深層学習 DAY3 レポート

2024 年 06 月 16 日 新規作成

再帰型ニューラルネットワークの概念

　今までのフィードフォワード型のニューラルネットワークでは時系列データをうまく扱えなかった。例えば株価のような時系列データでは過去の情報が将来の結果に少なからず影響を与える。このようなタスクに対応するのが再帰ニューラルネットワーク(RNN)である。RNN では、入力データに対して過去の情報を活用して予測や生成を行うことができるため、音声認識、自然言語処理、時系列予測などのタスクに広く使用される。パラメータの最適化には Backpropagation Through Time(BPTT)が使われる。

確認テスト 1

Q:サイズ 5x5 の入力画像をサイズ 3x3 のフィルタで畳込んだ時の出力画像のサイズ
ストライド=2, パディング=1 のとき、

$$
\begin{aligned}
OH &= \frac{5 + 2 \times 1 - 3}{2} + 1 \\
&= 3 \\
OW &= \frac{5 + 2 \times 1 - 3}{2} + 1 \\
&= 3
\end{aligned}
$$

確認テスト 2

Q:RNN ネットワークには大きく分けて 3 つの重みがある。入力から中間層を定義する際の重み、中間層から出力を定義する際の重み以外の残り 1 つを説明する。

同じ層の一つ前のノードの出力を次のノードに入力する際の重み。これは中間層の再帰的な重みで、一つ前の出力を次の時刻の状態に変換するための重みとなっており、過去の情報を保持し次の状態を生成することができる。

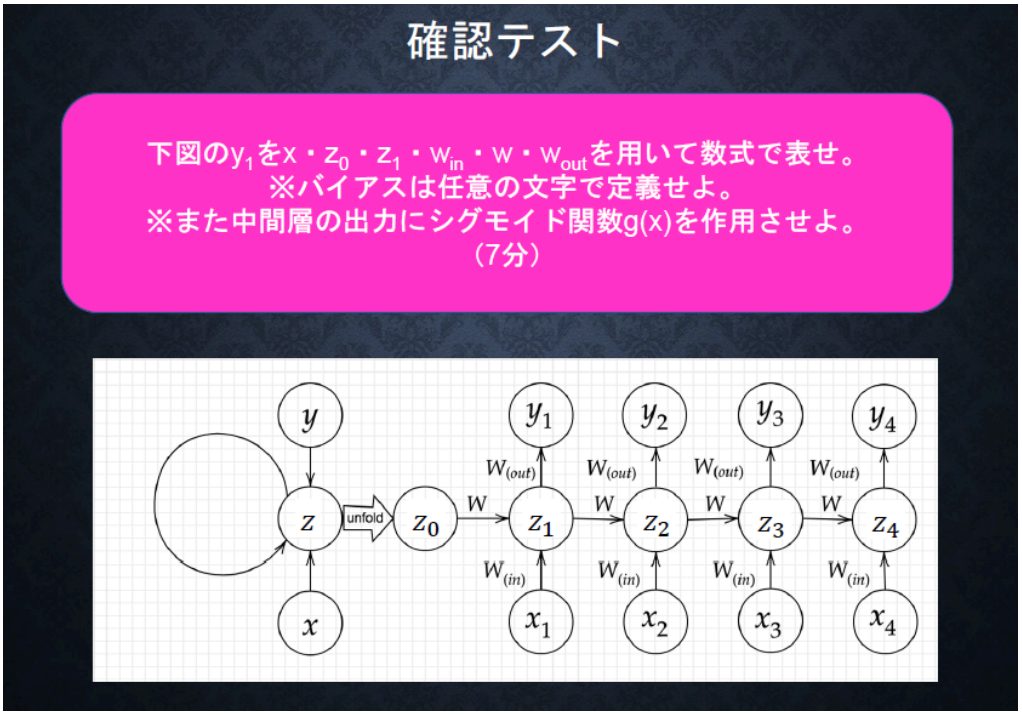
確認テスト 3

dz/dxを求める。

$$z = t^2 \tag{1}$$
$$t = x + y \tag{2}$$

$$\begin{aligned}\frac{dz}{dx} &= \frac{dz}{dt} \cdot \frac{dt}{dx} \\ &= \big(2t\big) \cdot \big(1\big) \\ &= 2t\end{aligned}$$

確認テスト 4



$$y_1 = g(W_{(out)} z_1 + b_{out})$$
$$z_1 = W_{(in)} x_1 + W z_0 + b_{(in)}$$

実装演習

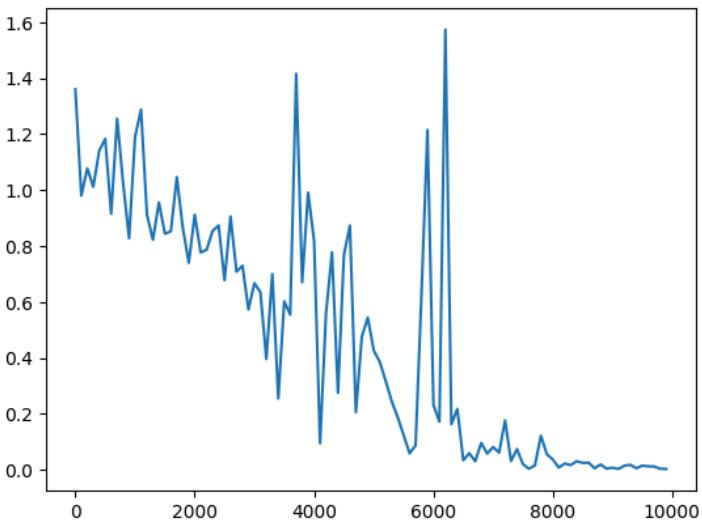
```
# データを用意
# 2進数の桁数
binary_dim = 8
# 最大値 + 1
largest_number = pow(2, binary_dim)
# largest_numberまで2進数を用意
binary = np.unpackbits(
    np.array([range(largest_number)], dtype=np.uint8).T, axis=1)

input_layer_size = 2
hidden_layer_size = 16
output_layer_size = 1

weight_init_std = 1
learning_rate = 0.5

iters_num = 10000
plot_interval = 100

# ウェイト初期化 (バイアスは簡単のため省略)
W_in = weight_init_std * np.random.randn(input_layer_size, hidden_layer_size)
W_out = weight_init_std * np.random.randn(hidden_layer_size, output_layer_size)
W = weight_init_std * np.random.randn(hidden_layer_size, hidden_layer_size)
# Xavier
# W_in = np.random.randn(input_layer_size, hidden_layer_size) / (np.sqrt(input_layer_size))
# W_out = np.random.randn(hidden_layer_size, output_layer_size) / (np.sqrt(hidden_layer_size))
# W = np.random.randn(hidden_layer_size, hidden_layer_size) / (np.sqrt(hidden_layer_size))
# He
# W_in = np.random.randn(input_layer_size, hidden_layer_size) / (np.sqrt(input_layer_size)) * np.sqrt(2)
# W_out = np.random.randn(hidden_layer_size, output_layer_size) / (np.sqrt(hidden_layer_size)) * np.sqrt(2)
# W = np.random.randn(hidden_layer_size, hidden_layer_size) / (np.sqrt(hidden_layer_size)) * np.sqrt(2)


# 勾配
W_in_grad = np.zeros_like(W_in)
W_out_grad = np.zeros_like(W_out)
W_grad = np.zeros_like(W)

u = np.zeros((hidden_layer_size, binary_dim + 1))
z = np.zeros((hidden_layer_size, binary_dim + 1))
y = np.zeros((output_layer_size, binary_dim))

delta_out = np.zeros((output_layer_size, binary_dim))
delta = np.zeros((hidden_layer_size, binary_dim + 1))

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

        # 誤差
        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

        out_bin[binary_dim - t - 1] = np.round(y[:, t])

    for t in range(binary_dim)[::-1]:
        X = np.array([a_bin[-t - 1], b_bin[-t - 1]]).reshape(1, -1)

        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])
#         delta[:,t] = (np.dot(delta[:,t+1].T, W.T) + np.dot(delta_out[:,t].T, W_out.T)) * functions.d_relu(u[:,t
#         delta[:,t] = (np.dot(delta[:,t+1].T, W.T) + np.dot(delta_out[:,t].T, W_out.T)) * d_tanh(u[:,t+1])

        # 勾配更新
        W_out_grad += np.dot(z[:, t + 1].reshape(-1, 1),
                             delta_out[:, t].reshape(-1, 1))
```

