深層学習 day3

秋葉洋哉

2024 年 7 月 29 日

1　day3：再帰型ニューラルネットワーク (RNN)

1.1　概要

RNN は、時系列データを扱う際に有効なニューラルネットワークである。時系列データとは、時間的順序を追って一定間隔毎に観測され、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のことである。例えば、音声データ、テキストデータ、株価データなどが挙げられる。(テキストデータが時系列データとなるのは、単語の並び順によって文章が構築されるためである。) RNN は、時系列データの特徴を捉えるために、中間層の出力が次の時刻の入力として用いられる。

1.2　RNN における順伝搬

以降では、RNN の基本的な構造について説明するため、系列 $\mathbf{x_1}, \mathbf{x_2}, \cdots$ を RNN に順番に入力したときに、対応する出力 $\mathbf{y_1}, \mathbf{y_2}, \cdots$ を得ることを考える。まず、入力層・中間層・出力層の各ユニットのインデックスを i, j, k とする。そして、時刻 t における中間層ユニットへの出力を $\mathbf{u^t} = (u_j^t)$ と $\mathbf{z^t} = (z_j^t)$、出力層のユニットへの出力を $\mathbf{v^t} = (v_k^t)$ と $\mathbf{y_t} = (y_k^t)$ とする。入力層と中間層間の重みを $W^{(\mathrm{in})} = W_{ji}^{(\mathrm{in})}$、中間層から中間層への重みを $W = (W_{jj'})$、中間層と出力層間の重みを $W^{(\mathrm{out})} = W_{kj}^{(\mathrm{out})}$ とする。

この時、時刻 t における中間層 j ユニットへの出力 u_j^t は、以下のように表される。

$$u_j^t = \sum_i W_{ji}^{(\mathrm{in})} x_i^t + \sum_{j'} W_{jj'} z_{j'}^{t-1} \tag{1}$$

これは、時刻 t における中間層の各ユニットへの入力が、入力層からの入力と、前の時刻 $t-1$ における中間層の出力からの入力の和であることを示している。ここでは、$W_{j0}^{(\mathrm{in})}$ をバイアスとして与えている。以上の導出過程の概略図を図 1 に示す。

活性化関数を $f(u_j^t)$ とすると、中間層の出力 z_j^t は、

$$z_j^t = f(u_j^t) \tag{2}$$

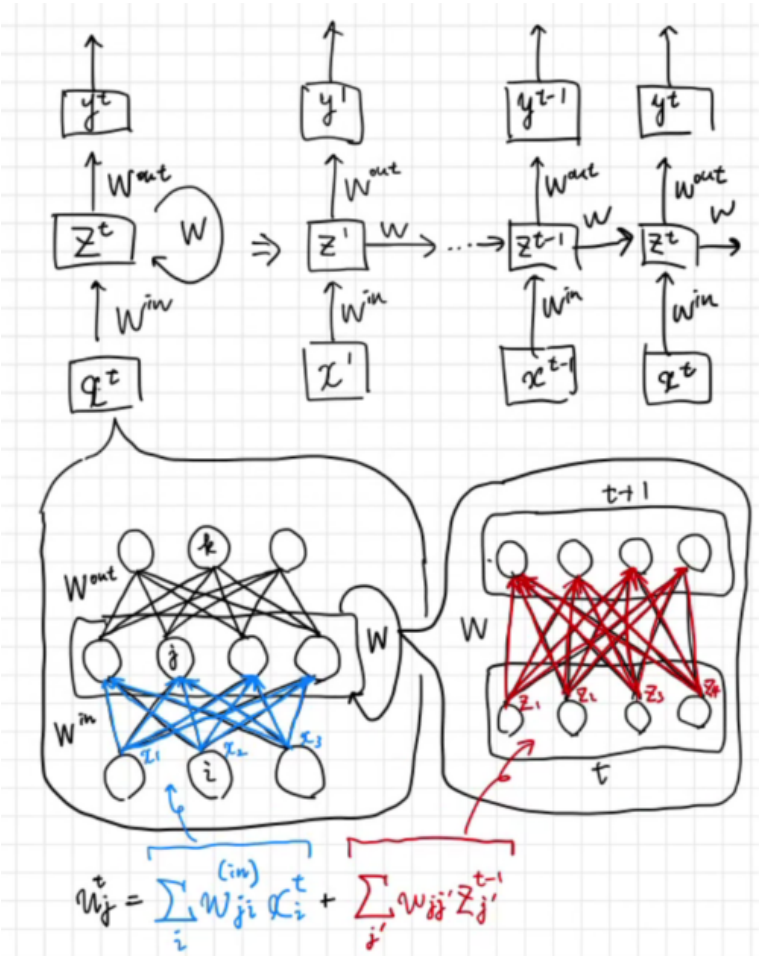

図 1: RNN における順伝搬の概略図

となる。これらをまとめると、時刻 t における中間層の入力 $\mathbf{u^t}$、出力 $\mathbf{z^t}$、出力層の入力 $\mathbf{v^t}$、出力 $\mathbf{y^t}$ は、

$$\mathbf{u^t} = \mathbf{W}^{(\mathrm{in})}\mathbf{x^t} + \mathbf{W}\mathbf{z^{t-1}} \tag{3}$$

