深層学習 day3

このレポートでは、深層学習 day1、day2 と同じ用語と記法を用いる。また、全てのベクトルを縦ベクトルと見なす。

1　再帰型ニューラルネットの概念

$\{(\mathbf{x}_i, \mathbf{y}_i)\}_{i=1}^N$ をラベル付けされたデータとし、N をデータの数、$\mathbf{x}_i$ を T 個の m 次元特徴ベクトル $\mathbf{x}_i^1, \ldots, \mathbf{x}_i^T$ の組みとし、$\mathbf{y}_i$ を $\mathbf{x}_i$ のラベルとする。このようなデータを**時系列データ**という。T 個の m 次元特徴ベクトルからなる時系列全体の集合を $S(m,T)$ で表すことにする。**再帰型ニューラルネット**は未知の時系列データ $\mathbf{x} = (\mathbf{x}^1, \ldots, \mathbf{x}^T) \in S(m,T)$ に対して、$\mathbf{x}$ のラベル $\mathbf{y}$ を予測するための深層学習モデルである。$\mathbf{W}^{(in)}$ を $n \times m$ 行列、$\mathbf{W}$ を $n \times n$ 行列、$\mathbf{W}^{(out)}$ を $r \times n$ 行列、$\mathbf{b}$ を n 次元ベクトル、$f : \mathbb{R}^n \to \mathbb{R}^n, g : \mathbb{R}^r \to \mathbb{R}^r$ を関数とする。このとき、$RN(\mathbf{W}^{(in)}, \mathbf{W}, \mathbf{W}^{(out)}, \mathbf{b}, f, g) : S(m,T) \to \mathbb{R}^r$ を以下のように定義する。$\mathbf{x} = (\mathbf{x}^1, \ldots, \mathbf{x}^T) \in S(m,T)$ とする。z_0 を n 次元零ベクトルとし、$1 \leq t \leq T$ に対して、z^t を

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

で定義し、$\mathbf{y}^t = g(\mathbf{W}^{(out)}z^t)$ と定義する。このとき、$RN(\mathbf{W}^{(in)}, \mathbf{W}, \mathbf{W}^{(out)}, \mathbf{b}, f, g)(\mathbf{x}) = \mathbf{y}^T$ と定義する。

例 1.1. (コードは再帰型ニューラルネットの概念.ipynb) 再帰型ニューラルネットを実装する。

```
import numpy as np

class RNN:
    def __init__(self, Wx, Wh, b):
        self.params = [Wx, Wh, b]
        self.grads = [np.zeros_like(Wx), np.zeros_like(Wh), np.zeros_like(b)]
        self.cache = None

    def forward(self, x, h_prev):
        Wx, Wh, b = self.params
        t = np.dot(h_prev, Wh) + np.dot(x, Wx) + b
        h_next = np.tanh(t)

        self.cache = (x, h_prev, h_next)
        return h_next
```

```
def backward(self, dh_next):
    Wx, Wh, b = self.params
    x, h_prev, h_next = self.cache

    dt = dh_next * (1 - h_next ** 2)
    db = np.sum(dt, axis=0)
    dWh = np.dot(h_prev.T, dt)
    dh_prev = np.dot(dt, Wh.T)
    dWx = np.dot(x.T, dt)
    dx = np.dot(dt, Wx.T)

    self.grads[0][...] = dWx
    self.grads[1][...] = dWh
    self.grads[2][...] = db

    return dx, dh_prev
```

