grad$$

$$\textcircled{3} \mathbf{w}^{t+1} = \mathbf{w}^t - \varepsilon \frac{\partial E}{\partial \mathbf{w}} = \mathbf{w}_{(in)}^t - \varepsilon \sum_{z=0}^{T_t} \delta^{t-z} [x^{t-z-1}]^T$$

$$W := \text{learning_rate} * W_grad$$

$$\textcircled{4} \mathbf{b}^{t+1} = \mathbf{b}^t - \varepsilon \frac{\partial E}{\partial \mathbf{b}} = \mathbf{b}^t - \varepsilon \sum_{z=0}^{T_t} \delta^{t-z}$$

$$\textcircled{5} \mathbf{c}^{t+1} = \mathbf{c}^t - \varepsilon \frac{\partial E}{\partial \mathbf{c}} = \mathbf{c}^t - \varepsilon \delta^{out,t}$$

$$(\ast \varepsilon = \text{学習率})$$

◇誤差関数

$$\begin{aligned} E^t &= \text{loss}(\boldsymbol{y}^t, \boldsymbol{d}^t) \\ &= \text{loss}(\delta(\boldsymbol{W}_{(out)} \boldsymbol{z}^t + \boldsymbol{c}), \boldsymbol{d}^t) \\ &= \text{loss}(\delta(\boldsymbol{W}_{(out)} f(\boldsymbol{W}_{(in)} \boldsymbol{x}^t + \boldsymbol{W} \boldsymbol{z}^{t-1} + \boldsymbol{b}) + \boldsymbol{c}), \boldsymbol{d}^t) \\ &= \text{loss}(\delta(\boldsymbol{W}_{(out)} f(\boldsymbol{W}_{(in)} \boldsymbol{x}^t + \boldsymbol{W} f(\boldsymbol{u}^{t-1}) + \boldsymbol{b}) + \boldsymbol{c}), \boldsymbol{d}^t) \\ &= \text{loss}(\delta(\boldsymbol{W}_{(out)} f(\boldsymbol{W}_{(in)} \boldsymbol{x}^t + \boldsymbol{W} f(\boldsymbol{W}_{(in)} \boldsymbol{x}^{t-1} + \boldsymbol{W} \boldsymbol{z}^{t-2} + \boldsymbol{b}) + \boldsymbol{b}) + \boldsymbol{c}), \boldsymbol{d}^t) \\ &\vdots \end{aligned}$$

誤差関数 E は、z の時間 t が数珠つなぎとなっている。

```
all_losses = []
```

```
for i in range(iters_num):
```

```
    # A, B 初期化 (a + b = d)
```

```
    a_int = np.random.randint(largest_number/2)
```

```
    a_bin = binary[a_int] # binary encoding
```

```
    b_int = np.random.randint(largest_number/2)
```

```
    b_bin = binary[b_int] # binary encoding
```

```
    # 正解データ
```

```
    d_int = a_int + b_int
```

```
    d_bin = binary[d_int]
```

```
    # 出力バイナリ
```

```
    out_bin = np.zeros_like(d_bin)
```

```
    # 時系列全体の誤差
```

```
    all_loss = 0
```

```
    # 時系列ループ
```

```
    for t in range(binary_dim):
```

```
        # 入力値
```

```
        X = np.array([a_bin[-t-1], b_bin[-t-1]]).reshape(1, -1)
```

```
        # 時刻 t における正解データ
```

```
        dd = np.array([d_bin[binary_dim-t-1]])
```

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


#

```
loss = functions.mean_squared_error(dd, y[:,t])
```

```
delta_out[:,t] = functions.d_mean_squared_error(dd, y[:,t]) * functions.d_sigmoid(y[:,t])
```

```
all_loss += loss
```