$$\mathbf{z^t} = f(\mathbf{W}^{(\mathrm{in})}\mathbf{x^t} + \mathbf{W}\mathbf{z^{t-1}}) \tag{4}$$

$$\mathbf{v^t} = \mathbf{W}^{(\mathrm{out})}\mathbf{z^t} + c \tag{5}$$

$$\mathbf{y^t} = g(\mathbf{W}^{(\mathrm{out})}\mathbf{z^t} + c) \tag{6}$$

となる。ただし、c は出力層のバイアスである。また、g は出力層の活性化関数である。

1.3 RNN における誤差逆伝播法

RNN で誤差逆伝播法を用いて重みを更新する際には、RTRL(Real Time Recurrent Learning) や BPTT(Back Propagation Through Time) といった手法が用いられる。前者はメモリ効率が良く、後者は計算効率が良いという特徴がある。ここでは BPTT について説明する。

RNN では、時刻 t の中間層 j の入力 u_j^t を以下で表した。

$$u_j^t = \sum_i W_{ji}^{(\mathrm{in})} x_i^t + \sum_{j'} W_{jj'} z_{j'}^{t-1} \tag{7}$$

これは、時刻 t における入力層からの入力 x_i^t と、時刻 $t-1$ における中間層の出力 $z_{j'}^{t-1}$ からの入力に各重み成分を掛け合わせたの和としてあらわされている。

ここで、ニューラルネットワークの逆伝播では以下のように表せた。

$$\frac{\partial E}{\partial w_{ji}} = \sum_t \frac{\partial E}{\partial u_j^t} \frac{\partial u_j^t}{\partial w_{ji}} \tag{8}$$

$$= \sum_t \delta_j^t z_i^t \tag{9}$$

ただし、δ_j^t は中間層 j の誤差であり、以下のように表される。

$$\delta_j = \frac{\partial E}{\partial u_j} = \frac{\partial E}{\partial z_j} \frac{\partial z_j}{\partial u_j} \tag{10}$$

RNN における δ_j^t は、時刻 t における出力層 k の入力 v_k^t と、時刻 $t+1$ における中間層 j' の入力 $u_{j'}^{t+1}$ に依存するため、以下のように表される。

$$\delta_j^t = \frac{\partial E}{\partial u_j^t} \tag{11}$$

$$= \sum_k \frac{\partial E}{\partial v_k^t} \frac{\partial v_k^t}{\partial u_j^t} + \sum_{j'} \frac{\partial E}{\partial u_{j'}^{t+1}} \frac{\partial u_{j'}^{t+1}}{\partial u_j^t} \tag{12}$$

まず、前半部分は、以下のように展開できる。

$$\sum_k \frac{\partial E}{\partial v_k^t} \frac{\partial v_k^t}{\partial u_j^t} = \sum_k \delta_k^t \frac{\partial v_k^t}{\partial u_j^t} \tag{13}$$

$$= \sum_k \delta_k^t \frac{\partial}{\partial u_j^t} \left(\sum_j W_{kj}^{(\mathrm{out})} z_j^t + c \right) \tag{14}$$

$$= \sum_k \delta_k^t W_{kj}^{(\mathrm{out})} \frac{\partial z_j^t}{\partial u_j^t} \tag{15}$$

$$= \sum_k \delta_k^t W_{kj}^{(\mathrm{out})} \frac{\partial g(u_j^t)}{\partial u_j^t} \tag{16}$$

$$= \sum_k \delta_k^t W_{kj}^{(\mathrm{out})} g'(u_j^t) \tag{17}$$

そして、後半部分は、以下のように展開できる。

$$\sum_{j'} \frac{\partial E}{\partial u_{j'}^{t+1}} \frac{\partial u_{j'}^{t+1}}{\partial u_j^t} = \sum_{j'} \delta_{j'}^{t+1} \frac{\partial u_{j'}^{t+1}}{\partial u_j^t} \tag{18}$$

$$= \sum_{j'} \delta_{j'}^{t+1} \frac{\partial}{\partial u_j^t} \left(\sum_i W_{j'i}^{(\mathrm{in})} x_i^{t+1} + \sum_j W_{j'j} z_j^t \right) \tag{19}$$

$$= \sum_{j'} \delta_{j'}^{t+1} \frac{\partial}{\partial u_j^t} \left(\sum_i W_{j'i}^{(\mathrm{in})} x_i^{t+1} + \sum_j W_{j'j} f(u_j^t) \right) \tag{20}$$

$$= \sum_{j'} \delta_{j'}^{t+1} W_{j'j} f'(u_j^t) \tag{21}$$

よって、δ_j^t は以下のように表される。

$$\delta_j^t = \sum_k \delta_k^t W_{kj}^{(\mathrm{out})} g'(u_j^t) + \sum_{j'} \delta_{j'}^{t+1} W_{j'j} f'(u_j^t) \tag{22}$$