2 LSTM

$\{(\mathbf{x}_i, \mathbf{y}_i)\}_{i=1}^N$ をラベル付けされたデータとし、N をデータの数、$\mathbf{x}_i$ を T 個の m 次元特徴ベクトル $\mathbf{x}_i^1, \ldots, \mathbf{x}_i^T$ の組みとし、$\mathbf{y}_i$ を $\mathbf{x}_i$ のラベルとする。再帰型ニューラルネット $RN(\mathbf{W}^{(in)}, \mathbf{W}, \mathbf{W}^{(out)}, \mathbf{b}, f, g)$ を用いて、未知の時系列データ $\mathbf{x} = (\mathbf{x}^1, \ldots, \mathbf{x}^T) \in S(m, T)$ に対して、$\mathbf{x}$ のラベル $\mathbf{y}$ を予測することを試みる。$RN(\mathbf{W}^{(in)}, \mathbf{W}, \mathbf{W}^{(out)}, \mathbf{b}, f, g)$ に誤差逆伝搬法（これを **BPTT**（Backpropagation Through Time）という）を適用すると、多くの勾配が掛け合わされるため、勾配消失あるいは勾配爆発が起こりやすいことが分かる。この問題を解決するために考案されたのが、**LSTM**（Long Short-Term Memory）である。$\mathbf{W}_\mathbf{x}^{(\mathbf{f})}, \mathbf{W}_\mathbf{h}^{(\mathbf{f})}, \mathbf{W}_\mathbf{x}^{(\mathbf{g})}, \mathbf{W}_\mathbf{h}^{(\mathbf{g})}, \mathbf{W}_\mathbf{x}^{(\mathbf{i})}, \mathbf{W}_\mathbf{h}^{(\mathbf{i})}, \mathbf{W}_\mathbf{x}^{(\mathbf{o})}, \mathbf{W}_\mathbf{h}^{(\mathbf{o})}$ を行列とし、$\mathbf{b}^{(\mathbf{f})}, \mathbf{b}^{(\mathbf{g})}, \mathbf{b}^{(\mathbf{i})}, \mathbf{b}^{(\mathbf{o})}$ をベクトルとする。記法の簡略化のため、行列のリスト $\mathbf{W}_\mathbf{x}^{(\mathbf{f})}, \mathbf{W}_\mathbf{h}^{(\mathbf{f})}, \mathbf{W}_\mathbf{x}^{(\mathbf{g})}, \mathbf{W}_\mathbf{h}^{(\mathbf{g})}, \mathbf{W}_\mathbf{x}^{(\mathbf{i})}, \mathbf{W}_\mathbf{h}^{(\mathbf{i})}, \mathbf{W}_\mathbf{x}^{(\mathbf{o})}, \mathbf{W}_\mathbf{h}^{(\mathbf{o})}$ を $\mathbf{W}$ で略記し、ベクトルのリスト $\mathbf{b}^{(\mathbf{f})}, \mathbf{b}^{(\mathbf{g})}, \mathbf{b}^{(\mathbf{i})}, \mathbf{b}^{(\mathbf{o})}$ を $\mathbf{b}$ で略記する。このとき、$LSTM(\mathbf{W}, \mathbf{b}) : S(m, T) \to \mathbb{R}^n$ を以下のように定義する。$\mathbf{x} = (\mathbf{x}^1, \ldots, \mathbf{x}^T) \in S(m, T)$ とする。$\mathbf{h}_0, \mathbf{c}_0$ を零ベクトル、σ をシグモイド関数とし、$1 \leq t \leq T$ に対して、

$$\mathbf{f} = \sigma(\mathbf{W}_\mathbf{x}^{(\mathbf{f})}\mathbf{x}^t + \mathbf{W}_\mathbf{h}^{(\mathbf{f})}\mathbf{h}_{t-1} + \mathbf{b}^{(\mathbf{f})}) \tag{1}$$

$$\mathbf{g} = \tanh(\mathbf{W}_\mathbf{x}^{(\mathbf{g})}\mathbf{x}^t + \mathbf{W}_\mathbf{h}^{(\mathbf{g})}\mathbf{h}_{t-1} + \mathbf{b}^{(\mathbf{g})}) \tag{2}$$

$$\mathbf{i} = \sigma(\mathbf{W}_\mathbf{x}^{(\mathbf{i})}\mathbf{x}^t + \mathbf{W}_\mathbf{i}^{(\mathbf{i})}\mathbf{h}_{t-1} + \mathbf{b}^{(\mathbf{i})}) \tag{3}$$

$$\mathbf{o} = \sigma(\mathbf{W}_\mathbf{x}^{(\mathbf{o})}\mathbf{x}^t + \mathbf{W}_\mathbf{o}^{(\mathbf{o})}\mathbf{h}_{t-1} + \mathbf{b}^{(\mathbf{o})}) \tag{4}$$

$$\mathbf{c}_t = \mathbf{f} \otimes \mathbf{c}_{t-1} + \mathbf{g} \otimes \mathbf{i} \tag{5}$$

$$\mathbf{h}_t = \mathbf{o} \otimes \tanh(\mathbf{c}_t) \tag{6}$$

と定義し（$\otimes$ はアダマール積）、$LSTM(\mathbf{W},\mathbf{b})(\mathbf{x}) = \mathbf{h}_T$ と定義する。上の数式は以下のように理解するとよい。

(1) 忘却ゲート $\mathbf{f}$: $\mathbf{f}$ は忘れる記憶の量を調整する。
(2) 記憶セルへの追加 $\mathbf{g}$: $\mathbf{g}$ は記憶の追加を行う。
(3) 追加する記憶の量の調整 $\mathbf{i}$: $\mathbf{i}$ は $\mathbf{g}$ が追加する記憶の量を調整する。
(4) 記憶セル $\mathbf{c}_t$: $\mathbf{c}_t$ は各時間 t ごとの記憶を格納する。

例 2.1. (コードは LSTM.ipynb) LSTM を実装する。

```
import numpy as np

class LSTM:
    def __init__(self, Wx, Wh, b):
        '''
        Parameters
        ----------
        Wx: 入力‘x‘用の重みパラーメタ（4 つ分の重みをまとめる）
        Wh: 隠れ状態‘h‘用の重みパラメータ（4 つ分の重みをまとめる）
        b: バイアス（4 つ分のバイアスをまとめる）
        '''
        self.params = [Wx, Wh, b]
        self.grads = [np.zeros_like(Wx), np.zeros_like(Wh), np.zeros_like(b)]
        self.cache = None

    def forward(self, x, h_prev, c_prev):
        Wx, Wh, b = self.params
        N, H = h_prev.shape

        A = np.dot(x, Wx) + np.dot(h_prev, Wh) + b

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

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