参考サイト：<https://qiita.com/kiminaka/items/87afd4a433dc655d8cfd>

コード演習問題

```
def bptt(x, y, W, U, V):
    """
    y: labels, [batch_size, n_seq, output_size)
    """
    # 初期化
    hidden, outputs = np.zeros(W, U, V)
    # 初期値の0, 0, 0を代入する必要がある
    dh = np.zeros_like(W)
    dt = np.zeros_like(U)
    dv = np.zeros_like(V)
    # 逆方向伝播の方向に流す必要がある
    do = -calculate_dout(outputs, y)
    # dh, dt, [batch_size, n_seq, output_size)
    batch_size, n_seq = y.shape[1:2]
    # 逆方向伝播の方向に、バッチサイズとシーケンス長を流す (バッチサイズはバッチサイズ)
    # do[0,0] = do[batch_size, n_seq, 0]
    # do[0,1] = do[batch_size, n_seq, 1]
    # do[0,2] = do[batch_size, n_seq, 2]
    # do[1,0] = do[batch_size, 0, 0]
    # do[1,1] = do[batch_size, 0, 1]
    # do[1,2] = do[batch_size, 0, 2]
    # do[2,0] = do[batch_size, 1, 0]
    # do[2,1] = do[batch_size, 1, 1]
    # do[2,2] = do[batch_size, 1, 2]
    for t in reversed(range(n_seq)):
        dv += np.dot(doi, tl, hidden[t, t] / batch_size
        delta_t = doi, tl, dt(V)
        # 逆方向伝播の方向に、バッチサイズとシーケンス長を流す (バッチサイズはバッチサイズ)
        # do[0,0] = do[batch_size, n_seq, 0]
        # do[0,1] = do[batch_size, n_seq, 1]
        # do[0,2] = do[batch_size, n_seq, 2]
        # do[1,0] = do[batch_size, 0, 0]
        # do[1,1] = do[batch_size, 0, 1]
        # do[1,2] = do[batch_size, 0, 2]
        # do[2,0] = do[batch_size, 1, 0]
        # do[2,1] = do[batch_size, 1, 1]
        # do[2,2] = do[batch_size, 1, 2]
        for bptt_step in reversed(range(tell)):
            dw += np.dot(delta_t, x[bptt_step] / batch_size
            du += np.dot(delta_t, hidden[bptt_step-1] / batch_size
            dv_t = (A)
            delta_t =
```

左の図はBPTTを行うプログラムである。なお簡便化のため活性化関数は恒等関数であるとする。

また、calculate_dout関数は損失関数の出力に關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。

- (1) $\delta_t \cdot \dot{W}$
- (2) $\delta_t \cdot \dot{U}$
- (3) $\delta_t \cdot \dot{V}$
- (4) $\delta_t \cdot V$

正解: 2

【解説】
RNNでは中間層出力 $h_t(t)$ が過去の中間層出力 $h_{t-1}(t-1), \dots, h_1(1)$ に依存する。

RNNにおいて損失関数を重み W だけに關して偏微分するときは、それを考慮する必要がある。

$$dh_t(t)/dh_{t-1}(t-1) = U$$

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、 $\delta_t \cdot U = \delta_t \cdot \dot{U}$ となる。

- (1) $\delta_t \cdot \dot{W}$
- (2) $\delta_t \cdot \dot{U}$
- (3) $\delta_t \cdot \dot{V}$
- (4) $\delta_t \cdot V$

〈RNN の課題〉

- ・ 課題①…時系列を遡れば遡るほど、勾配が消失していく（長い時系列の学習が困難）

⇒解決策：構造自体を変えたモデル…LSTM

- ・課題②…勾配爆発（層を逆伝播するごとに指数関数的に勾配が大きくなっていく。

⇒解決策：勾配クリッピング（勾配ノルムが閾値を超えたら、勾配ノルムを閾値に正規化する）

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大きい値を超えたら、勾配のノルムを小さ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。