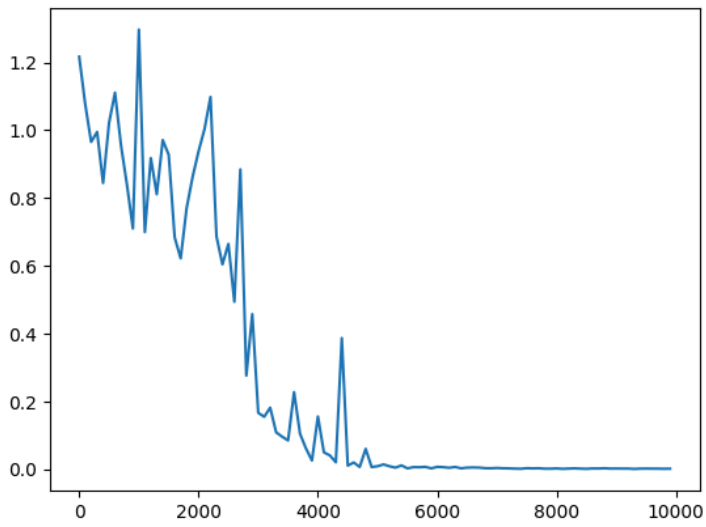
```
        W_grad += np.dot(z[:, t].reshape(-1, 1), delta[:, t].reshape(1, -1))
        W_in_grad += np.dot(X.T, delta[:, t].reshape(1, -1))

    # 勾配適用
    W_in -= learning_rate * W_in_grad
    W_out -= learning_rate * W_out_grad
    W -= learning_rate * W_grad

    W_in_grad *= 0
    W_out_grad *= 0
    W_grad *= 0

    if (i % plot_interval == 0):
        all_losses.append(all_loss)
        print("iters:" + str(i))
        print("Loss:" + str(all_loss))
        print("Pred:" + str(out_bin))
        print("True:" + str(d_bin))
        out_int = 0
        for index, x in enumerate(reversed(out_bin)):
            out_int += x * pow(2, index)
        print(str(a_int) + " + " + str(b_int) + " = " + str(out_int))
        print("------------")

lists = range(0, iters_num, plot_interval)
plt.plot(lists, all_losses, label="loss")
plt.show()
```

