深層学習前編（day3）レポート

2024/6/23

1　Section1: 再帰型ニューラルネットワークの概念

再帰型ニューラルネットワーク（RNN）とは、時系列データに対応可能にしたニューラルネットワークである。図 1 に RNN の全体像を示す。図 1 のように、RNN では、その時刻での入力層からの入力だけでなく、前の時刻での中間層の出力を考慮して、その時刻での中間層の出力を決定する。時刻 t での中間層の出力 z^t と出力層への出力 y^t は以下の式 1,2 で表される。ただし、f,g は活性化関数、b,c はバイアスである。

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

$$y^t = g(W_{(out)}z^t + c) \tag{2}$$

パラメータの更新には、誤差逆伝搬法を RNN で使えるように改良した BPTT を用いる。

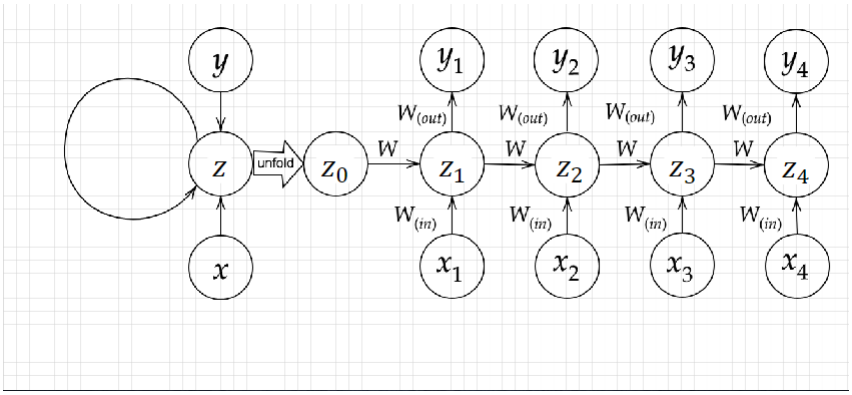


図 1: RNN の全体像

1.1　実装演習

1.1.1　3_1_simple_RNN.ipynb

2 つのバイナリ値の加算結果がどうなるかを学習させた結果を図 2 に示す。ここで、横軸は学習回数、縦軸は学習誤差である。また、中間層への出力、出力層への出力

に用いる活性化関数はシグモイド関数である。学習回数 4000 回程度でほぼ 0 付近に収束しており、学習が進んでいることが分かる。一方で、活性化関数に ReLU 関数を用いた場合を図 3 に示す。ReLU 関数の方が勾配消失問題が起きにくいためか、より早く学習が進んでいるが、重みの値を調べたところ、1.33637312e+06 のように、非常に大きな or 小さな値になっていたため、勾配爆発が起きている可能性がある。