(a) 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 (a)
    return grad
```

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

解答

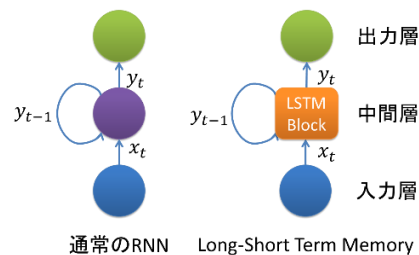
正解：1

【解説】

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

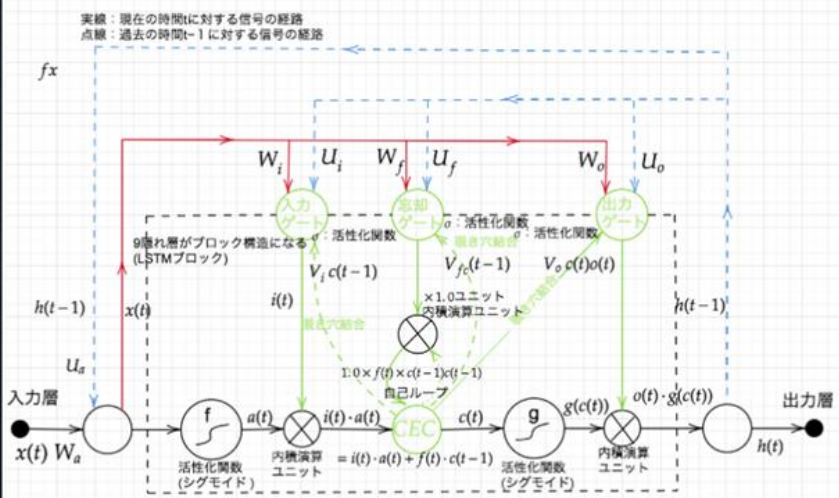
●Section2 : LSTM (Long Short-Term Memory)

長い時系列の学習/記憶が困難な RNN の改良モデル。



LSTMの全体図

LSTMモデルの定式化



◇CEC (Constant Error Carousel)

役割：記憶機能しか持たない（メモリセル）。

$$c^{(t)} = i^{(t)} \circ a^{(t)} + f^{(t)} \circ c^{(t-1)}$$

誤差消滅/爆発問題に対応するために導入された非常にシンプルなアプローチ。

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

を満たせば理論上、無限時間であっても誤差は消滅/爆発も起こさず伝播する。

〈CEC の課題〉

入力データに対し、時間依存度に関係なく重みが一律である（学習機能が無い）。

⇒矛盾する重み更新（入力重み衝突/出力重み衝突）が頻発し、LSTM 学習が進まない…。

◇入力/出力ゲート

入力/出力ゲートを導入し、追加の重みパラメタを持たせることで、「前のユニット（1つ前の時間のユニット）の入力を受け取るか否か」を判断させる、すなわち、必要に応じて誤差信号の伝播をゲート部で止め、必要な誤差信号だけが適切に伝播するようにゲートを開いたり閉じたりする。

⇒各々のゲートへの入力値に掛かる重み行列 W, U で可変可能（BPTT により更新）とする事で、CEC の課題を解決できる。

◇忘却ゲート

CEC は、過去の情報が全て保管され続けてしまい、不要になった情報に影響されてしまいます事がある…。過去の情報が不要となった時点で、情報を削除する機能が必要。

⇒忘却ゲートは、誤差信号を受け取ることで、一度 CEC で記憶した内容を一気に「忘れる」ことを学習する。

補足（YouTube）<https://youtu.be/oxygME2UBFc>



回答：忘却ゲート

ゲートの導入によって勾配「消滅」問題が解決できたが、勾配「爆発」問題には対応できていない。

⇒勾配のノルムに対して一定の制約値(hard constraint)を設け、ミニバッチの学習毎に大きくなりすぎた勾配のノルムを補正するという方法＝勾配クリッピングが取られるようになった。

$$\hat{g} \leftarrow \frac{\partial \epsilon}{\partial \theta}$$

$$\text{if } \|\hat{g}\| \geq \text{threshold}$$

$$\text{then } \hat{g} \leftarrow \frac{\text{threshold}}{\|\hat{g}\|} \hat{g}$$

更新の度に、 $-\text{threshold} < \|\text{gradient}\| < \text{threshold}$ を保証すればよい。

(注) これは、BPTT 法によって学習する一般の RNN に適用できる勾配消失/爆発問題への対処法をも示唆している。

参考サイト：https://qiita.com/t_Signull/items/21b82be280b46f467d1b

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用いる関数である。
(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,)
    """
    # セルへの入力ゲートをまとめた計算し、中間
    lstm_in = _activation(x.dot(W.T) + prev_h.dot(U.T) + b)
    a, i, f, o = np.hsplit(lstm_in, 4)

    # 閾値を適用、セルへの入力(1-3)、ゲート(4, 2)
    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、細胞状態の計算
    c = ( )
    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

解答

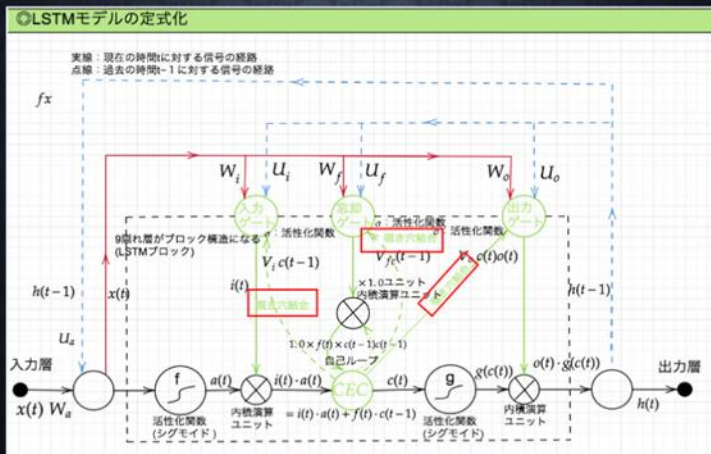
正解：3
【解説】
新しいセルの状態は、計算されたセルへの入力と1ステップ前のセルの状態に入力ゲート、忘却ゲートを掛けて足し合わせたものと表現される。つまり、input_gate * a + forget_gate * cである。