中間層サイズの違い

`hidden_layer_size = 16`

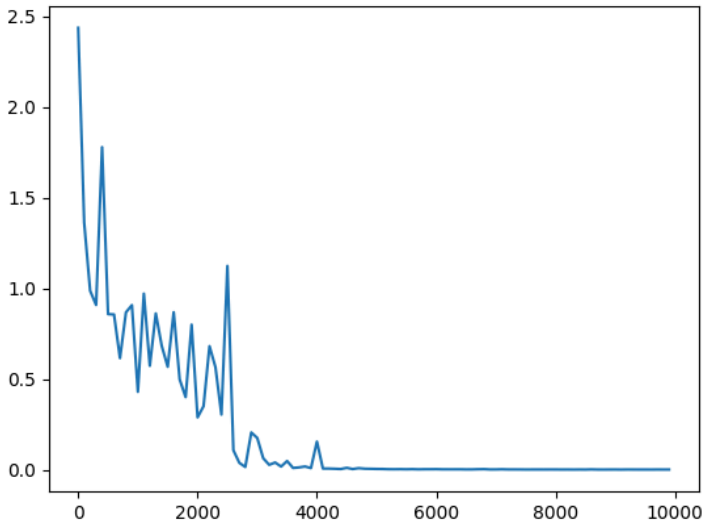


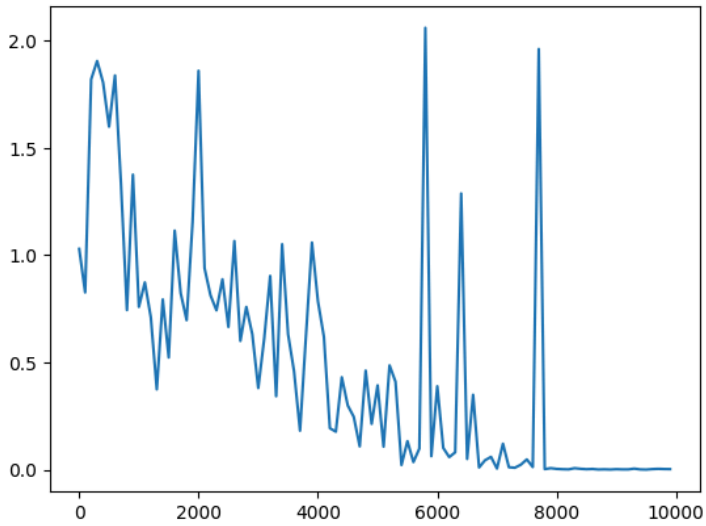
`hidden_layer_size = 32`



`hidden_layer_size = 64`

`hidden_layer_size = 128`

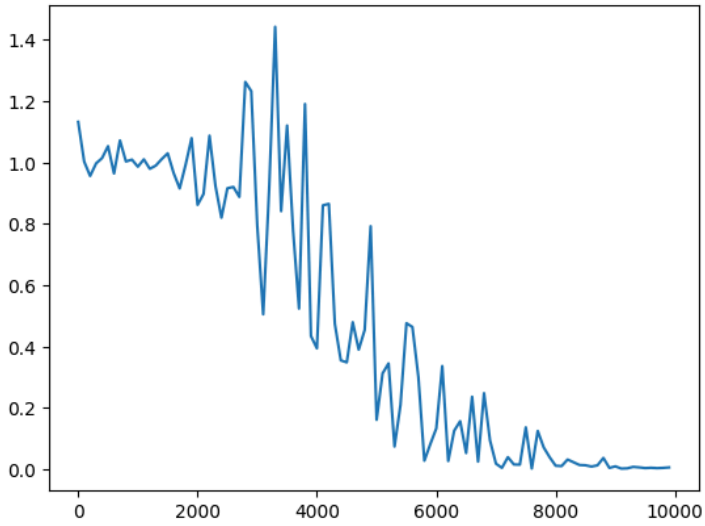


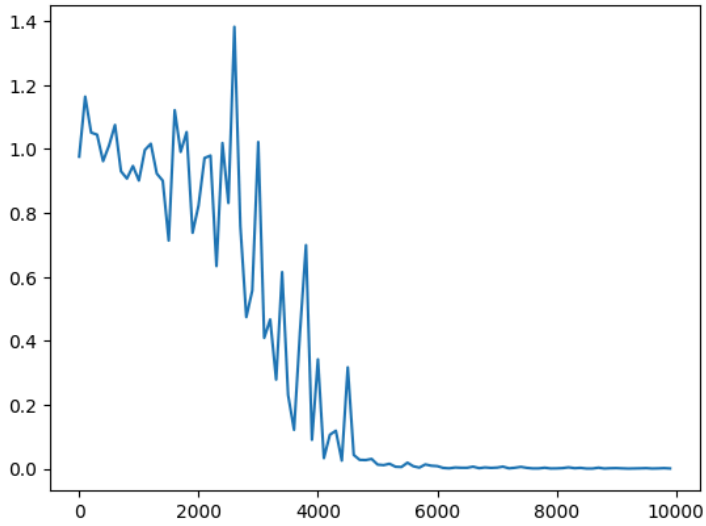


重み初期化方法の違い（Sigmoid）

`Xavier`

`He`

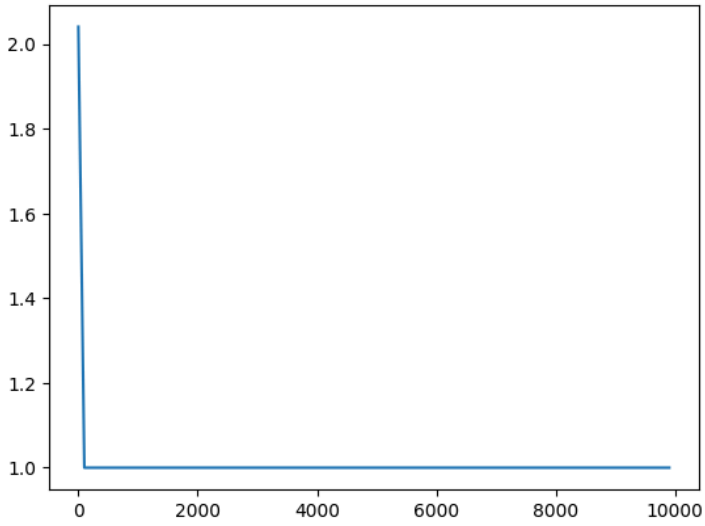


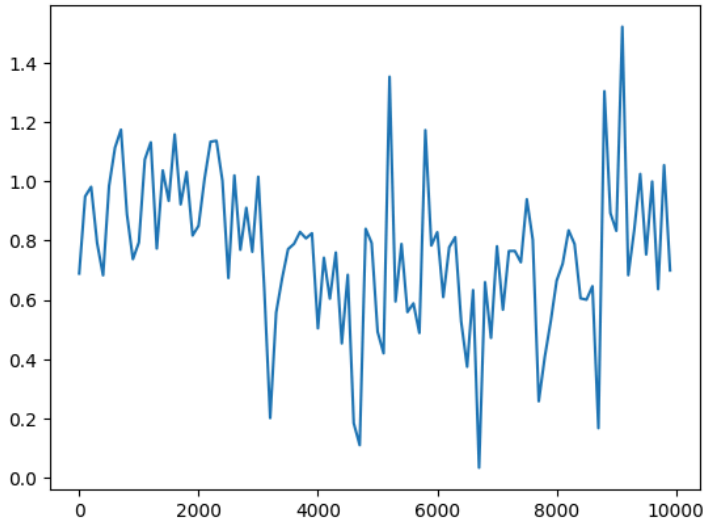


中間層活性化関数の違い

`ReLU`

`tanh`





考察

中間層のサイズを大きくしていったら、64 までは誤差の収束が速くなっているようだが、128 では逆に長くなった。誤差の収束が速くなったとしても汎化性能については懸念される。中間層の活性化関数を`ReLU`にした場合では、重みの更新はされなくなった。これは中間層への入力が 0 以下になり活性化関数の出力が 0 になったためではないかと考えられる。これは「死んだニューロン」問題と呼ばれている。活性化関数を`tanh`にした場合では学習が不安定になっており、これは勾配消失または勾配爆発ではないかと考えられる。

LSTM

RNN では時系列が長くなると見かけ上の層が深くなってしまい、勾配消失や勾配爆発といった問題が生じた。この問題を解決するために LSTM(Long Short-Tern Memory)が考案された。LSTM では 3 つのゲートと一つのセル状態を持つ。

- 入力ゲート
  新しい情報をどの程度セル状態に追加するかを決定する。
- セル状態の更新
  忘却ゲートと入力ゲートの出力を使ってセル状態を更新する
- 忘却ゲート
  忘れるべき情報を決定する。過去のセル状態を次の状態にどの程度残すか決める。
- 出力ゲート
  次の隠れ状態を決定する。

入力ゲートと忘却ゲートにより重要な情報を長期間にわたって保持することができるようになる。また、勾配消失と勾配爆発が緩和され安定した学習が可能になる。

確認テスト 1

シグモイド関数の微分

$$
\begin{aligned}
\frac{d\sigma(x)}{dx} &= \frac{d}{dx}\frac{1}{1+e^{-ax}} \\
&= \frac{-1}{(1+e^{-ax})^2}\cdot -ae^{-ax} \\
&= \frac{a}{1+e^{-ax}}\cdot\frac{1+e^{-ax}-1}{1+e^{-ax}} \\
&= a\frac{1}{1+e^{-ax}}\cdot\left(1-\frac{1}{1+e^{-ax}}\right) \quad = a\sigma(x)(1-\sigma(x))
\end{aligned}
$$

$\sigma(0)=0.5$なので$a=1$のとき$\frac{d\sigma(0)}{dx}=0.25$

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。文中の「とても」という言葉はなくても影響を及ぼさない。この場合にはどのゲートが作用すると考えられるか。

忘却ゲート

考察

　RNN では勾配を更新する際に行列の積を計算していたが、LSTM では要素ごとの積（アダマール積）を計算する。毎時刻、異なるゲート値によって要素ごとの積の計算が行われることで情報の選択と保持が可能になる。特に、忘却ゲートが情報を保持するか忘れるかを決定するため LSTM は勾配消失問題を緩和し、長期的な依存関係を保持することが期待できる。

GRU

　LSTM は勾配消失問題を緩和することができたが、パラメータ数が多く計算に時間がかかった。そこで LSTM に代わるゲート付き RNN として GRU が提案された。GRU では LSTM と違って記憶セルを持たず、LSTM をよりシンプルにしたアーキテクチャである。使われるゲートはリセットゲートとアップデートゲートの二つである。

- リセットゲート
  過去の隠れ状態をどれだけ無視するか決定する。
- アップデートゲート
  隠れ状態を更新する。これは LSTM の忘却ゲートと入力ゲートの二つの役割を担う。

確認テスト 1

Q:LSTM と CEC が抱える課題
LSTM はパラメータ数が多く計算に時間がかかり、CEC でデータを保持するためメモリ使用量を増加させる。

確認テスト 2

Q:LSTM と GRU の違い
LSTM は入力ゲート、忘却ゲート、出力ゲート、CEC の 4 つの構成に対して、GRU ではリセットゲートとアップデートゲートの二つとなっている。

実装演習

`DNN_code_colab_day3\notebook\3_2_tf_languagemodel\predict_word.ipynb` を実行

```
ln = Language()

# 学習済みのパラメータをロード
ln.load_weights("./data_for_predict/predict_model")

# 保存したモデルを使って単語の予測をする
ln.predict("some of them looks like")
```

結果

```
...
well-known : 1.865201e-21
fair : 6.3084947e-22
below : 2.9081717e-23

Prediction: some of them looks like et
```

考察

　このコードは英文を学習し次の単語を予測するモデルとなっている。モデルには LSTM が使用されている。予測結果は「et」という不自然な単語を返している。これは学習データに偏りがあるかデータセットが少なかったためではないかと考えられる。また、chunk_size は 5 と入力シーケンスが短いためモデルが文脈を正しく理解できていない可能性もある。

双方向 RNN

　LSTM の情報伝達の流れは過去から未来に向かっていたが、双方向 RNN では過去から未来方向に加えて未来から過去方向にも情報を伝達する。文章翻訳などのタスクでは翻訳すべき文章がすべて与えられているので、文章を左から右にだけでなく、右から左に処理し学習することもできる。

実装演習

単純 RNN

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

tf.keras.backend.clear_session()
model_2 = tf.keras.models.Sequential()
model_2.add(layers.Input((NUM_DATA_POINTS, 1)))
model_2.add(layers.SimpleRNN(128))
model_2.add(layers.Dense(10, activation='softmax'))
model_2.summary()
model_2.predict(sample[0]).shape
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,
)
```

結果

```
219/219 [==============================] - 118s 535ms/step
- loss: 2.3746 - accuracy: 0.1000 - val_loss: 2.3463 - val_accuracy: 0.0773
```

GRU

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

tf.keras.backend.clear_session()
model_3 = tf.keras.models.Sequential()
model_3.add(layers.Input((NUM_DATA_POINTS, 1)))
model_3.add(layers.GRU(128))
model_3.add(layers.Dense(10, activation='softmax'))
model_3.summary()
model_3.predict(sample[0]).shape
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,
)
```

結果

```
219/219 [==============================] - 10s 39ms/step
- loss: 2.3868 - accuracy: 0.1069 - val_loss: 2.3029 - val_accuracy: 0.1387
```

双方 RNN(LSTM)

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

tf.keras.backend.clear_session()
model_4 = tf.keras.models.Sequential()
model_4.add(layers.Input((NUM_DATA_POINTS, 1)))
model_4.add(layers.Bidirectional(layers.LSTM(64)))
model_4.add(layers.Dense(10, activation='softmax'))
model_4.summary()
model_4.predict(sample[0]).shape
model_4.compile(
    loss=tf.keras.losses.SparseCategoricalCrossentropy(),
    optimizer=tf.keras.optimizers.Adam(),
    metrics=['accuracy']
)

model_4.fit(
    dataset_prep_train,
    validation_data=dataset_prep_valid,
)
```

結果

```
219/219 [==============================] - 16s 60ms/step
- loss: 2.2779 - accuracy: 0.1469 - val_loss: 2.1503 - val_accuracy: 0.1867
```

LSTM(勾配クリッピング)

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

tf.keras.backend.clear_session()
model_5 = tf.keras.models.Sequential()
model_5.add(layers.Input((NUM_DATA_POINTS, 1)))
model_5.add(layers.LSTM(64))
model_5.add(layers.Dense(10, activation='softmax'))
model_5.summary()
model_5.predict(sample[0]).shape
model_5.compile(
    loss=tf.keras.losses.SparseCategoricalCrossentropy(),
    optimizer=tf.keras.optimizers.Adam(clipvalue=0.5),
    metrics=['accuracy']
)

model_5.fit(
    dataset_prep_train,
    validation_data=dataset_prep_valid,
)
```