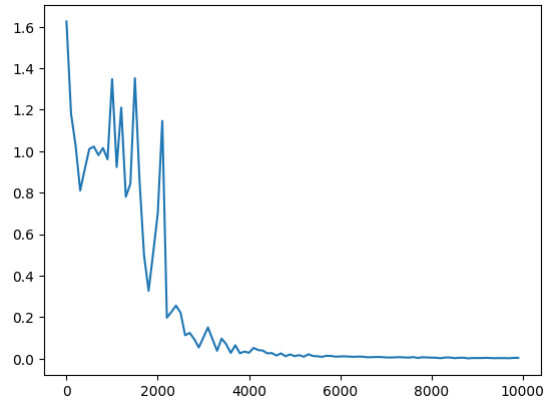


図 2: バイナリ加算の RNN での学習結果（シグモイド関数）

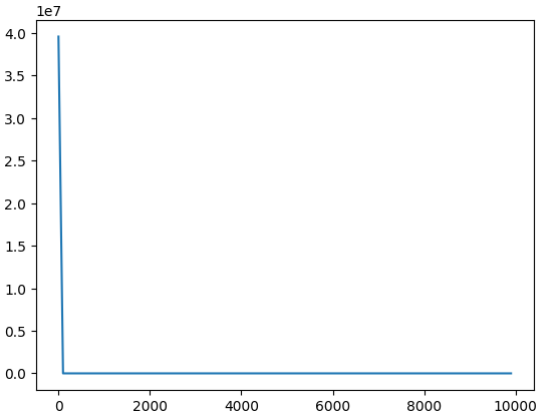


図 3: バイナリ加算の RNN での学習結果（ReLU 関数）

1.2 確認テスト

1.2.1 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。

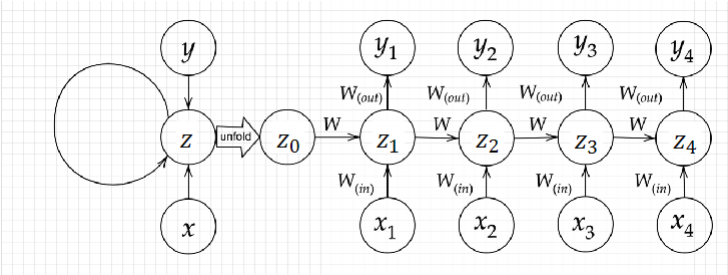
前回データの中間層から今回データの中間層にかけられる重み

1.2.2 連鎖律の原理を使い、dz/dx を求めよ。

$$z = t^2$$
$$t = x + y$$

$$\frac{\mathrm{d}z}{\mathrm{d}x} = \frac{\mathrm{d}z}{\mathrm{d}t}\frac{\mathrm{d}t}{\mathrm{d}x} = 2t = 2(x+y)$$

1.2.3 下図の y1 を x・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。



バイアスを b,c とする。

$$z_1 = f(W_{in}x_1 + Wz_0 + b)$$
$$y_1 = g(W_{out}z_1 + c)$$

2 Section2: LSTM

RNN の課題として、長い時系列を学習すると、時系列を遡るほど勾配が消失していく勾配消失問題や、逆に勾配が指数関数的に上昇する勾配爆発が起きやすいことが挙げられる。そこで、RNN の構造自体を変えて勾配消失問題や勾配爆発に対応したモデルが LSTM である。LSTM の全体像を図 4 に示す。LSTM では、RNN に以下の機構が加えられている。

- CEC
  勾配が 1 となるように値を調整する。過去の時系列や勾配情報を蓄積する。ただし、勾配が 1 になるため、学習能力を持たない。

- 入力ゲート
  重み行列 W,U を学習することで、現在時刻と過去時刻のデータをどのくらいの割合で中間層への出力に用いるかを学習する。
- 出力ゲート
  重み行列 W,U を学習することで、現在時刻と過去時刻のデータをどのくらいの割合で出力層への出力に用いるかを学習する。
- 忘却ゲート
  CEC が持つ過去の情報をどれくらい忘却させるかを学習する。

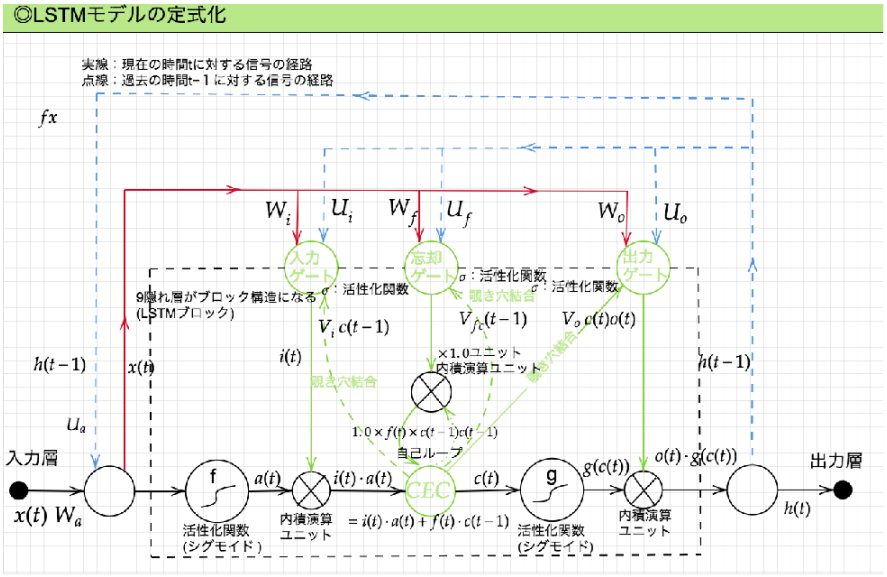


図 4: LSTM の全体像

2.1 確認テスト

2.1.1 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

(2)