◇覗き穴結合

CEC 自身の値は、ゲート制御に影響を与えていない。

⇒CEC 自身の値に、重み行列を介して伝播可能した構造

覗き穴結合



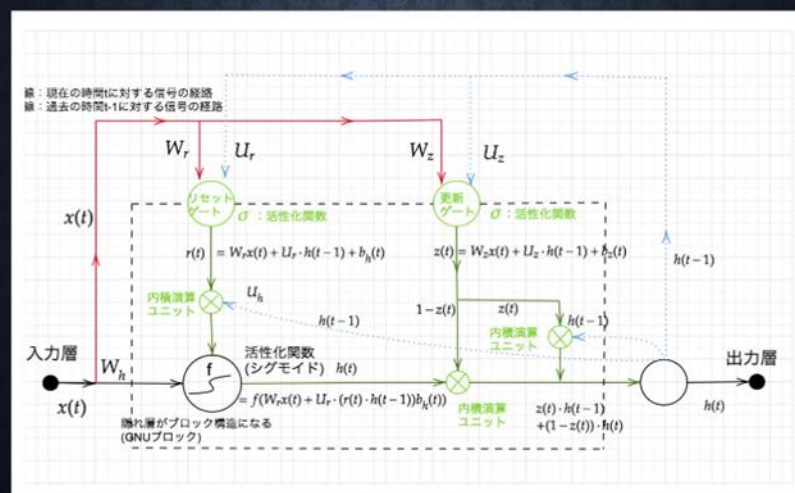
●Section3 : GRU (Gated Recurrent Unit)

LSTM はパラメータ数が多く、計算負荷（学習コスト）が高い・・・。

⇒パラメータを大幅に削減でき、精度は同等（またはそれ以上）が出せるモデル

(LSTM のフルモデルチェンジ版)

GRUの全体像



CEC/入力ゲート/出力ゲート/忘却ゲートが除かれ、リセットゲート/更新ゲートが追加されたシンプルな構造であり、計算負荷が小さい。

補足(YouTube): <https://youtu.be/K8ktkhAEuLM>

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) $(1-z) * h + z * h_bar$

解答

正解: 4

【解説】

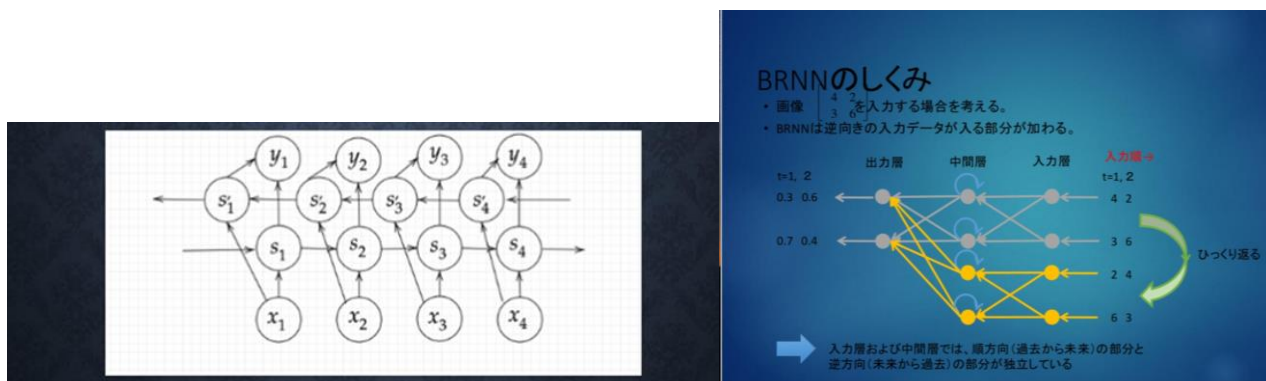
新しい中間状態は、1ステップ前の中間表現と計算された中間表現の線形和で表現される。つまり更新ゲート z を用いて、 $(1-z) * h + z * h_bar$ と書ける。

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

●Section4：双方向 RNN (Bi-directional RNN)

RNN は順伝播において過去→未来への一方向のみである事に対し、B-RNN は、過去の情報だけでなく、未来→過去の逆伝播を加味する事で、精度を向上させるためのモデル。

用途：文章の推敲、機械翻訳など



参考サイト：[双方向 RNN \(Bi-directional RNN\) とは？](#)

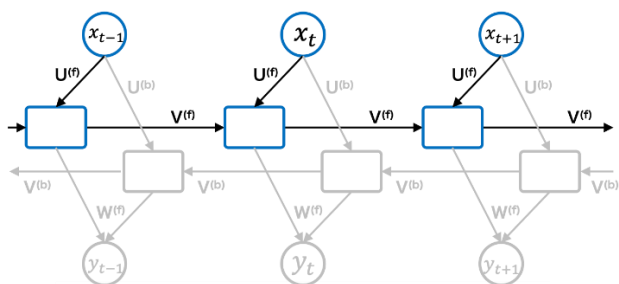
Bi-RNN では、学習時に、過去と未来の情報の入力が必要とすることから、運用時も過去から未来までのすべての情報を入力してはじめて予測できるようになる。

そのため、Bi-RNN の応用範囲が限定される。例えば、DNA 塩基を k-mer ごとに区切れば、塩基配列解析に Bi-RNN が使えるようになる。あるいは、1 文全体を入力して、文中にある誤字・脱字の検出などに応用されている。

順伝播では、状態 t における中間層は、入力値と状態 $t-1$ から入力を受け取り、それぞれに重みをかけて、活性化関数に代入して得られる値を、この中間層の出力値としている。

$$\vec{z}_t = U^{(f)}x_t + V^{(f)}z_{t-1} + b^{(f)}_z$$

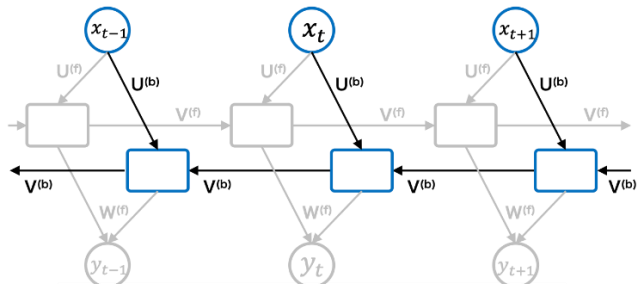
$$\vec{H}_t = \phi(\vec{z}_t)$$



$$\vec{z}_t = U^{(b)}x_t + V^{(b)}z_{t-1} + b^{(b)}_z$$

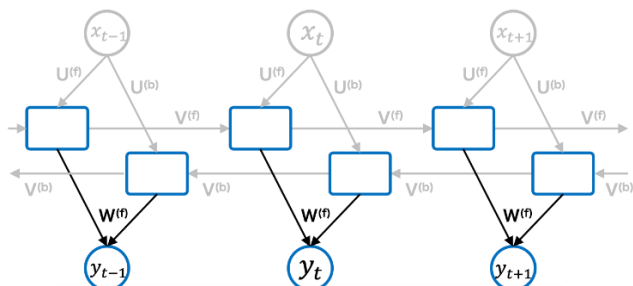
$$\vec{H}_t = \phi(\vec{z}_t)$$

逆伝播では、状態 t における中間層は、入力値と状態 $t+1$ から入力を受け取り、それぞれに重みをかけて、活性化関数に代入して得られる値を、この中間層の出力値としている。



状態 t における出力層の出力値は、順伝播および逆伝播の中間層の値から計算する。具体的に順伝播の状態を表すベクトル \vec{H}_t と逆伝播の状態を表すベクトル \vec{H}_t を縦に結合して一つのベクトルを作る。このとき、 $\vec{H}_t \in \mathbb{R}^{n \times h}$ および $\vec{H}_t \in \mathbb{R}^{n \times h}$ ならば、結合後のベクトルの次元は $\mathbf{R}_{n \times 2h}$ となる。ベクトルの結合後に、これに重みをかけ、それを活性化関数に代入して出力値を計算する。なお、出力層の活性化関数は恒等関数などが使われる。

$$y_t = \phi \left(\begin{pmatrix} \vec{H}_t \\ \vec{H}_t \end{pmatrix} W + b_y \right)$$



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 、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。(カ)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 = [
        (カ)
        for h_f, h_b in zip(hs_f, hs_b)]
    ys = hs.dot(V)
    return ys
```