結果

```
219/219 [==============================] - 11s 34ms/step
- loss: 2.3477 - accuracy: 0.1120 - val_loss: 2.2879 - val_accuracy: 0.1440
```

考察

`val_accuracy`が一番高いモデルは双方向 RNN だった。今回は音声に関する多クラス分類タスクだったので時系列データの一方向だけでなく逆方向についても学習できる双方向 RNN がマッチしていたのではないかと考えられる。

Seq2Seq

seq2seq は 2 つの RNN を利用して時系列データから別の時系列データに変換するモデルである。Encoder-Decoder モデルとも呼ばれる。入力データを Encoder で符号化し、その出力を Decoder に通して復号化を行う。Encoder では任意の長さの文章を固定長のベクトルに変換する処理を行い、このベクトルを Decoder が受け取って目的とするデータを生成する。seq2seq は機械対話や機械翻訳に使用される。

- HRED(Hierarchical Recurrent Encoder-Decoder)
  seq2seq は一問一答のため会話の流れをつかむことができない。HRED では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のため過去の発話の履歴を加味した回答を得ることができる。

- VHRED(Variational Hierarchical Recurrent Encoder-Decoder)
  HRED の拡張版で変分オートエンコーダーの概念を取り入れた。生成される会話応答の多様性を高めた。
- VAE
  ニューラルネットワークを使って次元削減する手法をオートエンコーダーという。オートエンコーダーでは Encoder と Decoder の構造になっており、入力データと正解データを同じデータにすることで、Encoder の出力が次元圧縮されたベクトルになる。VAE では Encoder 出力の潜在変数を標準化したものとなる。

確認テスト 1

Q:seq2seq について説明しているものを選ぶ

(1). 双方向 RNN
(2). seq2seq
(3). Recursive Neural Network
(4). LSTM

確認テスト 2

Q:seq2seq と HRED、HRED と VHRED の違い
seq2seq では履歴の情報を捉えることができなかったが、HRED では対話全体の文脈を反映できる。ところが HRED は多様性がない返答になってしまうことがあったため、VAE の概念を取り入れて多様性を確保したのが VHRED である。

確認テスト 3

Q:VAE に関する下記の説明に当てはまる言葉
自己符号化器の潜在変数に**正規分布**を導入したもの

考察

seq2seq を改良する方法には以下の二つがある。

- 入力データの反転
  入力データの文字の順番を逆転させる。この方法は多くの場合で学習の進みが速くなり最終的な精度が向上する。
- のぞき見
  Encoder の出力hを Decoder のほかのレイヤにも与える。

Word2vec

Word2vec は単語を意味のあるベクトルに変換する手法のことをいう。このベクトルは単語の分散表現と呼ばれる。代表的なニューラルネットワークのモデルには CBOW(Continuous bag-of-words)と skip-gram がある。分散表現は単語間の意味的な類似性を捉えたベクトルになる。テキストデータの分析や機械学習モデルの前処理として広く知用される。

- CBOW
  周囲のコンテキストからターゲットを予測する。
- skip-gram
  ターゲット単語から周囲の単語を予測する。

考察

シンボルグラウンディング問題がありましたが、自然言語と意味（と思われるもの）の結び付けができることが分かりました。機械翻訳やチャットボットに代表されるように自然言語処理は長足に発展してきましたが、単語をどうやって数値しか扱えないニューラルネットワークで処理しているのかという疑問を解消することができました。

Attention Mechanism

Attention はシーケンスデータを扱う際に重要な部分に注意を向けるための機構である。従来の seq2seq モデルでは長い文章であっても固定長のベクトルに変換しなければならず、特に長いシーケンスの場合、情報が圧縮されすぎて重要な情報が失われる問題があった。Attention 機構を使うことで、各単語の重要度を動的に計算し全体の情報を効果的に活用できるようになる。

確認テスト 1

Q:RNN と word2vec、seq2seq と Attention の違い
RNN は時系列データを取り扱うモデル、Word2vec は単語を意味のあるベクトルに変換する技術、seq2seq はシーケンスデータを別のシーケンスデータに変換するモデルで、Attention は重要な部分の重みを大きくして伝える機構。

考察

Transformer を理解する上で重要な Attention 機構について学びました。近年の AI ブームを支えていると言えるほど重要な技術だと思います。今後も技術発展に注意を払っていきたいと思います。

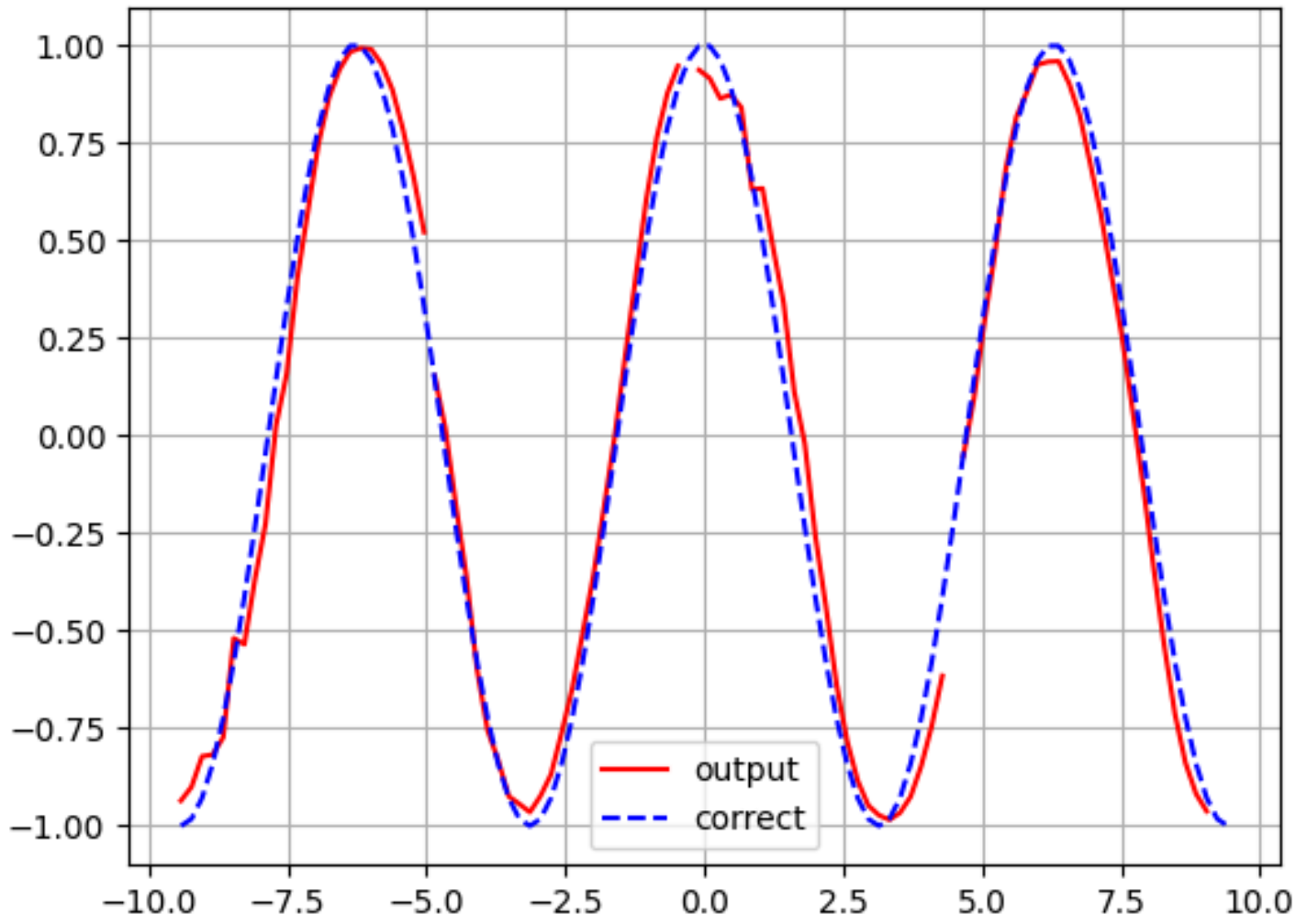
VQ-VAE

　オートエンコーダーをベースとしており、潜在変数が正規分布に従うように学習していた VAE と違って潜在変数が離散値になるように学習する。離散化にはベクトル量子化処理が行われる。離散化のメリットは次の 2 つが挙げられる。一つは言語が離散的であるため離散的表現を用いた方がより簡潔にデータの特徴を捉えることができると期待されるため、もう一つは posterior collapse と呼ばれる問題を回避するためである。VQ-VAE は生成されたデータの質を向上させることが期待できる。このモデルは openAI の DALL-E などで使われている。