2.1.2 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…。」

忘却ゲート

3 Section3: GRU

LSTM の課題として、パラメータ数が多く、計算負荷が高くなることが挙げられる。そこで、パラメータを大幅に削減し、計算負荷を抑えた RNN が GRU である。GRU の全体像を図 5 に示す。GRU では、CEC の役割を持つ機構がないため、その分計算負荷を抑えることに成功している。

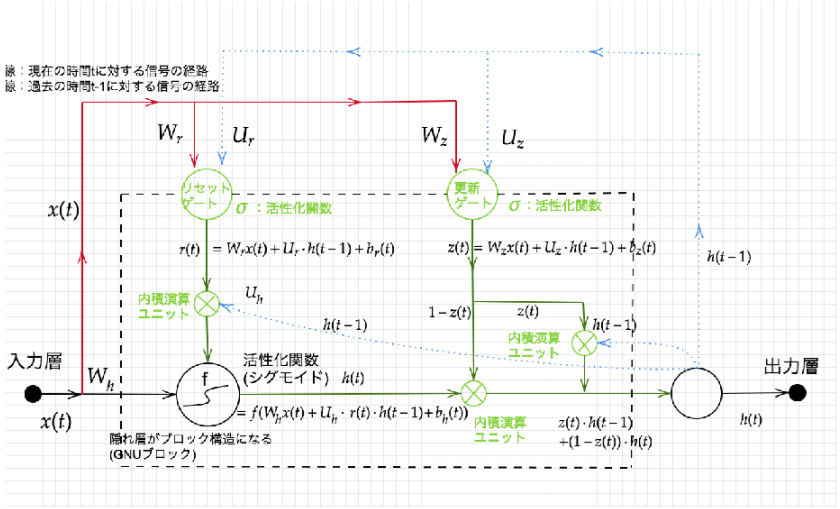


図 5: GRU の全体像

3.1 実装演習

3.1.1 3_4_spoken_digit.ipynb

spoken_digit データを用いて、単純 RNN、GRU それぞれで学習させた結果のキャプチャを図 6,7 に示す。単純 RNN、GRU それぞれの accuracy は 0.1034 と 0.1074 で差がない一方、学習時間は 131 秒と 10 秒で、GRU の方が早いことが分かる。これは、GRU が計算負荷を抑えた RNN であることが理由である。

```
model_2.compile(
    loss=tf.keras.losses.SparseCategoricalCrossentropy(),
    optimizer=tf.keras.optimizers.Adam(),
    metrics=['accuracy']
)

model_2.fit(
    dataset_prep_train,
    validation_data=dataset_prep_valid,
)
```

```
219/219 [==============================] - 131s 596ms/step - loss: 2.4004 - accuracy: 0.1034 - val_loss: 2.3508 - val_accuracy: 0.1200
<tensorflow.python.keras.callbacks.History at 0x7fd0bc08f410>
```

図 6: 単純 RNN で学習させた結果

```
model_3.compile(
    loss=tf.keras.losses.SparseCategoricalCrossentropy(),
    optimizer=tf.keras.optimizers.Adam(),
    metrics=['accuracy']
)

model_3.fit(
    dataset_prep_train,
    validation_data=dataset_prep_valid,
)
```

```
219/219 [==============================] - 10s 38ms/step - loss: 2.3616 - accuracy: 0.1074 - val_loss: 2.3092 - val_accuracy: 0.1307
<tensorflow.python.keras.callbacks.History at 0x7fd07a8246d0>
```