この結果を用いて、誤差関数を各重みで微分した値を導出できる。

$$\frac{\partial E}{\partial W_{ji}^{(\mathrm{in})}} = \sum_{t=1}^{T} \frac{\partial E}{\partial u_j^t} \frac{\partial u_j^t}{\partial W_{ji}^{(\mathrm{in})}} \tag{23}$$

$$= \sum_{t=1}^{T} \delta_j^t x_i^t \tag{24}$$

$$\frac{\partial E}{\partial W_{jj'}} = \sum_{t=1}^{T} \frac{\partial E}{\partial u_j^t} \frac{\partial u_j^t}{\partial W_{jj'}} \tag{25}$$

$$= \sum_{t=1}^{T} \delta_j^t z_{j'}^{t-1} \tag{26}$$

$$\frac{\partial E}{\partial W_{kj}^{(\mathrm{out})}} = \sum_{t=1}^{T} \frac{\partial E}{\partial v_k^t} \frac{\partial v_k^t}{\partial W_{kj}^{(\mathrm{out})}} \tag{27}$$

$$= \sum_{t=1}^{T} \delta_k^t z_j^t \tag{28}$$

確認テスト

Q: 図 2 について、y_1 を、$x, z_0, z_1, W_{(\mathrm{in})}, W, W_{(\mathrm{out})}$ を用いて数式で表せ。
A: 以下導出の通り。ただし、g は活性化関数、c はバイアスである。

$$y_1 = \sum W_{(\mathrm{out})} z_1 + c \tag{29}$$

$$= \sum W_{(\mathrm{out})} g(\sum W z_0 + \sum W^{(\mathrm{in})} x + c) + c \tag{30}$$

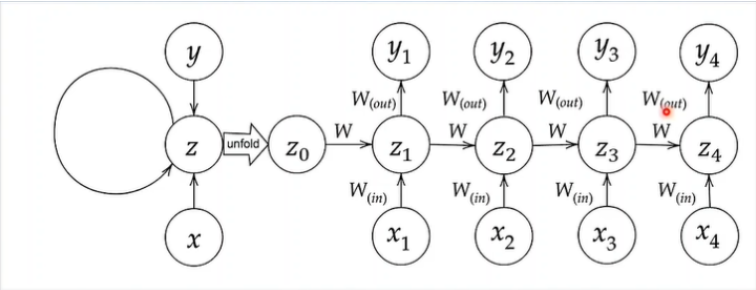

図 2

1.4 LSTM

RNN は、時刻 t における一つのニューラルネットワークを、$t = 0$ から $t = T$ まで繰り返し適用するモデルである。一つのニューラルネットワークでさえ、逆伝播において勾配消失問題について考える必要があるが、RNN では、時刻が進むにしたがってより多くのニューラルネットワークを重ねていくため、逆伝播における勾配消失問題がより顕著になる。また、逆に、時刻が進むにしたがって逆伝播における勾配が爆発する問題も発生する。

確認テスト

Q: RNN や深いモデルでは勾配の消失や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という手法が使われることがある。具体的には勾配のノルムが閾値を超えたら、勾配のノルムを閾値に正規化するというものである。この関数を完成させなさい。

A:

```
def gradient_clipping(grad, threshold):
  norm = np.linalg.norm(grad) # 勾配のノルムを計算
  rate = threshold / norm
  if rate < 1:
    return grad * rate
  return grad
```