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

正解：4

【解説】

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を合わせたものが特徴量となるので、 $\text{np.concatenate}([h_f, h_b[::-1]], \text{axis}=1)$ である。

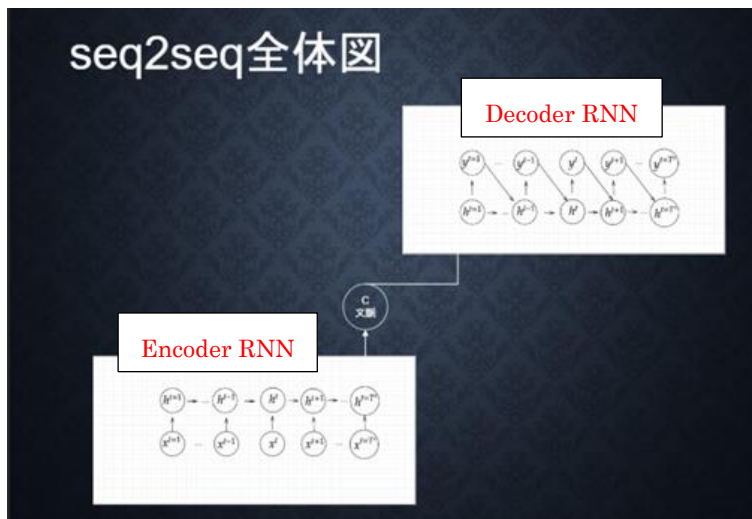
- (1) $h_f + h_b[::-1]$
- (2) $h_f * h_b[::-1]$
- (3) $\text{np.concatenate}([h_f, h_b[::-1]], \text{axis}=0)$
- (4) $\text{np.concatenate}([h_f, h_b[::-1]], \text{axis}=1)$