フレームワーク演習 Seq2Seq

```
import tensorflow as tf
import numpy as np
import matplotlib.pyplot as plt

NUM_ENC_TOKENS = 1
NUM_DEC_TOKENS = 1
NUM_HIDDEN_PARAMS = 10
NUM_STEPS = 24
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()
n = len(x) - NUM_STEPS
ex = np.zeros((n, NUM_STEPS))
dx = np.zeros((n, NUM_STEPS))
dy = np.zeros((n, NUM_STEPS))

for i in range(0, n):
  ex[i] = seq_in[i:i + NUM_STEPS]
  dx[i, 1:] = seq_out[i:i + NUM_STEPS - 1]
  dy[i] = seq_out[i: i + NUM_STEPS]

ex = ex.reshape(n, NUM_STEPS, 1)
dx = dx.reshape(n, NUM_STEPS, 1)
dy = dy.reshape(n, NUM_STEPS, 1)
BATCH_SIZE = 16
EPOCHS = 80

history = model_train.fit([ex, dx], dy, batch_size=BATCH_SIZE, epochs=EPOCHS, validation_split=0.2, verbose=False
model_pred_e = tf.keras.models.Model(inputs=[e_input], outputs=[e_state])
pred_d_input = tf.keras.layers.Input(shape=(1, 1))
pred_d_state_in = tf.keras.layers.Input(shape=(NUM_HIDDEN_PARAMS))
pred_d_output, pred_d_state = d_rnn(pred_d_input, initial_state=[pred_d_state_in])
pred_d_output = d_dense(pred_d_output)
pred_d_model = tf.keras.Model(inputs=[pred_d_input, pred_d_state_in], outputs=[pred_d_output, pred_d_state])
def predict(input_data):
  state_value = model_pred_e.predict(input_data)
  _dy = np.zeros((1, 1, 1))

  output_data = []
  for i in range(0, NUM_STEPS):
```

```
    y_output, state_value = pred_d_model.predict([_dy, state_value])

    output_data.append(y_output[0, 0, 0])
    _dy[0, 0, 0] = y_output

  return output_data
init_points = [0, 24, 49, 74]

for i in init_points:
  _x = ex[i : i + 1]
  _y = predict(_x)

  if i == 0:
    plt.plot(x[i : i + NUM_STEPS], _y, color="red", label='output')
  else:
    plt.plot(x[i : i + NUM_STEPS], _y, color="red")

plt.plot(x, seq_out, color = 'blue', linestyle = "dashed", label = 'correct')
plt.grid()
plt.legend()
plt.show()
```



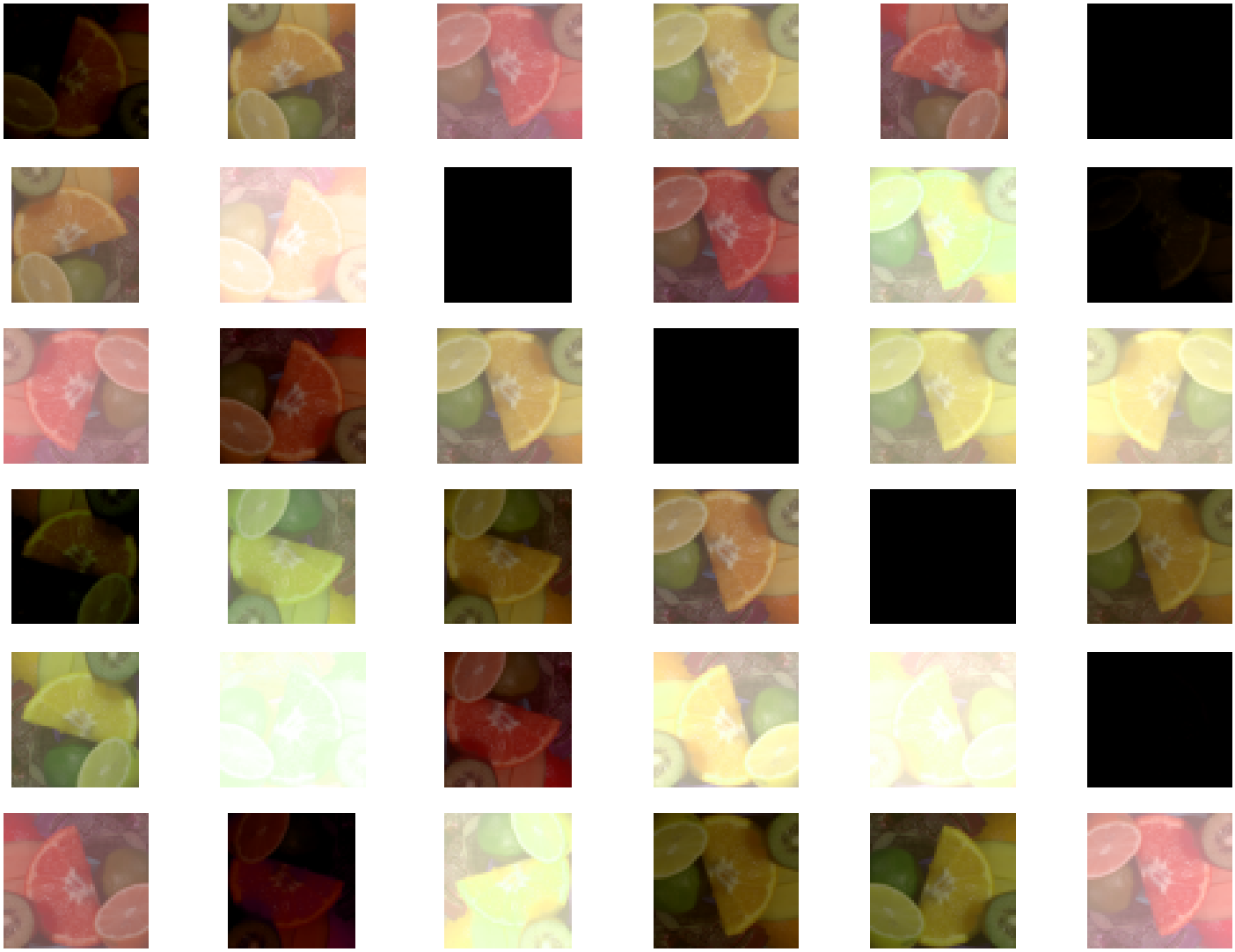
今回は SimpleRNN を使用しているがこれを LSTM や GRU に変更すればより良い結果が期待できる。長期依存関係をよりよく捉えられる可能性があり、勾配消失問題を軽減し長いシーケンスの処理に適している。また、Attention 機構を導入することも効果的と考えられる。

フレームワーク演習 data-augumentation

画像の認識精度向上にはデータの水増しが有効であることが知られている。以下にその手法を挙げる。

- 水平方向、上下方向の反転
- 画像のきりとり
- コントラスト、輝度、色相の変更
- 画像の回転
- 部分的なマスキング
- 画像ミックス

```
def data_augmentation(image):
    image = tf.image.random_flip_left_right(image)
    image = tf.image.random_flip_up_down(image)
    image = tf.image.random_contrast(image, lower=0.4, upper=0.6)
    image = tf.image.random_brightness(image, max_delta=0.8)
    image = tf.image.rot90(image, k=random.choice((0, 1, 2)))
    image = tf.image.random_hue(image, max_delta=0.1)
    return image

image = image_origin
show_images([data_augmentation(image).numpy() for _ in range(36)])
```