この問題を解決するために、LSTM(Long Short-Term Memory) が提案された。図 3 に LSTM の構造を示す。

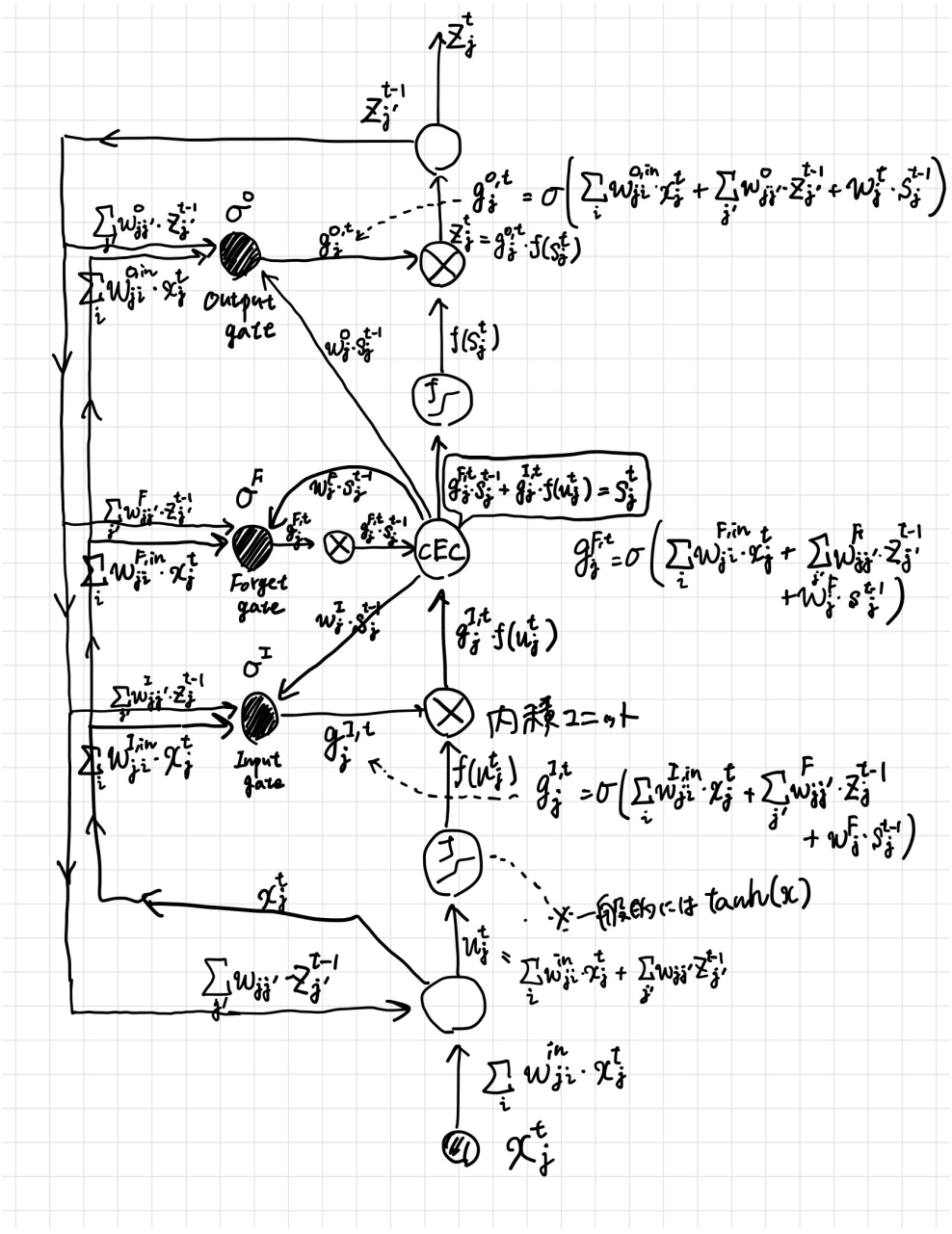

図 3: LSTM の構造

図 3 によれば、入力ゲート・出力ゲート・忘却ゲートの 3 つのゲートが存在し、入力・出力ゲートは層と層の間に設けられ、忘却ゲートは CEC(Constant Error Carrousel) と呼ばれる、過去の情報を保持するためのユニットとループ状態で接続されている。

1.4.1 CEC

CEC は記憶を保持するためのユニットであり、学習機能有さない。学習の機能は各種ゲートによって制御され、CEC に何を覚えさせるか、何を忘れさせるか、何を出力するかを制御する。

1.4.2 入力ゲート

入力ゲートでは、時刻 t の入力 x_j^t と、時刻 $t-1$ の出力 z_j^{t-1}、そして、時刻 $t-1$ の CEC の情報 s_j^{t-1} の 3 つを入力として、新しい情報を CEC に追加する。x_j^t, z_j^{t-1}, s_j^{t-1} に対してはそれぞれの別の重みが掛けられ、情報を重視する割合を調整する。この重みが学習によって最適化される。

1.4.3 出力ゲート

出力ゲートでは、時刻 t の入力 x_j^t と、時刻 $t-1$ の出力 z_j^{t-1}、そして、時刻 tCEC の情報 s_j^t の 3 つを入力として、最終的に出力する値の調整を行う。こちらにも割合を調整する重みが掛けられ、学習によって最適化される。

1.4.4 忘却ゲート

忘却ゲートでは、時刻 t の入力 x_j^t と、時刻 $t-1$ の出力 z_j^{t-1}、そして、時刻 $t-1$CEC の情報 s_j^{t-1} の 3 つを入力として、CEC の情報をどれだけ忘れるかを調整する。もし、忘却ゲートが無かったら、過去の情報が無限に保持されることになる。このことは過去の情報がいらなくなった場合に、削除することができないことを意味する。忘却ゲートによって、過去の情報を必要な時に保持し、不要な時に削除することができる。これによって、勾配消失や勾配爆発を防ぐことができる。

確認テスト

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無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: 忘却ゲートが作用する。忘却ゲートによって過去の情報の有無を判断しているため、「とても」という単語の影響が少ないことを学習することができる。

Q: LSTM の順伝搬を行うプログラムを書け。

A: 以下に示す。

```
def lstm(x, prev_h, prev_c, W, U, b):
  """
  x: inputs, (batch_size, input_size)
  prev_h: previous output at time t-1, (batch_size, hidden_size)
  prev_c: previous cell state at time t-1, (batch_size, hidden_size)
  W: upwrad weights, (4*state_size, input_size)
  U: lateral weights, (4*state_size, hidden_size)
  b: biases, (4*state_size,)
  """

  # セルへの入力やゲートをまとめて計算し分割する
  a = np.dot(x, W.T) + np.dot(prev_h, U.T) + b
  a, ai, af, ao = np.split(a, 4, axis=1)

  # 各ゲートの計算
  ig = sigmoid(ai)
  fg = sigmoid(af)
  og = sigmoid(ao)

  # セルの値の計算
  c = fg * prev_c + ig * np.tanh(a) # 要素同士の積を求めるため、np.dot にはならないことに注意
  h = og * np.tanh(c)

  return h, c
```