図 7: GRU で学習させた結果

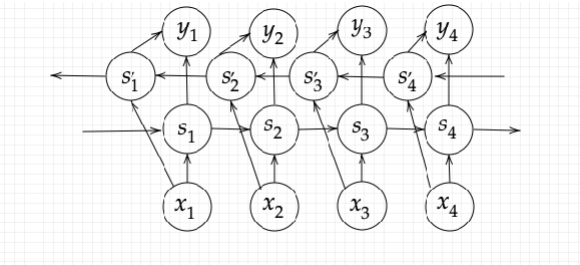


図 8: 双方向 RNN の全体像

3.2 確認テスト

3.2.1 LSTM と CEC が抱える課題について、それぞれ簡潔に述べよ。

LSTM：パラメータ数が多いため、計算量が多くなる。
CEC：勾配が 1 になるようにするため、記憶能力はあるが学習能力がない。

3.2.2 LSTM と GRU の違いを簡潔に述べよ。

LSTM では、入力ゲート・出力ゲート・忘却ゲート・CEC がある。一方で、GRU は CEC の役割をもつものがない。LSTM はパラメータが多く、GRU は LSTM に比べてパラメータが少ないため、計算量も少ない。

4 Section4: 双方向 RNN

双方向 RNN とは、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ある。双方向 RNN の全体像を図 8 に示す。現在時刻の出力層への出力には、RNN で用いられている過去の情報を考慮した中間層出力（s）に加えて、未来の情報を考慮した中間層出力（s'）が用いられている。双方向 RNN の実用例として、文章の推敲や機械翻訳などが挙げられる。

4.1 実装演習

4.1.1 3_4_spoken_digit.ipynb

spoken_digit データを用いて、双方向 RNN で学習させた結果のキャプチャを図 9 に示す。前述の、単純 RNN（図 6）と GRU（図 7）の結果と比べると、実行時間は 15 秒で、GRU ほどではないが単純 RNN より早いことが分かる。一方で accuracy は 0.1406 で、他の手法より最も良く、このデータに対しては双方向 RNN は有効なモデルであることが分かる。だが、学習に未来の情報も用いるので、実運用できる場面は限られる。

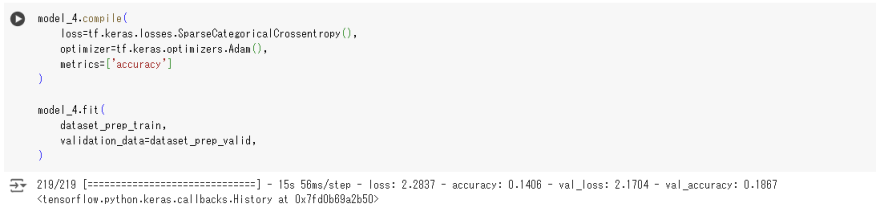


図 9: 双方向 RNN で学習させた結果

4.2 確認テスト

5 Section5: Seq2Seq

Seq2seq は、Encoder-Decoder モデルの一種であり、入力された文章から新しい文字列を出力する。Seq2seq の全体像を図 10 に示す。機械対話や機械翻訳に使用されている。以下の 2 つの RNN で構成されている。

- Encoder RNN
  入力された文章を「文脈」に変換する。具体的には、文章を単語等のトークン毎に分割し、ID を振る。その後、word2vec などを用いて、そのトークンを表す分散表現ベクトルに変換する。

- Decoder RNN
  Encoder から「文脈」を受け取り、新しい文字列を生成する。具体的には、受け取った分散表現ベクトルから単語等のトークンに変換（Encoder RNN の逆の操作）し、新しい文字列を生成する。

Seq2seq の課題として、一問一答しかできず、過去の文脈を考慮できていないことが挙げられる。そこで、Context RNN を用いて過去 $n-1$ 個の発話から次の発話を生成できるようにした HRED、HRED に VAE を用いてさらに改良した VHRED が考案されている。VAE とはオートエンコーダの一種で、潜在変数 z に確率分布 $N(0,1)$ を仮定することで、近い意味どうしの単語の分散表現ベクトルが距離的に近くなるようにした仕組みである。