●Section5：seq2seq

Encoder-Decoder モデルの一種。一つの時系列データから別の時系列データを得る事が可能であり、機械対話・翻訳などに使用されている。

(シーケンスのペアを大量に学習させることで、片方のシーケンスからもう一方を生成するモデル)

- 翻訳：英語 → フランス語 のペアを学習。英語を入力するとフランス語に翻訳。
- 構文解析：英語 → 構文木 のペアを学習。英語を入力すると構文木を返す。
- 会話 bot：問いかけ → 返答 のペアを学習。「お腹減った」に対して「ご飯行こうぜ」などと返してくれる。



入力を日本語の文章とし、出力を英語の文章とする「機械翻訳」、入力を人間の言語入力とし、応答をロボットの言語出力とする「チャットボット」これらは入力/出力のシーケンスをペアで学習させることで精度向上の実現が期待できる。

構造は LSTM などの RNN 系を用いたニューラルネットワーク二つの部分 (EncoderRNN-DecoderRNN) から構成される。

- 前処理：単語を ID 付与・one-hot ベクトル化し、Embedding 変換する。

単語	ID	one-hot	embedding
私	1	[1, 0, 0 ... 0]	[0.2 0.4 0.6 ... 0.1]
は	2	[0, 1, 0 ... 0]	[...]
刺身	3	[0, 0, 1 ... 0]	[...]
昨日	4	[...]	[...]
⋮	⋮	⋮	⋮
xxx	10000	[0, 0, 0 ... 1]	[...]


WordEmbedding…単語埋め込み：機械学習により、意味が似ている単語を似ているベクトルへ変換する。

https://www.tensorflow.org/tutorials/text/word_embeddings?hl=ja

① Encoder RNN：文章から特徴を抽出する機能

インプットされ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 (Tokenizer)。

Encoder RNN



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。

Embedding: IDから、そのトークンを表す分散表現ベクトルに変換。

Encoder RNN: ベクトルを順番にRNNに入力していく。

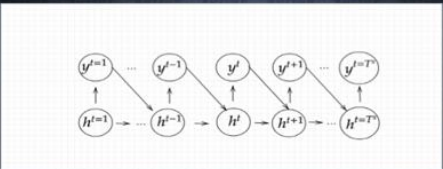
Encoder RNN処理手順

- ・ 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。
- ・ 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。

② Decoder RNN：特徴から文章を生成する機能

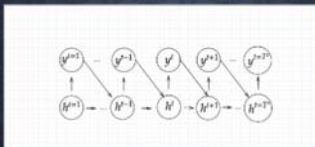
Decoder RNN

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。



お 腹 が 痛 い で す

Decoder RNNの処理



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び。

- (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。

(3分)

回答：(2)

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）時系列の情報をもち特徴へとエンコードされる。以下は、入力である文（文章）の時系列の情報をもち特徴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。