```
        c_next = f * c_prev + g * i
        h_next = o * np.tanh(c_next)

        self.cache = (x, h_prev, c_prev, i, f, g, o, c_next)
        return h_next, c_next

    def backward(self, dh_next, dc_next):
        Wx, Wh, b = self.params
        x, h_prev, c_prev, i, f, g, o, c_next = self.cache

        tanh_c_next = np.tanh(c_next)

        ds = dc_next + (dh_next * o) * (1 - tanh_c_next ** 2)

        dc_prev = ds * f

        di = ds * g
        df = ds * c_prev
        do = dh_next * tanh_c_next
        dg = ds * i

        di *= i * (1 - i)
        df *= f * (1 - f)
        do *= o * (1 - o)
        dg *= (1 - g ** 2)

        dA = np.hstack((df, dg, di, do))

        dWh = np.dot(h_prev.T, dA)
        dWx = np.dot(x.T, dA)
        db = dA.sum(axis=0)

        self.grads[0][...] = dWx
        self.grads[1][...] = dWh
        self.grads[2][...] = db

        dx = np.dot(dA, Wx.T)
        dh_prev = np.dot(dA, Wh.T)

        return dx, dh_prev, dc_prev
```

3 GRU

LSTM は良い深層学習モデルであるが、パラメーターが多く、計算には時間がかかる。そこで最近では、LSTM に代わる「ゲート付き RNN」が数多く提案されている。ここでは、**GRU** と呼ばれる有名で実績のあるゲート付き RNN を紹介する。$\mathbf{W}_{\mathbf{x}}^{(\mathbf{z})}, \mathbf{W}_{\mathbf{h}}^{(\mathbf{z})}, \mathbf{W}_{\mathbf{x}}^{(\mathbf{r})}, \mathbf{W}_{\mathbf{h}}^{(\mathbf{r})}, \mathbf{W}_{\mathbf{x}}, \mathbf{W}_{\mathbf{h}}$ を行列とし、$\mathbf{b}^{(\mathbf{z})}, \mathbf{b}^{(\mathbf{r})}, \mathbf{b}$ をベクトルとする。記法の簡略化のため、行列のリスト $\mathbf{W}_{\mathbf{x}}^{(\mathbf{z})}, \mathbf{W}_{\mathbf{h}}^{(\mathbf{z})}, \mathbf{W}_{\mathbf{x}}^{(\mathbf{r})}, \mathbf{W}_{\mathbf{h}}^{(\mathbf{r})}, \mathbf{W}_{\mathbf{x}}, \mathbf{W}_{\mathbf{h}}$ を $\mathbf{W}$ で略記し、ベクトルのリスト $\mathbf{b}^{(\mathbf{z})}, \mathbf{b}^{(\mathbf{r})}, \mathbf{b}$ を $\mathbf{b}$ で略記する。このとき、$GRU(\mathbf{W}, \mathbf{b}) : S(m, T) \to \mathbb{R}^n$ を以下のように定義する。$\mathbf{x} = (\mathbf{x}^1, \ldots, \mathbf{x}^T) \in S(m, T)$ とする。$\mathbf{h}_0$ を零ベクトル、σ をシグモイド関数とし、$1 \leq t \leq T$ に対して、

$$\mathbf{z} = \sigma(\mathbf{W}_{\mathbf{x}}^{(\mathbf{z})}\mathbf{x}^t + \mathbf{W}_{\mathbf{h}}^{(\mathbf{z})}\mathbf{h}_{t-1} + \mathbf{b}^{(\mathbf{z})}) \tag{7}$$

$$\mathbf{r} = \sigma(\mathbf{W}_{\mathbf{x}}^{(\mathbf{r})}\mathbf{x}^t + \mathbf{W}_{\mathbf{h}}^{(\mathbf{r})}\mathbf{h}_{t-1} + \mathbf{b}^{(\mathbf{r})}) \tag{8}$$

$$\tilde{\mathbf{h}} = \tanh(\mathbf{W}_{\mathbf{x}}\mathbf{x}^t + \mathbf{W}_{\mathbf{h}}(\mathbf{r} \otimes \mathbf{h}_{t-1}) + \mathbf{b}) \tag{9}$$

$$\mathbf{h}_t = (1 - \mathbf{z}) \otimes \mathbf{h}_{t-1} + \mathbf{z} \otimes \tilde{\mathbf{h}} \tag{10}$$

と定義し（$\otimes$ はアダマール積）、$GRU(\mathbf{W}, \mathbf{b})(\mathbf{x}) = \mathbf{h}_T$ と定義する。上の数式は以下のように理解するとよい。

(1) リセットゲート $\mathbf{r}$: $\mathbf{r}$ は過去の隠れ状態をどれだけ無視するのかを決定する。
(2) アップデートゲート $\mathbf{z}$: $\mathbf{z}$ は隠れ状態を更新する。

例 **3.1.** (コードは GRU.ipynb) GRU を実装する。