1.5 GRU(Gated Recurrent Unit)

LSTM では、パラメータ数が多く、計算コストが高いという問題があった。この問題を解決するために、GRU が提案された。GRU では、LSTM の入力ゲートと忘却ゲートを「更新ゲート」として統合し、また、出力ゲートを廃止することで、パラメータ数を削減している。また、「リセットゲート」を導入することで、過去の情報をどれだけ保持するかを調整することができる。CEC への入力値 $\mathbf{u^t}$, CEC の保持値 $\mathbf{s^t}$, 出力 $\mathbf{z^t}$ は以下のように表される。

$$\mathbf{u^t} = \mathbf{W}^{(\mathrm{in})}\mathbf{x^t} + \mathbf{W}\mathbf{g}^{R,t} \odot z^{t-1} \tag{31}$$
$$\mathbf{s^t} = \mathbf{g}^{U,t} \odot \mathbf{s^t} + (1 - \mathbf{g}^{U,t}) \odot f(\mathbf{u^t}) \tag{32}$$
$$\mathbf{z^t} = \mathbf{s^t} \tag{33}$$

ここで、$\odot$ は要素積を表す。また、$\mathbf{g}^{R,t}$ はリセットゲート、$\mathbf{g}^{U,t}$ は更新ゲートの出力値で、

$$\mathbf{g}^{R,t} = \sigma(\mathbf{W^R x^t} + \mathbf{U^R z^{t-1}}) \tag{34}$$
$$\mathbf{g}^{U,t} = \sigma(\mathbf{W^U x^t} + \mathbf{U^U z^{t-1}}) \tag{35}$$

と表される。σ はシグモイド関数である。式 (31) の右辺第二項の $\mathbf{g}^{R,t}$ が無かった時、更新ゲートの値が 0 であった場合に、過去の状態が常に同じ割合 ($\mathbf{W z^{t-1}}$) で現在に伝播されてしまうが、リセットゲートを導入することで、忘却ゲートのように過去の情報をどれだけ保持するかを調整することができる。

1.6 双方向 RNN

双方向 RNN は、時系列データを前方向と後方向の 2 つの RNN で処理し、それぞれの出力を結合することで、より正確な予測を行うことができる。文章の推敲や、機械翻訳などで利用される。双方向 RNN の Python コードを以下に示す。

```
import numpy as np

def bidriectinoal_rnn(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, hidden_size)
"""

  xs_f = np.zeros_like(xs)
  xs_b = np.zeros_like(xs)
  for i , x in enumerate
    xs_f[i] = x
    xs_b[i] = x[::-1]
  hs_f = forward_rnn(xs_f, W_f, U_f)
  hs_b = forward_rnn(xs_b, W_b, U_b)
  hs = [np.concatenate([h_f, h_b[::-1]], axis=1) for h_f, h_b in zip(hs_f, hs_b)]
  ys = hs.dot(V.T)
  return ys
```

ここで、numpy の concatenate は、配列を結合する関数である。具体的には、axis=0 の場合、縦方向に結合し、axis=1 の場合、横方向に結合する。そのあとの zip 関数は、複数のリストを同時にループ処理するための関数である。こうすることで、過去と未来の情報を無くすこと無く、演算することができる。

1.7 自己回帰モデル

自己回帰モデルは、時系列データを入力し、その出力を次の時刻の入力として再帰的に処理するモデルである。RNN は、自己回帰モデルの一種である。

$$\mathbf{x}^t = \phi_0 + \sum_{i=t-\rho}^{t-1} \phi_i \mathbf{x}^i + \epsilon^t \tag{36}$$

ここで、ϕ_0 は定数項、ϕ_i は係数、ϵ^t はノイズである。この式は、時刻 t における入力 $\mathbf{x}^t$ が、過去の入力 $\mathbf{x}^{t-1}, \mathbf{x}^{t-2}, \cdots$ に依存していることを示している。

1.8 Seq2Seq

Seq2Seq は、時系列データを入力として、長さの異なる時系列データを出力するモデルである。機械翻訳や対話モデルなどで利用される。Seq2Seq の構造は、図 4 のようになる。

Seq2Seq は、エンコーダとデコーダから構成される。エンコーダは、入力データを受け取り、中間層の出力をデコーダに渡す。デコーダは、エンコーダから受け取った中間層の出力を元に、出力データを生成する。エンコーダとデコーダは、LSTM や GRU などの RNN を用いて構築される。