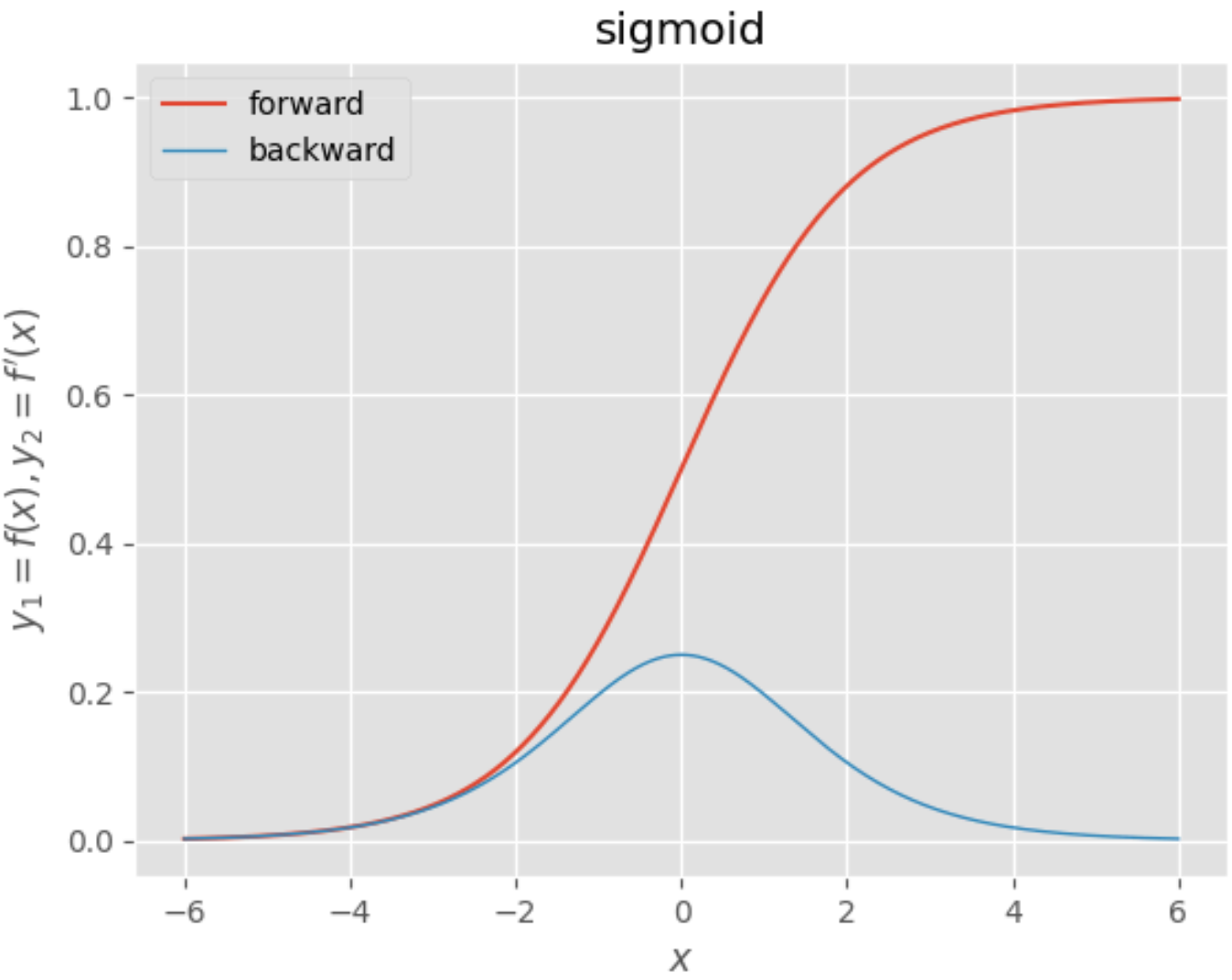


フレームワーク演習 activate_functions

　ニューラルネットワークの順伝播では、線形変換で得た値に対して非線形な変換を行う。非線形な変換を行う際に用いられる関数を活性化関数という。ニューラルネットワークのパラメータの最適化には連鎖率を用いた誤差逆伝播法が使われるため活性化関数は微分可能であることが望ましい。

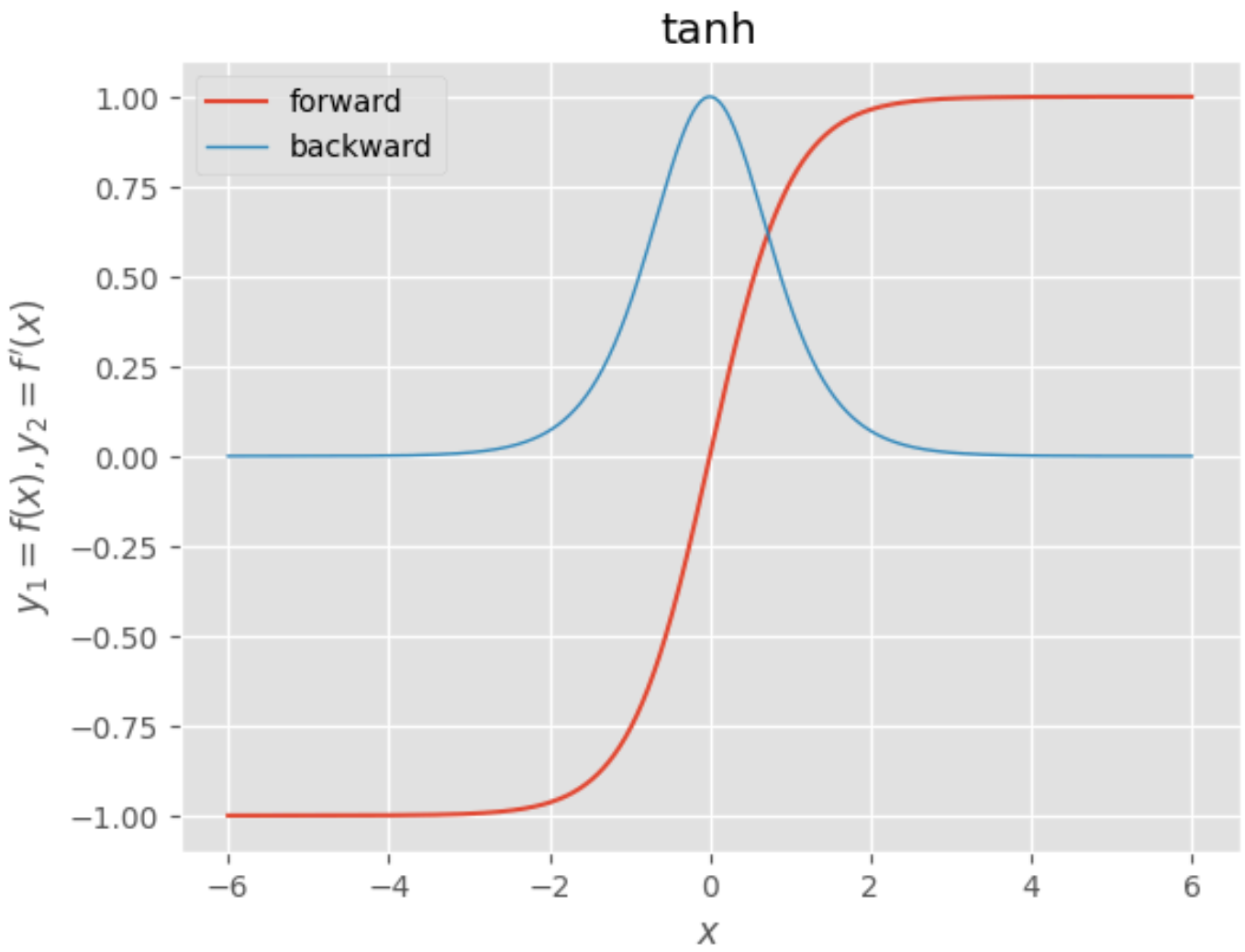
- Sigmoid
  誤差逆伝播法の黎明期によく用いられた。勾配消失問題が発生しやすい。

```python
def sigmoid(x):
    """forward
        sigmoid
        シグモイド関数
    """
    return 1.0 / (1.0 + np.exp(-x))

def d_sigmoid(x):
    """backward
        derivative of sigmoid
        シグモイド関数の導関数
    """
    dx = sigmoid(x) * (1.0 - sigmoid(x))
    return dx
```