```
import numpy as np


class GRU:
    def __init__(self, Wx, Wh, b):
        '''
        Parameters
        ----------
        Wx: 入力‘x‘用の重みパラーメタ（3 つ分の重みをまとめる）
        Wh: 隠れ状態‘h‘用の重みパラメータ（3 つ分の重みをまとめる）
        b: バイアス（3 つ分のバイアスをまとめる）
        '''
        self.params = [Wx, Wh, b]
        self.grads = [np.zeros_like(Wx), np.zeros_like(Wh), np.zeros_like(b)]
        self.cache = None

    def forward(self, x, h_prev):
        Wx, Wh, b = self.params
        H = Wh.shape[0]
        Wxz, Wxr, Wxh = Wx[:, :H], Wx[:, H:2 * H], Wx[:, 2 * H:]
        Whz, Whr, Whh = Wh[:, :H], Wh[:, H:2 * H], Wh[:, 2 * H:]
```

```
        bz, br, bh = b[:H], b[H:2 * H], b[2 * H:]

        z = sigmoid(np.dot(x, Wxz) + np.dot(h_prev, Whz) + bz)
        r = sigmoid(np.dot(x, Wxr) + np.dot(h_prev, Whr) + br)
        h_hat = np.tanh(np.dot(x, Wxh) + np.dot(r*h_prev, Whh) + bh)
        h_next = (1-z) * h_prev + z * h_hat

        self.cache = (x, h_prev, z, r, h_hat)

        return h_next

    def backward(self, dh_next):
        Wx, Wh, b = self.params
        H = Wh.shape[0]
        Wxz, Wxr, Wxh = Wx[:, :H], Wx[:, H:2 * H], Wx[:, 2 * H:]
        Whz, Whr, Whh = Wh[:, :H], Wh[:, H:2 * H], Wh[:, 2 * H:]
        x, h_prev, z, r, h_hat = self.cache

        dh_hat =dh_next * z
        dh_prev = dh_next * (1-z)

        # tanh
        dt = dh_hat * (1 - h_hat ** 2)
        dbh = np.sum(dt, axis=0)
        dWhh = np.dot((r * h_prev).T, dt)
        dhr = np.dot(dt, Whh.T)
        dWxh = np.dot(x.T, dt)
        dx = np.dot(dt, Wxh.T)
        dh_prev += r * dhr

        # update gate(z)
        dz = dh_next * h_hat - dh_next * h_prev
        dt = dz * z * (1-z)
        dbz = np.sum(dt, axis=0)
        dWhz = np.dot(h_prev.T, dt)
        dh_prev += np.dot(dt, Whz.T)
        dWxz = np.dot(x.T, dt)
        dx += np.dot(dt, Wxz.T)

        # rest gate(r)
```

```
        dr = dhr * h_prev
        dt = dr * r * (1-r)
        dbr = np.sum(dt, axis=0)
        dWhr = np.dot(h_prev.T, dt)
        dh_prev += np.dot(dt, Whr.T)
        dWxr = np.dot(x.T, dt)
        dx += np.dot(dt, Wxr.T)

        self.dWx = np.hstack((dWxz, dWxr, dWxh))
        self.dWh = np.hstack((dWhz, dWhr, dWhh))
        self.db = np.hstack((dbz, dbr, dbh))

        self.grads[0][...] = self.dWx
        self.grads[1][...] = self.dWh
        self.grads[2][...] = self.db