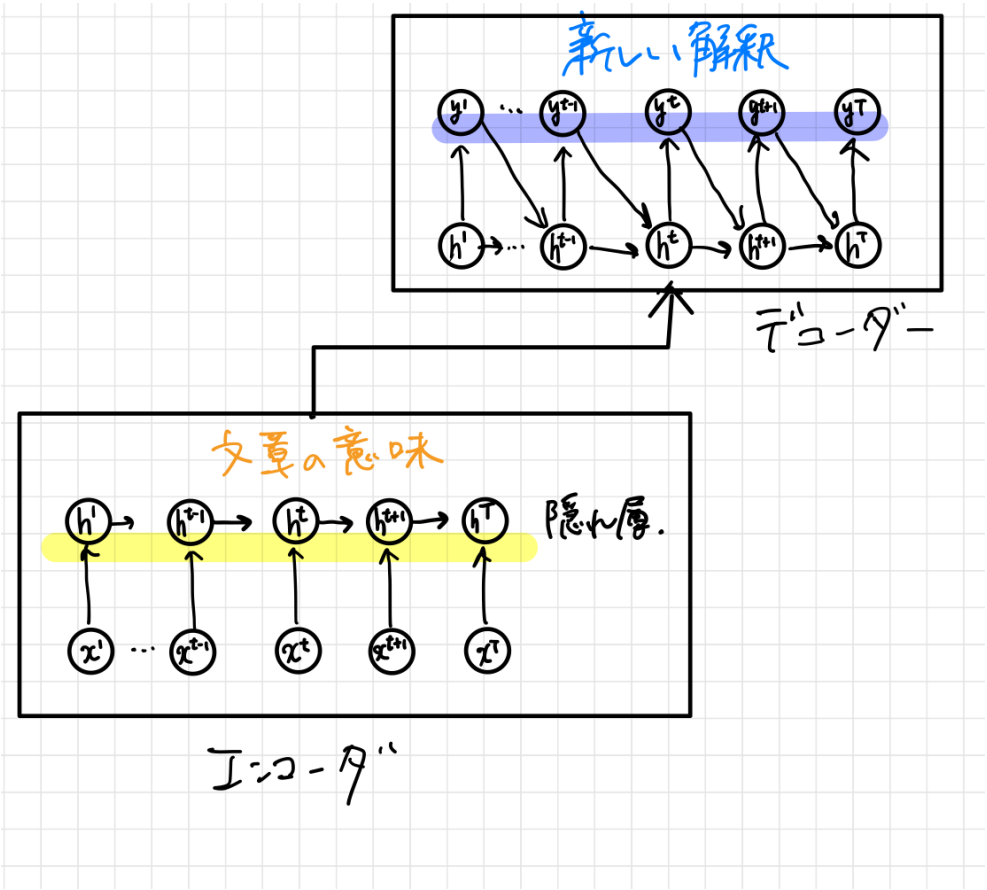

図 4: Seq2Seq の構造

1.8.1 Encoder RNN

エンコーダでは、主に、Taking, Embedding, RNN の 3 つの処理を行う。Taking では、ユーザーがインプットしたデータを単語毎に分割する。

Embedding では、分割した単語をベクトルに変換する。例えば英語なら、英単語のうち代表的なものを K 語選び、これを 1-of-K 符号化する。これを Embedding Matrix と呼ぶ。その後、入力に対して各単語に対応するベクトルを生成する。このベクトルは、似た単語の意味は似た配列となるように機械学習によって生成される。

RNN では、Embedding で生成されたベクトルを入力として受け取る。例えば、文章の途中を隠して単語を予測したり、後に続く文章を生成したりすることができる。

1.8.2 Decoder RNN

Encoder の内部状態を入力して、出力を生成する。この時、用いる Embedding Matrix は、必ずしも Encoder で利用した K 語の単語である必要はなく、異なる言語の Embedding Matrix を用いることもできる。こうすることで、異なる言語間での翻訳が可能となる。

1.8.3 HRED

HRED(Hierarchical Recurrent Encoder-Decoder) は、Seq2Seq の拡張モデルである。Seq2Seq では、文脈を考慮した応答ができないという問題があったが、HRED では、文脈を考慮した応答が可能となる。Seq2Seq におけるエンコーダの中間層の出力を、さらに過去のエンコーダ出力と RNN と繋いで処理する。この、会話コンテキスト全体を表すベクトルを Context RNN と呼ぶ。

ただし、HRED は同じコンテキストが与えられた場合に、同様の出力になってしまったり、短い返答を学んでしまう問題があった。この問題を解決するために、VHRED(Variational Hierarchical Recurrent Encoder-Decoder) が提案された。VHRED は、HRED に VAE(Variational Auto-Encoder) を組み合わせたモデルである。VAE は、データの生成過程を確率モデルとして表現し、データの生成過程を学習するモデルである。VHRED は、HRED の中間層の出力を VAE の入力として、文脈を考慮した応答を生成する。

1.9 VAE(Variational Auto-Encoder)

1.9.1 **オートエンコーダ**

オートエンコーダーは、教師無し学習を用いて、入力と出力が同じ値になるように、エンコーダとデコーダを作成していく、教師無し学習の一つである。入力と潜在変数の間にはエンコーダが、出力と潜在変数の間にはデコーダが存在する。エンコーダによって入力データを次元削減し、特徴を抽出する。デコーダによって、次元削減されたデータを元の次元に戻す。この時、入力と出力が同じになるように学習を行う。

1.10 word2vec

1.10.1 **概要**

word2vec は、単語をベクトルに変換する手法である。word2vec には、CBOW(Continuous Bag of Words) と Skip-gram の 2 つの手法がある。CBOW は、周囲の単語から中央の単語を予測するモデルである。Skip-gram は、中央の単語から周囲の単語を予測するモデルである。word2vec は、単語の意味をベクトル化することができるため、自然言語処理の分野で広く利用されている。

1.10.2 Skip-gram

Skip-gram では、ある単語を入力したとき、その周辺にどのような単語が表れやすいかを予測する。例えば以下のような例文があったとする。

> I want to eat an apple every day.