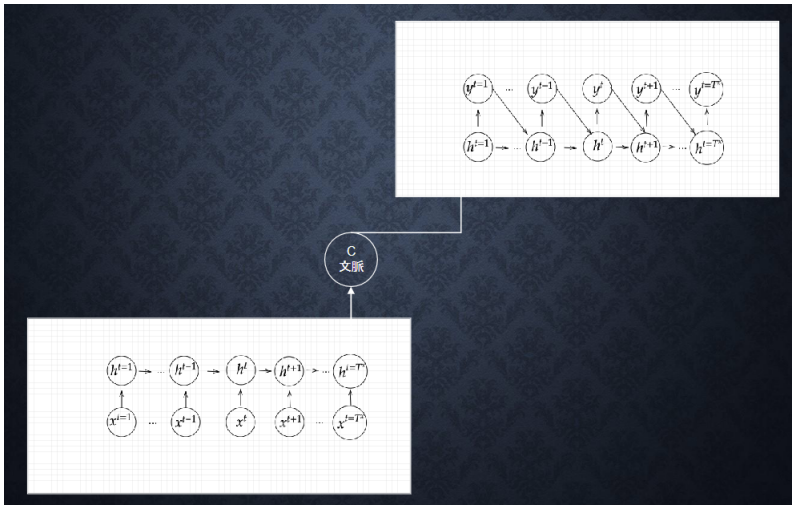


図 10: Seq2seq の全体像

5.1 確認テスト

5.1.1 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。

(2)

5.1.2 seq2seq と HRED、HRED と VHRED の違いを簡潔に述べよ。

seq2seq は一問一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生み出すニューラルネットワークである。HRED は、seq2seq の構造に、過去の文脈も考慮に加えれるように Decoder と Encorder ができるようにしたもの。VHRED は、HRED だと当たり障りのない回答しか作れなくなったことに対する VAE を用いた解決策である。

5.1.3 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。

確率分布

6 Section6: Word2vec

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ため、固定長形式で単語を表す必要がある。Word2vec は、単語を固定長のベクトル形式で表現するための方法である。自然言語処理を行う RNN の分散表現ベクトルの生成などに用いられる。
自然言語処理において、学習データから辞書を生成し、対象の単語が辞書に含まれていれば 1 とする one-hot ベクトルが提案されている。一方で、one-hot ベクトルだと辞書の単語数の長さのベクトルが生成され次元数が大きくなるため、word2vec で長さを小さく抑えている。

7 Section7: Attention Mechanism

seq2seq の課題として長い文章への対応が難しく、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ことが挙げられる。そのための解決策として、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。そこで、「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として、Attention Mechanism が考案されている。

7.1 確認テスト

7.1.1 RNN と word2vec、seq2seq と Attention の違いを簡潔に述べよ。

RNN は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である。word2vec は、単語の分散表現ベクトルを得るための手法である。seq2seq は、一つの時系列データから別の時系列データを得るニューラルネットワークである。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である。

8 VQ-VAE

VQ-VAE は、VAE の派生技術にあたる生成モデルである。「自然界の様々な事物の特徴を捉えるには離散変数の方が適している」という発想から、潜在変数が離散値となるように学習が行われる。これにより、従来の VAE で起こりやすいとされる“posterior collapse”の問題を回避し、高品質のデータを生成することが可能となる。

9 [フレームワーク演習] 双方向 RNN / 勾配のクリッピング

3_4_spoken_digit.ipynb では、spoken_digit データセットを各 RNN のモデルで学習させている。データセットの読込は、ある程度の前処理を行ったデータを手軽にダウンロードできる tensorflow_dataset を使っている。また、ライブラリの機能として、図 11 のように、データセットをシャッフルすることや分割することができる。図 11 では、70%を学習用、15%を検証用、15%をテスト用として分割している。
tensorflow.data.Dataset には、データセットの内容を編集できる機能がある。例

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

図 11: tensorflow_dataset を用いたデータセットの DL

を図 12 に示す。図 12 では、学習用と検証用のデータを NUM_DATA_POINTS の長さに揃えたうえで、入力データと教師ラベルのペアにしている。
tensorflow.keras には、簡単にモデルを生成できる仕組みが備わっている。例え

```
def cut_if_longer(el):
  return (
          tf.reshape(
              tf.cond(
                tf.greater(tf.shape(el['audio']), NUM_DATA_POINTS),
                true_fn=lambda: tf.slice(el['audio'], begin=[0], size=[NUM_DATA_POINTS]),
                false_fn=lambda: tf.slice(tf.concat([el['audio'], tf.zeros(NUM_DATA_POINTS, tf.int64)], axis=0), begin=[0], size=[NUM_DATA_POINTS])
              ),
              shape=(-1, 1)
          ),
          [el['label']]
        )


dataset_prep_train = dataset_train.map(cut_if_longer).batch(BATCH_SIZE)
dataset_prep_valid = dataset_valid.map(cut_if_longer).batch(BATCH_SIZE)

sample = next(iter(dataset_prep_valid))
sample
```