        return dx, dh_prev
```

4 双方向 RNN

$LSTM(\mathbf{W},\mathbf{b})$ を一つ用意する。$S'(m,T) = \{(\mathbf{x}^T,\dots,\mathbf{x}^1) \mid \mathbf{x} \in S(m,T)\}$ とおき、もう一つの $LSTM(\mathbf{W}',\mathbf{b}')$ を用意する。$\mathbf{V}$ を行列とし、$\mathbf{c}$ をベクトルとする。このとき、$BLSTM(\mathbf{W},\mathbf{W}',\mathbf{b},\mathbf{b}',\mathbf{V},\mathbf{c})$: $S(m,T) \times S'(m,T) \to \mathbb{R}^n$ を $BLSTM(\mathbf{W},\mathbf{W}',\mathbf{b},\mathbf{b}',\mathbf{V},\mathbf{c})(\mathbf{x},\mathbf{x}') = \mathbf{V}(\mathbf{h}_T,\mathbf{h}'_T)^T + \mathbf{c}$ で定義する。$BLSTM(\mathbf{W},\mathbf{W}',\mathbf{b},\mathbf{b}',\mathbf{V})$ を**双方向 RNN** という。上記の方法は GRU にもそのまま適用できる。

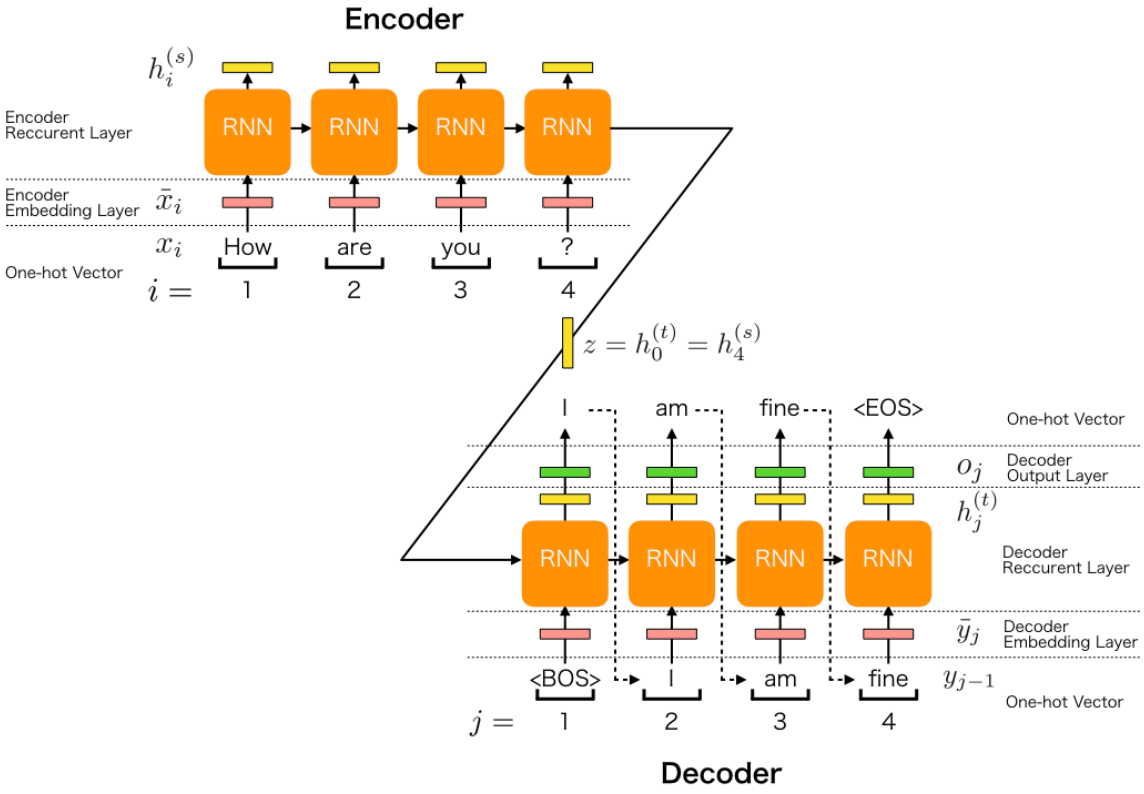
例 **4.1.** (コードは双方向 RNN.ipynb) 双方向 RNN を実装する。

```
import numpy as np

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

    def forward(self, xs):
        o1 = self.forward_lstm.forward(xs)
        o2 = self.backward_lstm.forward(xs[:, ::-1])
```

```
        o2 = o2[:, ::-1]

        out = np.concatenate((o1, o2), axis=2)
        return out

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

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

5 Seq2Seq

Seq2Seq は時系列データを別の時系列データへ変換するモデルである。Seq2Seq は二つのモジュール Encoder と Decoder を持つ。



Encoder と Decoder はいずれも再帰型ニューラルネットである。Encoder は時系列データをエンコードし、翻訳するために必要な情報を固定長のベクトルへ格納し、Decoder はそのベクトルを用いて、時系列データを生成する。例えば、Encoder は「吾輩は猫である」という文章をエンコードして、翻訳するために必要な情報を固定長のベクトルへ格納し、Decoder はそのベクトルを用いて、”I am a cat”という文章を生成する。