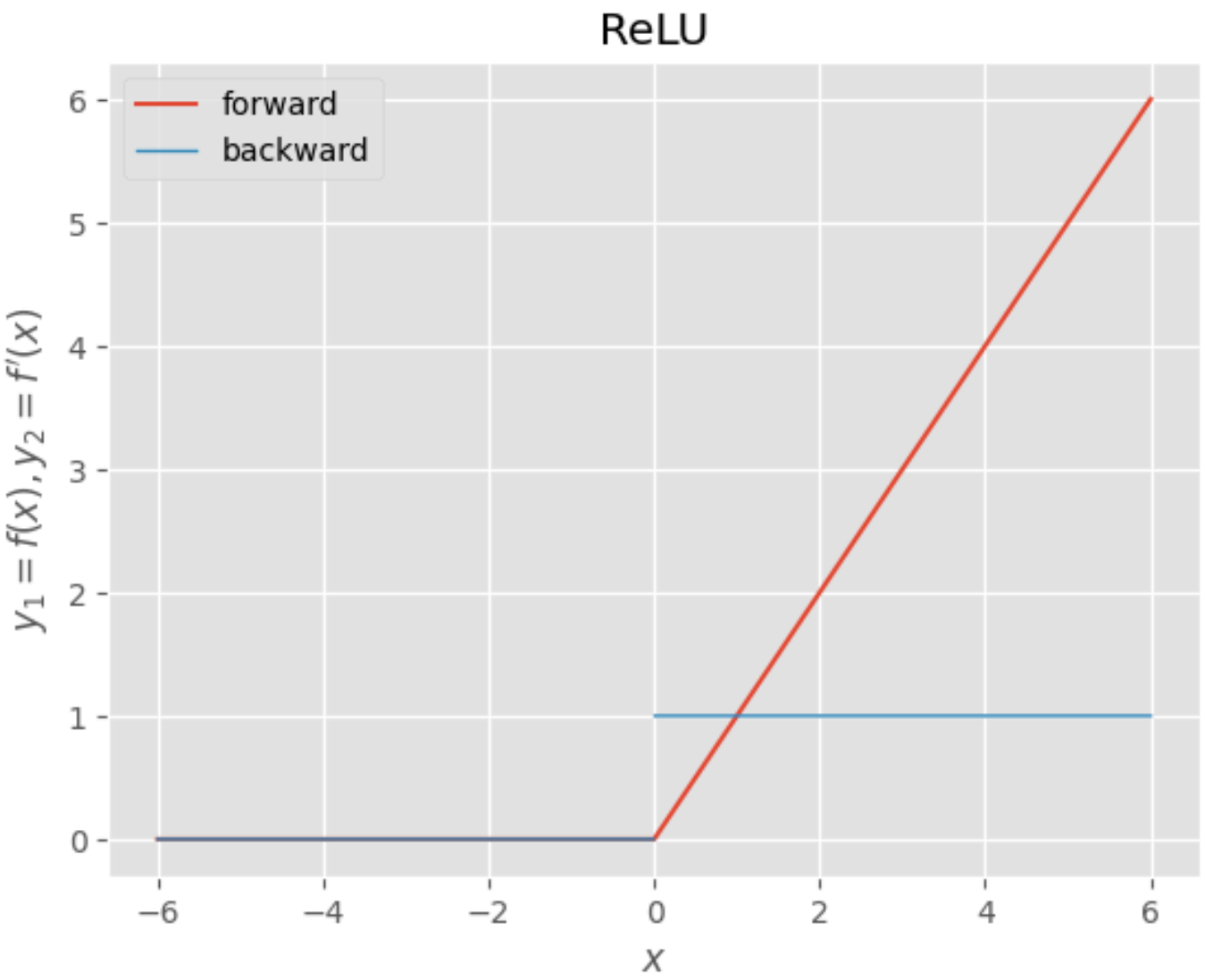
- tanh
  双曲線正接関数。中間層を深く重ねるほど勾配消失が発生しやすくなる。

```python
def tanh(x):
    """forward
        tanh
        双曲線正接関数
    """
    return np.tanh(x)
def d_tanh(x):
    """backward
        derivative of tanh
        双曲線正接関数の導関数
    """
    dx = 1.0 / np.square(np.cosh(x))
    return dx
```



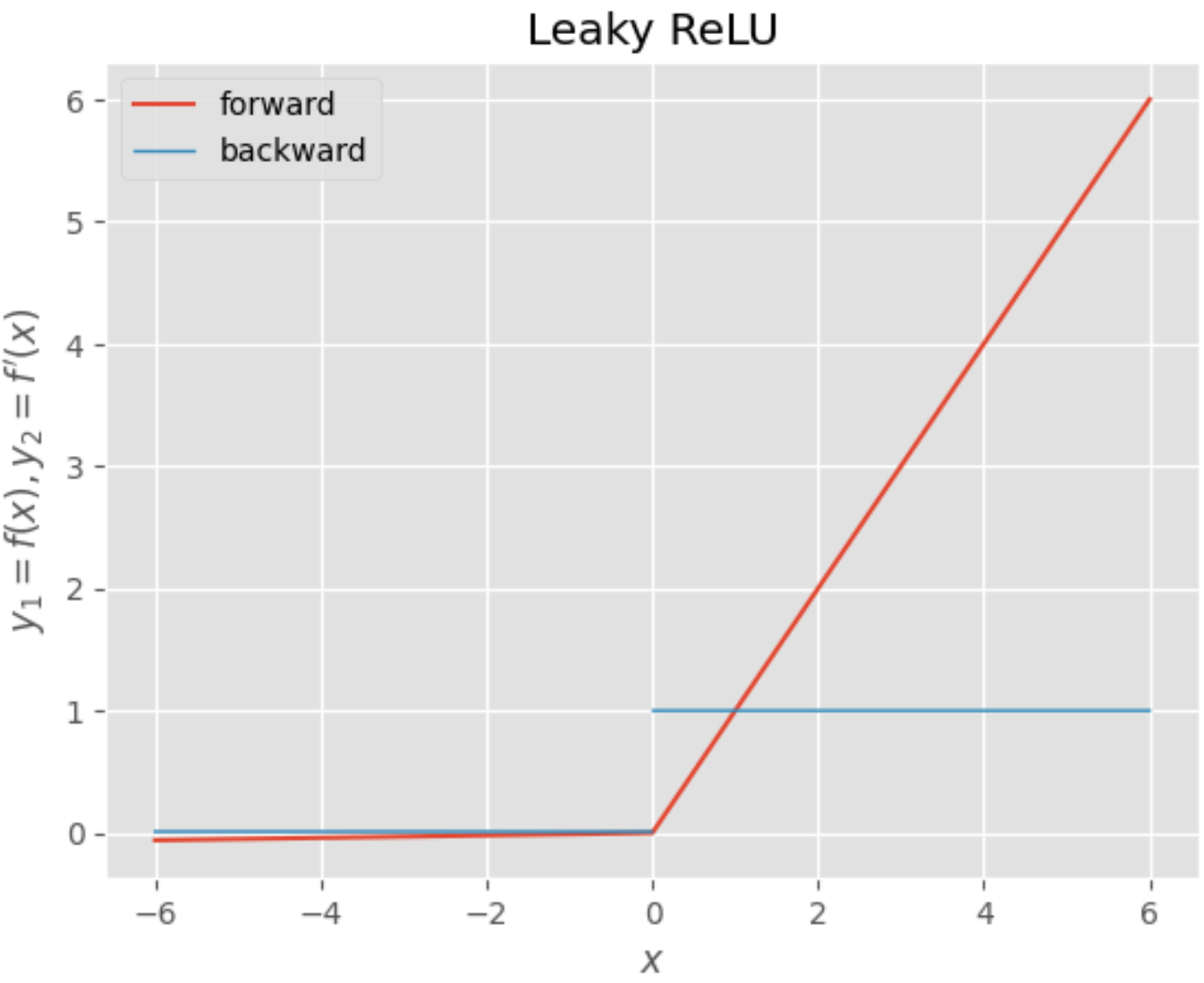
- ReLU
  正規化線形関数、ランプ関数。勾配消失が発生しにくいが入力値が負のとき、学習が進まない。

```python
def relu(x):
    """forward
        ReLU
        正規化線形関数
    """
    return np.maximum(0, x)

def d_relu(x):
    """backward
        derivative of ReLU
        正規化線形関数の導関数
    """
    dx = np.where(x > 0.0, 1.0, np.where(x < 0.0, 0.0, np.nan))
    return dx
```