図 12: tf.data.Dataset を用いたデータセット内容の編集例

ば図 13 のように、add メソッドを実行することで、小規模な CNN を生成することができる。

10 [フレームワーク演習] Seq2Seq

3_5_Seq2Seq(Encoder-Decoder)_sin-cos.ipynb では、sin 関数の値から cos 関数の値を Seq2Seq モデルに予測させることを試みている。モデルの定義は図 14 である。図 14 から、エンコーダとデコーダはコンテキスト以外に繋がりのない分離したモデル構造となっている。なお、学習した結果は図 15 となる（赤が予測結果、青が正解)。所々学習できていない箇所があるが、ほぼ予測できているといえる。

```
[ ]  import tensorflow as tf
     from tensorflow.keras.models import Sequential
     from tensorflow.keras import layers

     tf.keras.backend.clear_session()

     model_1 = tf.keras.models.Sequential()


     model_1.add(layers.Input((NUM_DATA_POINTS, 1)))

     model_1.add(layers.Conv1D(32, 4, strides=2, activation='relu'))
     model_1.add(layers.MaxPooling1D(2))

     model_1.add(layers.GlobalAveragePooling1D())

     model_1.add(layers.Dense(10, activation='softmax'))

     model_1.summary()


     model_1.predict(sample[0])
```

図 13: tensorflow.keras を用いた CNN の生成

```
tf.keras.backend.clear_session()

e_input = tf.keras.layers.Input(shape=(NUM_STEPS, NUM_ENC_TOKENS), name='e_input')
_, e_state = tf.keras.layers.SimpleRNN(NUM_HIDDEN_PARAMS, return_state=True, name='e_rnn')(e_input)

d_input = tf.keras.layers.Input(shape=(NUM_STEPS, NUM_DEC_TOKENS), name='d_input')
d_rnn = tf.keras.layers.SimpleRNN(NUM_HIDDEN_PARAMS, return_sequences=True, return_state=True, name='d_rnn')
d_rnn_out, _ = d_rnn(d_input, initial_state=[e_state])

d_dense = tf.keras.layers.Dense(NUM_DEC_TOKENS, activation='linear', name='d_output')
d_output = d_dense(d_rnn_out)

model_train = tf.keras.models.Model(inputs=[e_input, d_input], outputs=d_output)
model_train.compile(optimizer='adam', loss='mean_squared_error')

model_train.summary()
```

図 14: Seq2Seq のモデル定義

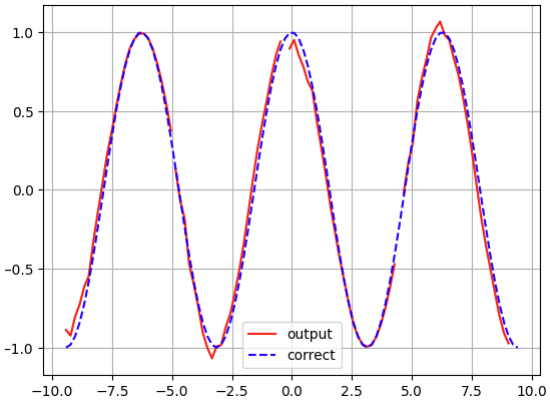


図 15: cos 関数の値を Seq2Seq に予測させた結果

11 [フレームワーク演習] data-augumentation

3_6_data_augmentation_with_tf.ipynb では、ある画像から疑似的に別の画像を生成するアプローチ（データ拡張：data augmentation）について実装している。data augmentation は、画像認識精度向上でよく用いられている手法であり、代表的な処理として以下が挙げられる。なお、data augmentation 前の元画像を図 16 に示す。

- Horizontal Flip（図 17）
  水平方向（左右）反転処理を行う。
- Vertical Flip（図 18）
  垂直方向（上下）反転処理を行う。
- Crop（図 19）
  あるサイズを画像中からランダムに切り出す処理を行う。
- Contrast（図 20）
  コントラストをランダムに調整する処理を行う。
- Brightness（図 21）
  輝度値 δ をランダムに調整する処理を行う。
- Hue（図 22）
  色相 δ をランダムに調整する処理を行う。
- Rotate（図 23）
  回転処理を行う。