例 **5.1.** (コードは Seq2Seq.ipynb) Seq2Seq を実装する。

```
import numpy as np

import sys
sys.path.append("/content/drive/MyDrive/Colab Notebooks")
from common.time_layers import *
from common.base_model import BaseModel

class Encoder:
    def __init__(self, vocab_size, wordvec_size, hidden_size):
        V, D, H = vocab_size, wordvec_size, hidden_size
        rn = np.random.randn

        embed_W = (rn(V, D) / 100).astype('f')
```

```
        lstm_Wx = (rn(D, 4 * H) / np.sqrt(D)).astype('f')
        lstm_Wh = (rn(H, 4 * H) / np.sqrt(H)).astype('f')
        lstm_b = np.zeros(4 * H).astype('f')

        self.embed = TimeEmbedding(embed_W)
        self.lstm = TimeLSTM(lstm_Wx, lstm_Wh, lstm_b, stateful=False)

        self.params = self.embed.params + self.lstm.params
        self.grads = self.embed.grads + self.lstm.grads
        self.hs = None

    def forward(self, xs):
        xs = self.embed.forward(xs)
        hs = self.lstm.forward(xs)
        self.hs = hs
        return hs[:, -1, :]

    def backward(self, dh):
        dhs = np.zeros_like(self.hs)
        dhs[:, -1, :] = dh

        dout = self.lstm.backward(dhs)
        dout = self.embed.backward(dout)
        return dout


class Decoder:
    def __init__(self, vocab_size, wordvec_size, hidden_size):
        V, D, H = vocab_size, wordvec_size, hidden_size
        rn = np.random.randn

        embed_W = (rn(V, D) / 100).astype('f')
        lstm_Wx = (rn(D, 4 * H) / np.sqrt(D)).astype('f')
        lstm_Wh = (rn(H, 4 * H) / np.sqrt(H)).astype('f')
        lstm_b = np.zeros(4 * H).astype('f')
        affine_W = (rn(H, V) / np.sqrt(H)).astype('f')
        affine_b = np.zeros(V).astype('f')

        self.embed = TimeEmbedding(embed_W)
        self.lstm = TimeLSTM(lstm_Wx, lstm_Wh, lstm_b, stateful=True)
```

```
        self.affine = TimeAffine(affine_W, affine_b)

        self.params, self.grads = [], []
        for layer in (self.embed, self.lstm, self.affine):
            self.params += layer.params
            self.grads += layer.grads

    def forward(self, xs, h):
        self.lstm.set_state(h)

        out = self.embed.forward(xs)
        out = self.lstm.forward(out)
        score = self.affine.forward(out)
        return score

    def backward(self, dscore):
        dout = self.affine.backward(dscore)
        dout = self.lstm.backward(dout)
        dout = self.embed.backward(dout)
        dh = self.lstm.dh
        return dh

    def generate(self, h, start_id, sample_size):
        sampled = []
        sample_id = start_id
        self.lstm.set_state(h)

        for _ in range(sample_size):
            x = np.array(sample_id).reshape((1, 1))
            out = self.embed.forward(x)
            out = self.lstm.forward(out)
            score = self.affine.forward(out)

            sample_id = np.argmax(score.flatten())
            sampled.append(int(sample_id))

        return sampled


class Seq2seq(BaseModel):
```

```
def __init__(self, vocab_size, wordvec_size, hidden_size):
    V, D, H = vocab_size, wordvec_size, hidden_size
    self.encoder = Encoder(V, D, H)
    self.decoder = Decoder(V, D, H)
    self.softmax = TimeSoftmaxWithLoss()

    self.params = self.encoder.params + self.decoder.params
    self.grads = self.encoder.grads + self.decoder.grads

def forward(self, xs, ts):
    decoder_xs, decoder_ts = ts[:, :-1], ts[:, 1:]

    h = self.encoder.forward(xs)
    score = self.decoder.forward(decoder_xs, h)
    loss = self.softmax.forward(score, decoder_ts)
    return loss

def backward(self, dout=1):
    dout = self.softmax.backward(dout)
    dh = self.decoder.backward(dout)
    dout = self.encoder.backward(dh)
    return dout

def generate(self, xs, start_id, sample_size):
    h = self.encoder.forward(xs)
    sampled = self.decoder.generate(h, start_id, sample_size)
    return sampled
```

6 Word2vec

ニューラルネットを用いて、単語を処理したいとする。この時、問題となるのは、ニューラルネットは、”you”や”say”などの単語をそのままでは処理できないということである。ニューラルネットで単語を処理するには、それを「固定長のベクトル」に変換する必要がある。その方法の一つが **one-hot 表現**である。one-hot 表現とは、ベクトルの要素の中で一つが 1 で、残りは全て 0 であるようなベクトルのことである。例として、”You say goodbye and I say hello.”という文章について考える。このとき、単語を one-hot 表現に変換するには、語彙数分（この例では 7）の要素を持つベクトルを用意して、単語の位置に該当する箇所を 1 に、残りの箇所を全て 0 に設定すればよい。例えば、”you”は $(1, 0, 0, 0, 0, 0, 0)$ に変換される。**Word2vec** は、one-hot 表現をより小さなベクトルへと変換するモデルである。この章では、Word2vec で使用されるニューラルネットワークの一つである **CBOW**（continuous bag-of-words）を紹介する。$\mathbf{W}_{in}$ を 7×3 行列、$\mathbf{W}_{out}$ を 3×7

行列とし、$CBOW(\mathbf{W}_{in}, \mathbf{W}_{out}) : \mathbb{R}^7 \times \mathbb{R}^7 \to \mathbb{R}$ を

$$CBOW(\mathbf{W}_{in}, \mathbf{W}_{out})(\mathbf{x}_1, \mathbf{x}_2) = Softmax(\frac{1}{2}(\mathbf{x}_1\mathbf{W}_{in} + \mathbf{x}_2\mathbf{W}_{in})\mathbf{W}_{out})$$

で定義する。次に、第 i 成分が 1 であり、残りの成分が全て 0 であるベクトルを $\mathbf{e}_i$ で表すことにする。データ集合 $D = \{((\mathbf{e}_i, \mathbf{e}_{i+2}), \mathbf{e}_{i+1})\}_{i=1,\ldots,5}$ を用意し、$CBOW(\mathbf{W}_{in}, \mathbf{W}_{out})$ を D 上で訓練する。訓練の結果得られた $\mathbf{W}_{in}$ は one-hot 表現をより小さなベクトルへと正しく変換することができる。上記の方法は、任意の長さの文章へ一般化することができ、$\mathbf{W}_{in}, \mathbf{W}_{out}$ の型も自由に変えることができる。