(a) 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.shape[0]
    h = np.zeros(hidden_size)
    for w in words:
        e = (8)
        h = _activation(W.dot(e) + U.dot(h) + b)
    return h
```

- (1) $E \cdot \text{dot}(w)$
- (2) $E.T \cdot \text{dot}(w)$
- (3) $w \cdot \text{dot}(E.T)$
- (4) $E * w$

※E:word embedding matrix…単語に対する embedding 表現の対応表

正解：1

【解説】

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、 $E \cdot \text{dot}(w)$ と書ける。

〈seq2seq の課題〉

一問一答しかできない（問いに対して文脈がなく、ただ機械的な応答がされる…。）

◇HRED（エイチ・レッド）：seq2seq + Context RNN

過去 $n-1$ 個の発話から次の発話 n を生成する（前の単語の流れに即した応答が出来る、より人間っぽい文章となる）

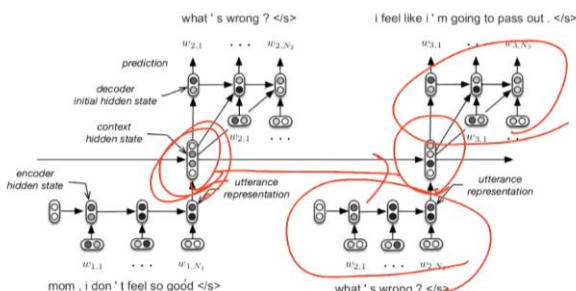


Figure 1: The computational graph of the HRED architecture for a dialogue composed of three turns. Each utterance is encoded into a dense vector and then mapped into the dialogue context, which is used to decode (generate) the tokens in the next utterance. The encoder RNN encodes the tokens appearing within the utterance, and the context RNN encodes the temporal structure of the utterances appearing so far in the dialogue, allowing information and gradients to flow over longer time spans. The decoder predicts one token at a time using a RNN. Adapted from Sordani et al. (2015a).

・ Context RNN: Encoder の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により、過去の発話の履歴を加味した返答ができる。

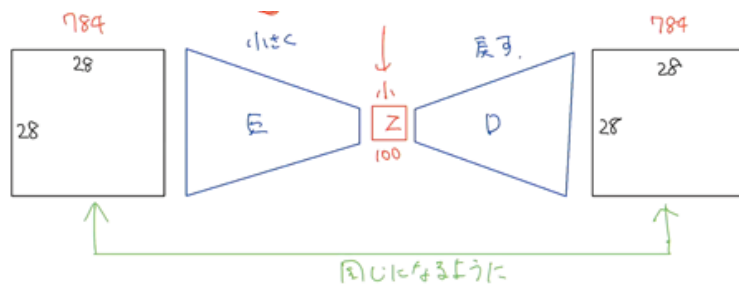
〈HRED の課題〉

HRED は確率的な多様性が字面にしかなく、会話の『流れ』の様な多様性が無い。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。HRED は短く情報量に乏しい答えをしがちである。（ありがちな短い答えを学ぶ傾向がある。ex) 「うん」「そうだね」「・・・」など）

⇒VAE (Variational AutoEncoder) の潜在変数の概念を追加。

※AutoEncoder (AE) : 教師なし学習の一つ（教師は入力データ自身）。

encoder で特徴量を少ない次元で抽出（次元削減を行う w_e, b_e を学習）し、decoder では、次元削減された特徴量から元に戻す（復元を行う w_d, b_d を学習する）。



AE では、何かしらの潜在変数 z にデータを圧縮しているものの、その構造の状態が分からない・・・。

VAE は、この潜在変数 z に確率分布 $z \sim \mathcal{N}(0,1)$ を仮定し、その分布構造に圧縮する。

※入力データ間の類似度（似たもの同士が持つ特徴）を残したまま、圧縮/復元が可能になる→高い汎用性が得られる。

◇VHRED (ブイ・エイチ・レッド)

HRED に、VAE の潜在変数の概念を追加したもの

●Section6 : word2vec

NN 入力の前処理 (Embedding 変換) 手法の一つ (一つのモデルの事ではない)。

RNN では、単語の様な可変長の文字列をそのまま NN に与える事は出来ない・・・。

one-hot では、単語数と同じ桁のベクトルとなり、計算が膨大になってしまう・・・。

単語	ID	one-hot
私	1	[1, 0, 0 ... 0]
は	2	[0, 1, 0 ... 0]
刺身	3	[0, 0, 1 ... 0]
昨日	4	[...]
⋮	⋮	⋮
×××	10000	[0, 0, 0 ... 1]

word2vec は、可変長の文字列→固定長のベクトル形式に変換 (単語の分散ベクトル化) する手法。

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。

→ X: ボキャブラリ×ボキャブラリだけの重み行列が誕生。
○: 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。

$[0, 1, 0, 0, 0, 0, 0, 0, 0, \dots]$ $\begin{bmatrix} w_{11}^{(l)} & \dots & w_{1i}^{(l)} \\ \vdots & \ddots & \vdots \\ w_{ji}^{(l)} & \dots & w_{ji}^{(l)} \end{bmatrix} = [1, 13, 15]$

ボキャブラリ数×単語ベクトルの次元数の重み行列。