図 16: 処理前の元画像

12 [フレームワーク演習] activate_functions

3_7_activation_functions.ipynb では、活性化関数について紹介している。活性化関数とは、ニューラルネットワークの順伝搬において、線形変換で得た値に対し

図 17: Horizontal Flip

図 18: Vertical Flip

図 19: Crop

図 20: Contrast

図 21: Brightness

図 22: Hue

図 23: Rotate

て非線形な変換を行う際に用いられる関数のことである。一般的に、微分の連鎖律を利用した誤差逆伝搬法を用いるために、微分可能かつ渡された値から直接的に微分値を求める導関数が利用できるという要件を満たした活性化関数が用いられる傾向にある。活性化関数として、以下が挙げられる。なお、各図において赤は関数値、青は微分値である。

- 中間層に用いる活性化関数
  - ステップ関数（図 24）
    入力値が閾値以上のときは 1、それ以外は 0 となる関数。閾値は基本的に 0 となる。導関数における入力値 0 は微分不可能。それ以外の微分値は 0。よって、誤差逆伝搬法に用いることはできない。導関数の実装では、入力値 0 の微分値を定義した劣微分が用いられる。

$$f(x) = \begin{cases} 1(x \geq 0) \\ 0(x < 0) \end{cases} \tag{3}$$

  - シグモイド関数（図 25）
    誤差逆伝播法の黎明期によく用いられた。導関数の最大値が 0.25 であり、入力値が 0 から遠ざかるほど 0 に近い微分値を取るため、中間層を深く重ねるほど勾配消失が発生しやすくなる（勾配消失問題）。

$$f(x) = \frac{1}{1 + e^{-x}} \tag{4}$$

  - tanh（図 26）
    シグモイド関数の代替とみなされていた。導関数の最大値が 1 であり、シグモイド関数の 0.25 以上だが、それでも入力値が 0 から遠ざかるほど 0 に近い微分値を取るため、中間層を深く重ねるほど勾配消失が発生しやすくなる。

$$f(x) = \tanh = \frac{\sinh x}{\cosh x} = \frac{e^x - e^{-x}}{e^x + e^{-x}} = 2\sigma(2x) - 1 \tag{5}$$

    ※ σ はシグモイド関数
  - ReLU（図 27）
    入力値が 0 以上のときは入力値と同じ値、それ以外のときは 0 を取る関数。原点において不連続（微分不可能な点が存在する）であるため、導関数の実装では、入力値 0 の微分値を定義した劣微分が用いられる。導関数における入力値が正のときの微分値が常に 1 なので、勾配消失が発生しにくい。一方で、導関数における入力値が負のとき、学習が進まない問題がある。

$$f(x) = \begin{cases} x(x > 0) \\ 0(x \leq 0) \end{cases} = \max(0, x) \tag{6}$$

  - Leaky ReLU（図 28）
    入力値が 0 以上のときは入力値と同じ値、それ以外のときは入力値に

定数 α を乗算した関数。α の値は基本的には 0.01 とする。原点において不連続（微分不可能な点が存在する）であるため、導関数の実装では、入力値 0 の微分値を定義した劣微分が用いられる。導関数における入力値が正のときの微分値が常に 1 なので、勾配消失が発生しにくい。導関数における入力値が負のときは、小さな傾きの 1 次関数となる。

$$f(x) = \begin{cases} x(x > 0) \\ \alpha x(x \leq 0) \end{cases} = \max(\alpha x, x) \tag{7}$$