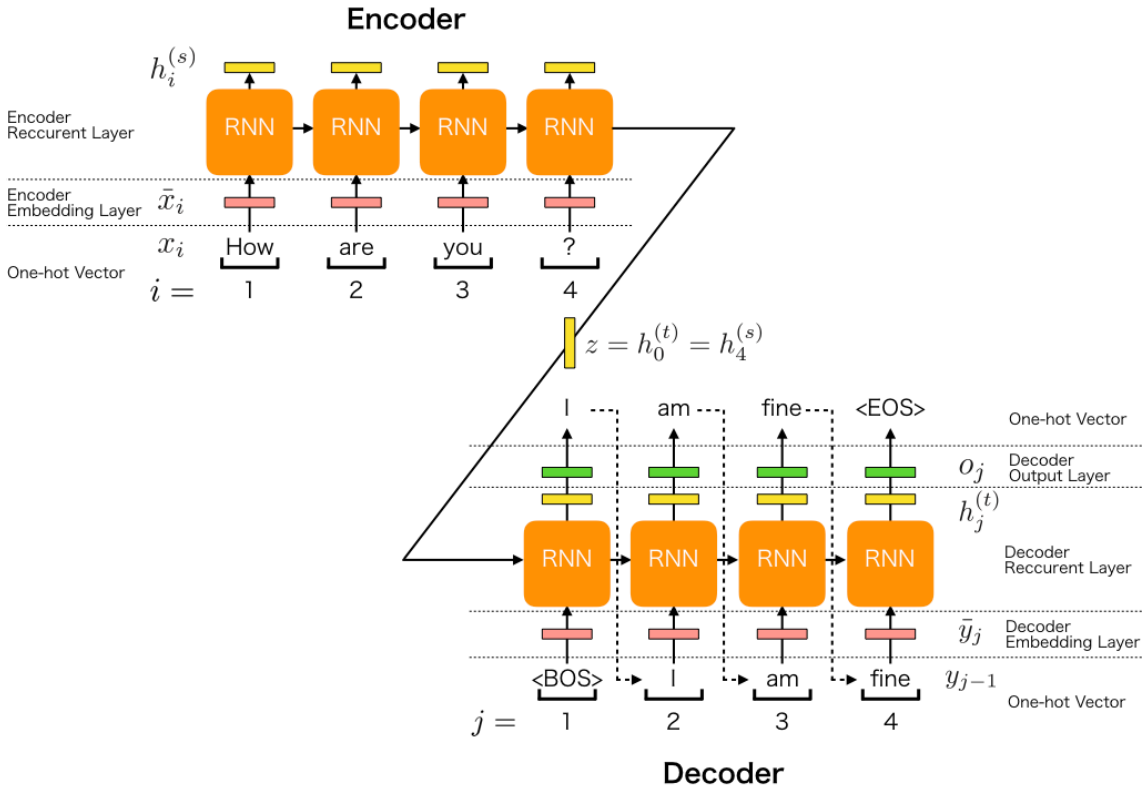
例 6.1. (コードは Word2vec.ipynb) CBOW を実装する。

```
import numpy as np
import sys
sys.path.append("/content/drive/MyDrive/Colab Notebooks")
from common.layers import MatMul, SoftmaxWithLoss

class SimpleCBOW:
    def __init__(self, vocab_size, hidden_size):
        V, H = vocab_size, hidden_size

        # 重みの初期化
        W_in = 0.01 * np.random.randn(V, H).astype('f')
        W_out = 0.01 * np.random.randn(H, V).astype('f')

        # レイヤの生成
        self.in_layer0 = MatMul(W_in)
        self.in_layer1 = MatMul(W_in)
        self.out_layer = MatMul(W_out)
        self.loss_layer = SoftmaxWithLoss()

        # すべての重みと勾配をリストにまとめる
        layers = [self.in_layer0, self.in_layer1, self.out_layer]
        self.params, self.grads = [], []
        for layer in layers:
            self.params += layer.params
            self.grads += layer.grads

        # メンバ変数に単語の分散表現を設定
        self.word_vecs = W_in

    def forward(self, contexts, target):
        h0 = self.in_layer0.forward(contexts[:, 0])
```

```
        h1 = self.in_layer1.forward(contexts[:, 1])
        h = (h0 + h1) * 0.5
        score = self.out_layer.forward(h)
        loss = self.loss_layer.forward(score, target)
        return loss

    def backward(self, dout=1):
        ds = self.loss_layer.backward(dout)
        da = self.out_layer.backward(ds)
        da *= 0.5
        self.in_layer1.backward(da)
        self.in_layer0.backward(da)
        return None
```