eat を入力としたとき、周辺には apple や orange といった単語が表れる確率が高く予測し、tank や network といった単語が表れる確率が低いと予測するのが、Skip-gram の手法である。この出現確率を、ボキャブラリ内のすべてに対して計算することで、各単語のベクトルを生成する。このベクトルは、単語の意味を表すベクトルであり、単語ベクトルと呼ばれる。

Skip-gram は教師あり学習である。入力として、eat のような単語を与え、正解データとして apple, orange などの周辺の単語を与える。学習が進むと、入力単語に対して周辺の単語が出現する確率が高くなるようになる。

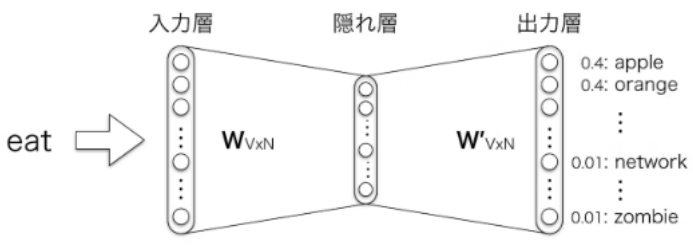

図 5: Skip-gram では周辺の単語の出現確率を予測する。(https://qiita.com/Hironsan/items/11b388575a058dc8a46a より引用)

■入力層 Skip-Gram の入力層では、ボキャブラリ内の単語を 1-of-V 符号化する。例えば、ボキャブラリが I, want, to, eat, an, apple, every, day の 8 単語で構成されている場合、eat を入力としたとき、入力層は以下のようになる。

$$\begin{pmatrix} 0 \\ 0 \\ 0 \\ 1 \\ 0 \\ 0 \\ 0 \\ 0 \end{pmatrix} \tag{37}$$

■中間層 中間層では、入力層の単語をベクトルに変換する。このベクトルは、各単語の意味を表すベクトルであり、単語ベクトルと呼ぶ。例えば、入力層のベクトルが 10000 次元 (10000 単語のボキャブラリ)、中間層のレイヤー数を 300 とすると、各単語ベクトルは 300 次元のベクトルとなる。この単語ベクトルは、ある単語一つに対する、他の 9999 単語の出現確率を予測する重み行列である。

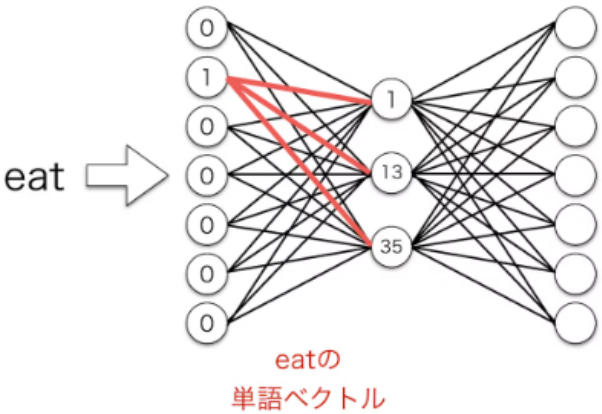

図 6: この例では、eat という単語だけを入力で与えた時に抽出される重みは 1, 13, 35 となる。(https://qiita.com/Hironsan/items/11b388575a058dc8a46a より引用)

■出力層 出力層では、中間層の 1×300 のベクトルを、出力層との間の単語ベクトルとの内積を計算し、出力層の単語の出現確率を計算する。出力層に入力された値は、ソフトマックス関数を通して、確率に変換される。

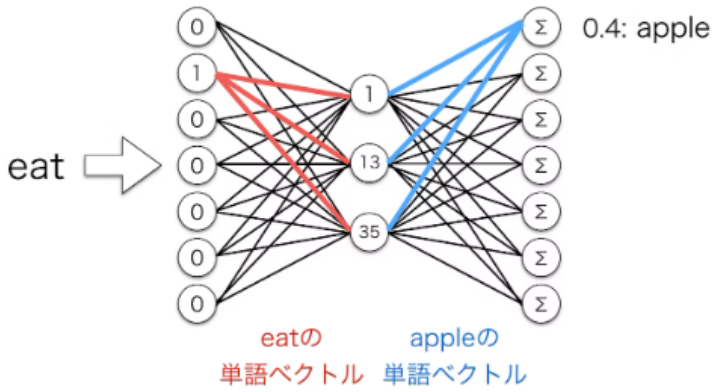

図 7: eat を与えた時に apple を予測する場合に使われる単語ベクトル (赤・青のルート) (https://qiita.com/Hironsan/items/11b388575a058dc8a46a より引用)

1.11 Attention Mechanism (注意機構)

Attention Mechanism は、Seq2Seq の問題点である、長い文章を処理する際に、過去の情報が失われてしまう問題を解決するために提案された。Attention Mechanism は、入力データの各単語に重みを付け、重要な単語に大きな重みを付けることで、過去の情報を保持することができる。Attention Mechanism にはいくつかの種類があり、その中でも、自己注意機構 (Self-Attention Mechanism) は、先ほどの Skip-gram と同様に、単語のベクトルを生成する手法であり、Transformer と呼ばれるモデルに用いられる。