- Swish（図 29）
  ReLU の代替候補として注目されている関数。ReLU や Leaky ReLU とは異なり、原点において連続（微分不可能な点が存在しない）。β の値は基本的に 1 とする。

$$f(x) = x\sigma(\beta x) = \frac{x}{1 + e^{-\beta x}} \tag{8}$$

  ※ σ はシグモイド関数

- 出力層に用いる活性化関数
  - 2 値分類: シグモイド関数（図 25）
    出力層のサイズ（ユニット数）は 1。シグモイド関数の出力値を、正例である確信度（Positive confidence）と呼ぶ。正例である確信度の取り得る値は、 0 から 1 迄の範囲内。負例である確信度（Negative confidence）は、 1 に対して正例である確信度を減算して求める。分類するために、閾値を設定する。閾値は基本的には 0.5。正例である確信度が閾値を上回るときは正例（クラス番号 1 ）、それ以外は負例（クラス番号 0 ）に分類する。
  - 多値分類: ソフトマックス関数（図 30）
    出力層のサイズ（ユニット数）はクラス数と同じ。シグモイド関数を多値分類用に拡張したもので、出力値の総和が 1 となる。

$$g(\boldsymbol{x})_i = \frac{e^{x_i}}{\sum_{k=1}^{n} e^{x_k}} \quad \{i \in \mathbb{N} | 1 \leq i \leq n\} \tag{9}$$

  - 回帰: 恒等関数 (活性化関数なし)（図 31）
    出力層のサイズ（ユニット数）は 1。出力値は入力値と同じ。

$$g(x) = x \tag{10}$$

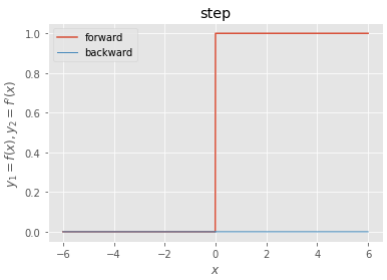


図 24: ステップ関数

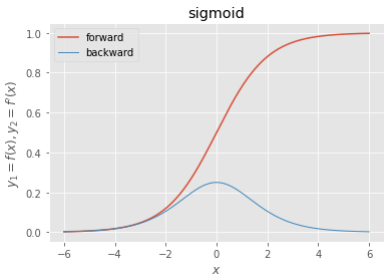


図 25: シグモイド関数

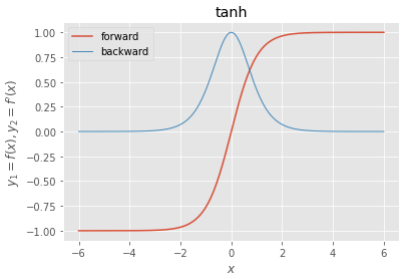


図 26: tanh

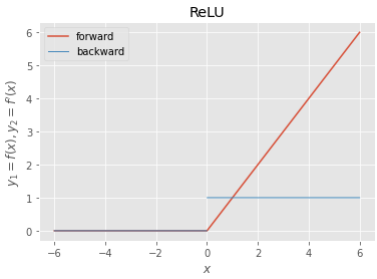


図 27: ReLU

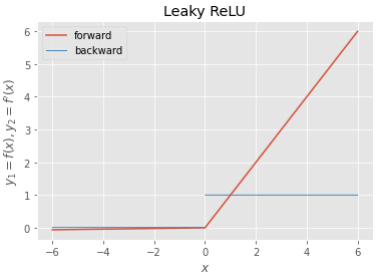


図 28: Leaky ReLU

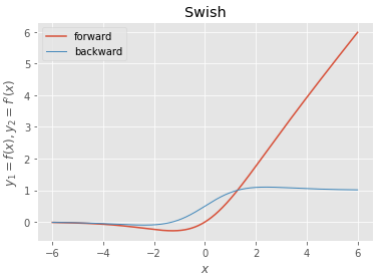


図 29: Swish

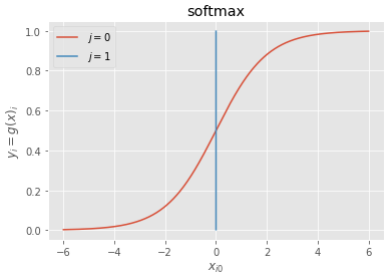


図 30: ソフトマックス関数

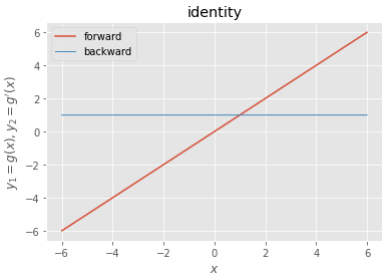


図 31: 恒等関数