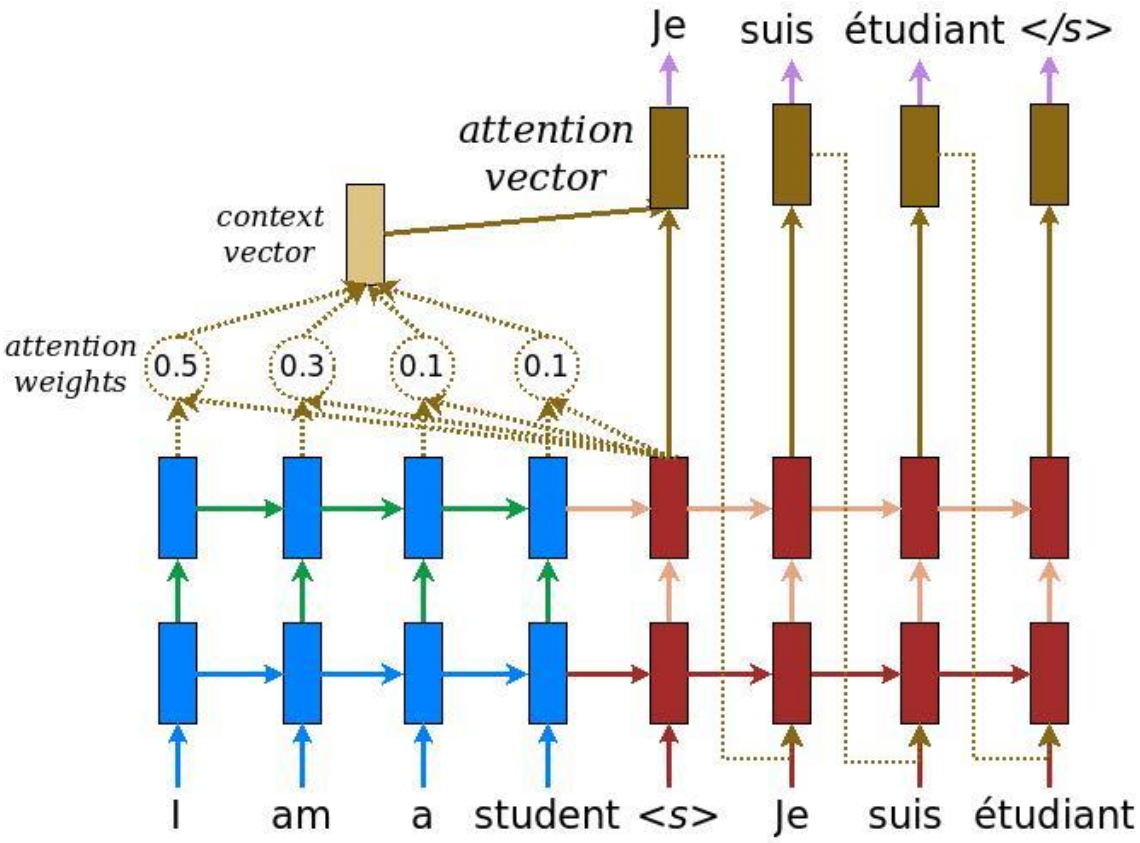
7 Attention Mechanism

Seq2Seq モデル



を考える。Encoder は時系列データをエンコードし、翻訳するために必要な情報を固定長のベクトルへ格納

し、Decoder はそのベクトルを用いて、時系列データを生成する。ここで問題となるのは、翻訳するために必要な情報が固定長のベクトルへ格納されるということである。何故ならば、入力文の長さに関わらず、情報が格納されるベクトルの長さが固定されているため、本来必要な情報がベクトルに格納されない恐れがあるからである。**Attention Mechanism** はこの問題を解決する方法である。Attention Mechanism では、Decoder は RNN の各時刻の出力において、Encoder の各時刻の出力に attention weight を掛け合わせて足し合わせることで、context vector を出力する。



その後、context vector は上の層へと伝搬され、softmax 関数を用いて確率のベクトルが出力され、最終的に確率の最も高い単語が出力される。学習によって、attention weight のパラメーターは更新され、モデルは、Encoder の出力の中で最も重要な部分に注目するようになる。

参考文献

[1] Andriy Burkov. (2019). The hundred-page machine learning book.

[2] Francois Chollet. (2018). Deep learning with python. Manning Publications Co.

[3] Marc Peter Deisenroth., A. Aldo Faisal., Cheng Soon Ong. (2020). Mathematics for machine learning. Cambridge University Press.

[4] Aurëlien Gëron. (2019). Hands-on machine learning with Scikit-Learn, Keras & TensorFlow. 2nd Edition. Oreilly.

[5] 小縣信也., 斎藤翔汰., 溝口聡., 若杉一幸. (2021). ディープラーニング E 資格エンジニア問題集. インプ

レス.
[6] 岡谷貴之. (2015). 深層学習. 講談社.
[7] Sebastian Raschka., Vahid Mirjalili. (2019). Python machine learning. Third Edition. Packt.
[8] 斎藤康毅. (2016). ゼロから作る deep learning. オライリージャパン.
[9] 斎藤康毅. (2018). ゼロから作る deep learning 2. オライリージャパン.
[10] 瀧雅人. (2017). これならわかる深層学習. 講談社.