1.12 Spoken Degit

Spoken Degit は、音声データを入力として、0 から 9 までの数字を出力するモデルである。音声データは、MFCC(Mel-Frequency Cepstral Coefficients) と呼ばれる特徴量に変換され、RNN に入力される。RNN は、音声データを時系列データとして処理し、出力として 0 から 9 までの数字を出力する。

TensorFlow で Spoken Degit を実装する場合、

```
dataset_train, dataset_valid, dataset_test = tfds.load(
  name='spoken_digit',
  split=['train[:70%]', 'validation[70%:85%]', 'test[85%:]'],
  shuffle_files=True
)
```

のように用意する。上述の split オプションでは、訓練データを 70%、検証データを 15%、テストデータを 15% に分割している。データの形式は dict 形式で、audio, audio/filename, label の 3 つのキーを持つ。audio は音声データ、audio/filename は音声データのファイル名、label は音声データのラベルである。イテレータを作成して一件取り出す場合は、

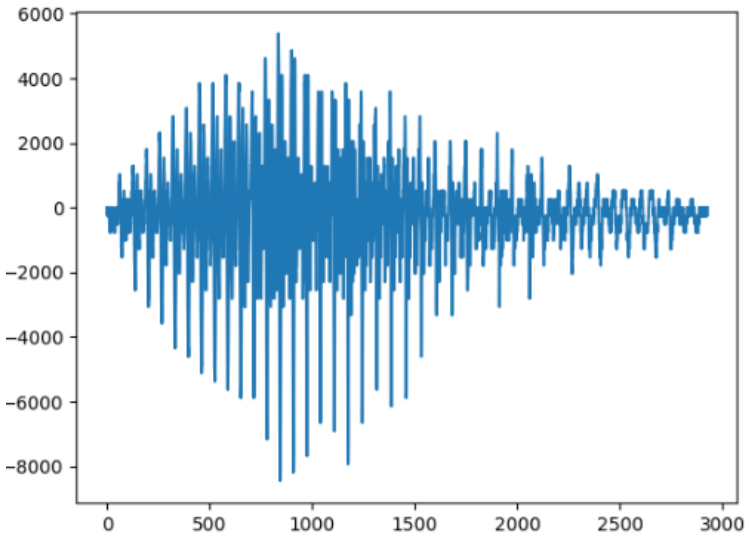

図 8: Spoken Degit で取得したある音声データ波形

```
original_iter = iter(dataset_train)
1 next(original_iter)
plt.plot(next(original_iter)['audio'])
```

とすることで、音声データ波形を表示することができる。ミリ秒程度の音声データであるため、波形が短いことがわかる。

TensorFlow には map 関数が用意されており、データセットに対して関数を適用することができる。例えば、

```
def normalize(example):
  audio = example['audio']
  audio = audio / tf.reduce_max(audio)
  return {'audio': audio, 'label': example['label']}

dataset_train = dataset_train.map(normalize)
```

のように、normalize 関数を定義し、map 関数を用いてデータセットに適用することができる。この場合、音声データを正規化している。

map 関数を用いて、教師ラベルと音声データをペアで用意したものを用いて、TensorFlow の Keras API を使って LSTM で 0 から 9 までの数字を予測するモデルを構築する場合、以下の手順を経る。

1. データセットの準備 (上述のようにデータセットを用意する)
2. モデルの構築 (model = tf.keras.models.Sequential())
3. レイヤーの追加 (model.add(layers.Input), model.add(layers.LSTM), model.add(layers.Dense))
4. モデルのコンパイル (model.compile())

5. モデルの学習 (model.fit())
6. モデルの評価 (model.evaluate())

評価には、テストデータを正しく予測できているか目で確認してみることも大切である。その際には、

```
predictions = model.predict(dataset_test)
print(predictions[0])
print('予測: ', np.argmax(predictions[0]))
print('正解: ', dataset_test[1][0])
```

とすると、予測と正解が表示できる。

1.13 Data Augmentation

Data Augmentation は、データの水増しを行う手法である。データの水増しは、データセットのサイズを増やすことで、過学習を防ぐ効果がある。Data Augmentation の手法としては、Horizontal Flip (水平方向の反転処理), Vertical Flip (垂直方向の反転処理), Random Crop (ランダムに切り取る処理), Random Rotation (ランダムに回転させる処理), Contrast (コントラスト), Brightness (明るさ), Hue (色相) などがある。

また、TensorFlow には、Random Erasing(ランダムにマスクする処理), Mixup(2 つの画像を混ぜ合わせる処理), CutMix(画像の一部を切り取り、貼り付ける処理) などの Data Augmentation の手法が用意されている。

これらの手法を組み合わせて用いることで、合計で 100 倍程度のデータセットを用意することができる。これによって、過学習を防ぐことができる。

■参考文献

1. 岡谷貴之/深層学習 改訂第 2 版 [機械学習プロフェッショナルシリーズ]/ 講談社サイエンティフィク/ 2022-01-17
2. RNN 入門 PART1~3/@_livesomewhere1090 https://www.youtube.com/@_livesomewhere1090
3. 絵で理解する Word2vec の仕組み https://qiita.com/Hironsan/items/11b388575a058dc8a46a
4. TensorFlow, Keras の基本的な使い方（モデル構築・訓練・評価・予測）https://note.nkmk.me/python-tensorflow-keras-basics/