- Leaky ReLU
  漏洩正規化線形関数。入力値が負のときは、小さな傾きの 1 次関数となる。

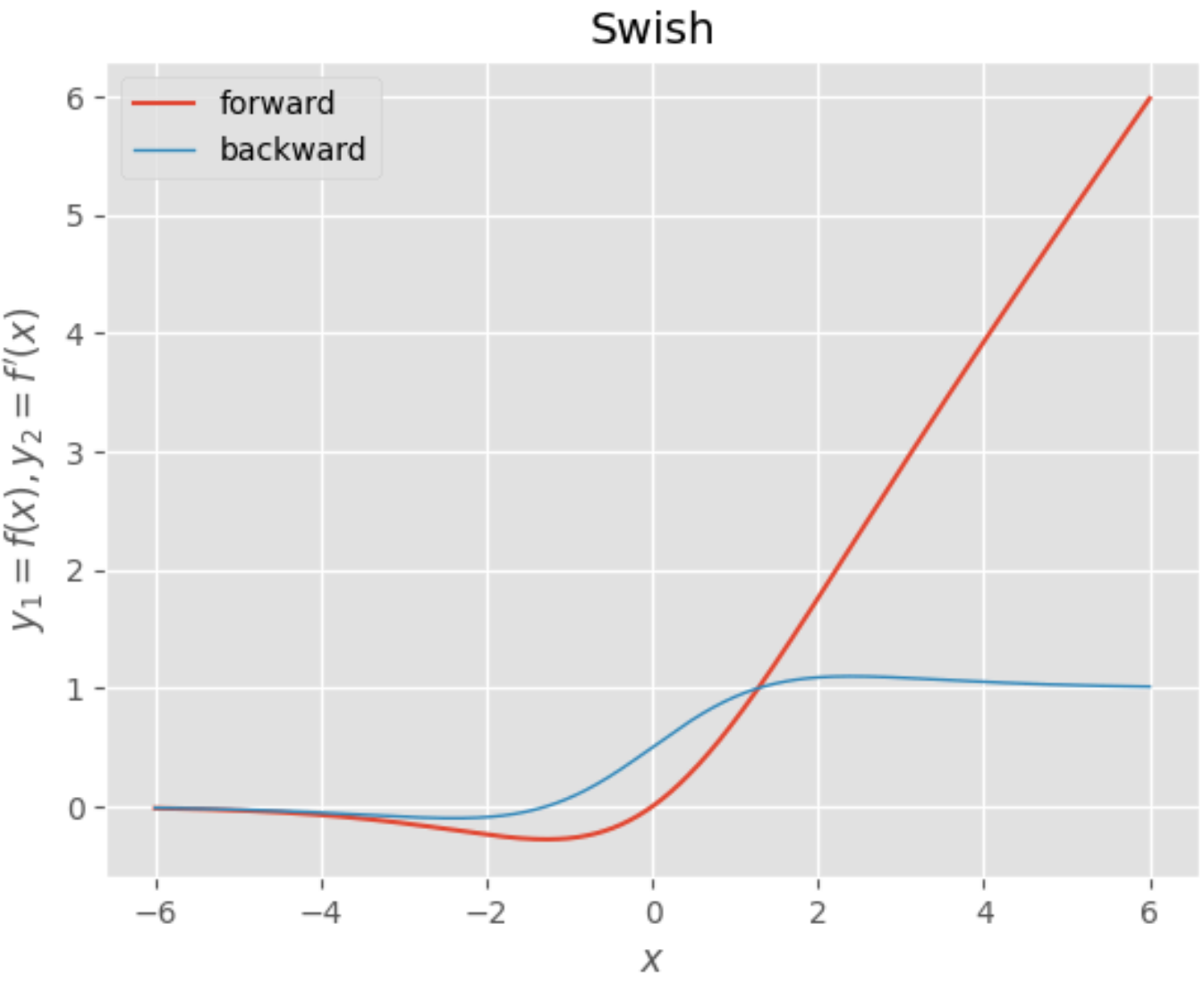
```
alpha = 0.01

def lrelu(x):
    """forward
        Leaky ReLU
    漏洩正規化線形関数
    """
    return np.maximum(alpha*x, x)

def d_lrelu(x):
    """backward
        derivative of Leaky ReLU
        漏洩正規化線形関数の導関数
    """
    dx = np.where(x > 0.0, 1.0, np.where(x < 0.0, alpha, np.nan))
    return dx
```



- Swish
  シグモイド加重線形関数。ReLU の代替候補として注目されている。ReLU と違って原点におい

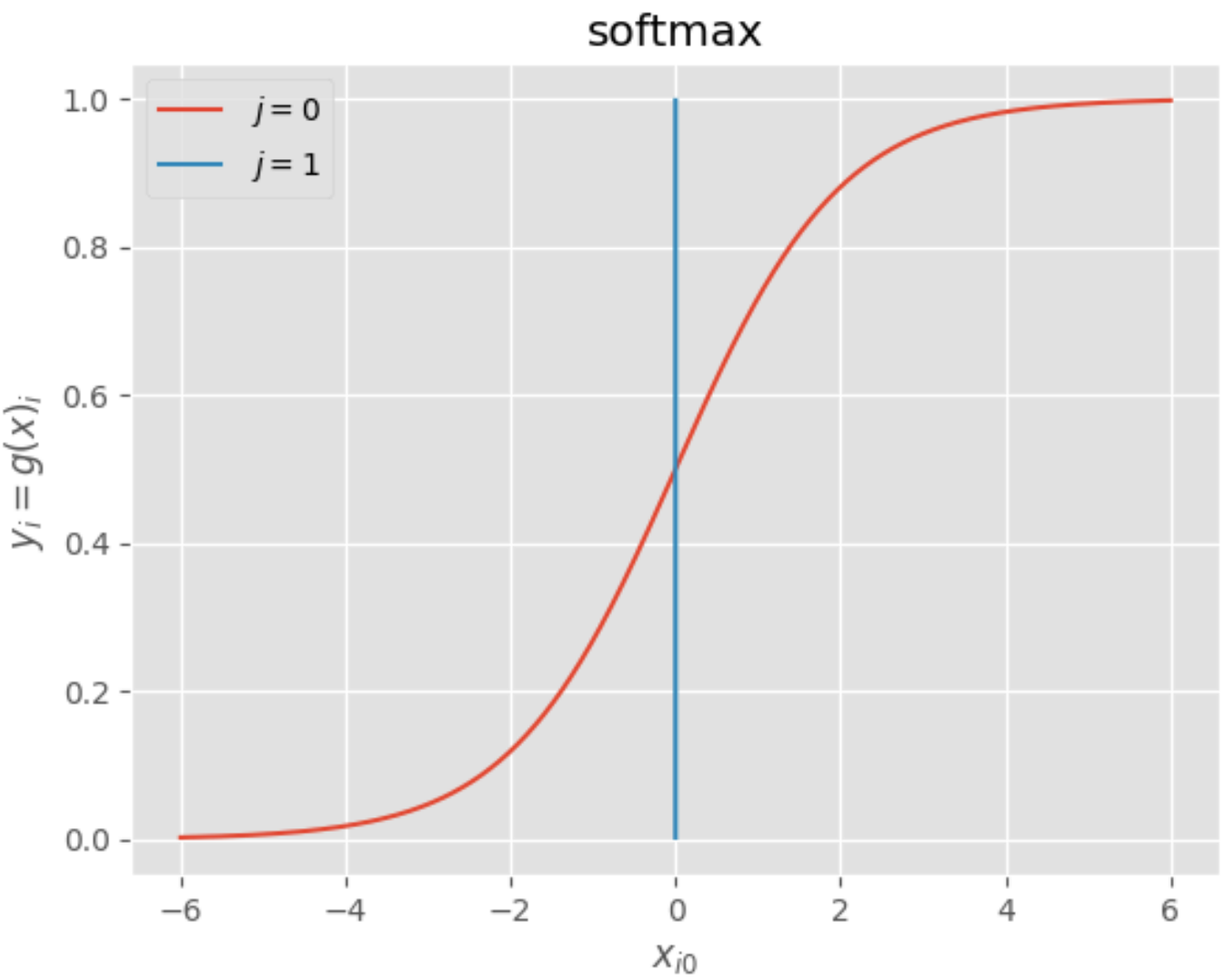
ても連続になる。

```
beta = 1.0

def swish(x):
    """forward
        Swish
        シグモイド加重線形関数
    """
    return x * sigmoid(beta*x)

def d_swish(x):
    """backward
        derivative of Swish
        シグモイド加重線形関数の導関数
    """
    dx = beta*swish(x) + sigmoid(beta*x)*(1.0 - beta*swish(x))
    return dx
```



- Softmax
  シグモイド関数を多値分類用に拡張したもの

```
def softmax(x):
    """forward
        softmax
        ソフトマックス関数
    """
    if x.ndim == 2:
        x = x.T
        x = x - np.max(x, axis=0)
        y = np.exp(x) / np.sum(np.exp(x), axis=0)
        return y.T

    x = x - np.max(x)  # オーバーフロー対策
    return np.exp(x) / np.sum(np.exp(x))

def d_softmax(x):
    """backward
        derivative of softmax
        ソフトマックス関数の導関数
    """
    y = softmax(x)
    dx = -y[:,:,None] * y[:,None,:]        # ヤコビ行列を計算（i≠jの場合）
    iy, ix = np.diag_indices_from(dx[0])  # 対角要素の添字を取得
    dx[:,iy,ix] = y * (1.0 - y)            # 対角要素値を修正（i＝jの場合）
    return dx
```


softmax
$j = 0$
$j = 1$
1.0
0.8
0.6
0.4
0.2
0.0
$y_i = g(x)_i$
−6
−4
−2
0
2
4
6
x_{i0}