補足(YouTube): <https://youtu.be/0CXCqXQAKKQ>

単語を表すベクトルを、それよりも低次元空間に写像する順伝播型ニューラルネットワークは Embedding 層、すなわち、やってることは Dense 層と変わらない (もちろん Embedding 層の入力形式はスカラーで Dense 層の入力形式はベクトルという点は異なる)。

より厳密にいうと、Dense 層はバイアスを含んでいるため、線形写像というよりはアフィン写像をしていることも異なります。

すなわち、Dense 層において全てのバイアスを 0 にして、かつ活性化関数を恒等関数するという特別な場合が Embedding 層といえる。

(では、なぜ Embedding 層を使うのか？)

分散表現の定義そのものを表すのが Embedding 層であり、明示的に層の役割を定めることで、使用性や移

植性・利便性が上がるためと考えます。

また、Embedding 層は使用頻度的にも多いので、毎回 Dense 層を使用して Embedding 層を定義しなおしては大変だからと考えられます。)

・ Embedding 層で得られる分散表現：

まず、Word2Vec は分散表現の中でも CBOW や Skip-gram を使用したときに得られる特別な場合であり、Embedding 層で獲得できるであろう全ての分散表現 = Word2Vec とは限りません。

そのため、Embedding 層を使えば必ず Word2Vec 表現が得られるという理解は間違っています。

Embedding 層が学習後に Word2Vec なる分散表現を表現できるのは、CBOW や Skip-gram を構築して学習させたときのみです。

でも、Embedding 層が使われるのは、自然言語処理の面で使用しやすく、また、Skip-gram ですでに学習した重み行列を移植することも可能であるからといえます。

では、Word2Vec のように CBOW や Skip-gram で得られる分散表現以外の表現は意味をなさないか？といえば、そうではありません。

逆に、まっさらな Embedding 層を毎回使用することで、所望のタスクに特化した分散表現が得られることも多いと考えられます。

・ 埋め込み行列と埋め込みベクトル：

Embedding 層の内部アーキテクチャは、大きく 2 つに分けられ、

1 つ目…入力記号を示す数値を OneHot ベクトルに変換すること、

2 つ目…その OneHot ベクトルをより低次のベクトル空間上に線形写像することです。

Embedding 層は、物理的計測の不可能な記号集合をベクトル空間に写像する分散表現そのものの定義を実現するもので、COW や Skip-gram を使えば単語同士の意味によるベクトル計算が可能な分散表現を得ることができます。

それ以外の分散表現も学習可能で、それぞれのタスクに特化して埋め込みを学習することで、言語のような物理的計測が不可能な記号集合とニューラルネットワークのかけ橋として重要な役割を担っている。

引用サイト：<https://agirobots.com/word2vec-and-embeddinglayer/>

●Section7：Attention Mechanism

時系列データの中身の関連性に対し、重みを付与する手法。

補足(YouTube):<https://youtu.be/bPdyuIebXWM>

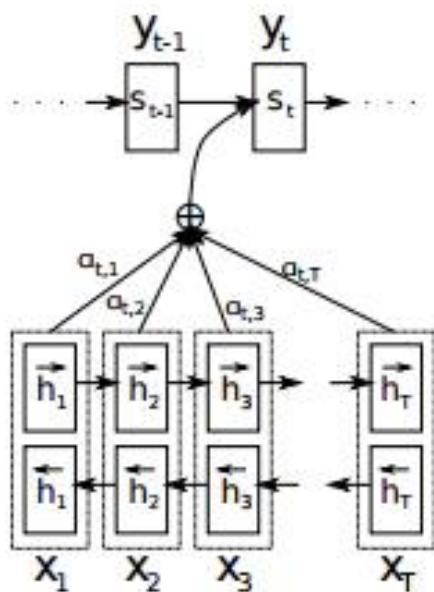
seq2seq の問題は長い文章への対応が難しいことです。具体的には上の例での「意味のようなもの」を表すのが固定次元のベクトルであることに問題があります。つまり、3 単語のとても短い文であっても、50 単語あるとても長い文であっても、その意味のある固定次元ベクトルの中に押し込まなくてはなりません。

そこで、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ような何らかの仕組みが必要になります。Attention Mechanism ではこの問題に対して、「入力と出力のどの単語が関連しているのか」を学習させることで対応します。下図のようにネットワークは翻訳前後の単語の対応関係を学習し、単語列の出力時に対応する入力の単語を引っ張ってくることで長い文書でも翻訳の精度をあげます。

Attention Mechanism 具体例



※「a」については、そもそも関連度が低く、「I」については「私」との関連度が高い。



- 入力側
 - x_j : j 番目の入力単語
 - h_j : j 番目の入力に対応する隠れ層
 - 入力側はバイディレクショナル RNN になっており、隠れ層には順方向のものと逆方向のものがあ
ります。それぞれ矢印の向きで表現されています。
- 出力側
 - y_t : t 番目の出力単語
 - s_t : t 番目の出力単語に対応する隠れ層
- Attention Mechanism
 - α_{ij} : i 番目の単語に対して j 番目の単語が関連している確率もしくは結びつきの強さ

時刻 i の出力の隠れ状態 s_i を計算するときに以下のような処理がはしります。

FNN によって、前時刻の出力の内部状態 s_{i-1} 及び各時刻 j の入力 h_j を入力として、時刻 i の出力に最も関連のありそ
うな入力時刻 j が計算されます (e_{ij}, α_{ij})。そして、その時刻の入力の隠れ層が c_i として s_i への入力の一部になります。

引用サイト : <https://qiita.com/halhorn/items/614f8fe1ec7663e04bea>

確認テスト

seq2seqとHRED、HREDとVHREDの違いを簡潔に述べよ。
(5分)

Seq2seq…一問一答処理しか出来ない。

HRED…文脈の意味を汲み取った変換が可能だが、ありきたりな返答になりがち。

VHRED…HRED に VAE を導入し、HRED より人間らしい表現が可能