通信理論に特化した深層学習
第8回ゼミ資料
Kerasレイヤーの作り方

豊橋技術科学大学
電気・電子情報工学系
准教授　竹内啓悟

少しだけ発展的な学習例

MNISTデータ（教師データ）

0～9の手書き数字の答え付画像データ（28×28ピクセル、ピクセル値0～255）

訓練用データ60000個、評価用データ10000個

学習目標

未知の画像データの正答率が最大になるように、T層全結合型順伝播ネットワークの中間層のバイアスを任意の値に固定し、重みのみを学習したい。

ソースコードの作成手順

- バイアスの初期値を設定し、学習対象外とするレイヤーを独自に定義する。
- KerasのFunctional APIを使って層数が一般のネットワークを作成する。

自作コードのインポート方法1（基本編）

plot_weights関数とplot_history関数のような、何度も使用する関数を一元管理したい。

手順1

plot_weights関数とplot_history関数の定義（第7回資料参照）を./tools/output.pyに記述する。（ディレクトリ名やファイル名は任意）

手順2

これらの関数を使用するソースコードを.//に置き、その冒頭に以下のインポート文を記述する。

```
from tools.output import plot_weights, plot_history
```

「/」ではなく「.」。拡張子「.py」は省略できる。

自作コードのインポート方法2（発展編）

クラスを定義して、plot_weights関数やplot_history関数をインポートする。

手順1　./tools/output_class.pyに以下を記述する。（ファイル名等は任意）

```
class Output():
    def plot_weights(self, weights):
        以下は同様

    def plot_history(self, training_history):
        以下は同様
```

手順2　このクラスを使用するソースコード（主コードと呼ぶ）を.//に置き、冒頭のimport文の後に以下を記述する。

```
from tools.output_class import Output
out = Output()
```

手順3　主コードで、plot_weights(weights)とplot_history(training_history)をout.plot_weights(weights)とout.plot_history(training_history)に変更する。

自作Kerasレイヤーの概要

三つのメソッド__init__、build、callを持つクラスMyLayerを定義する。

```
class MyLayer(tf.keras.layers.Layer):
    def __init__(self, units, init_bias):
        ・・・・

    def build(self, input_shape):
        ・・・・

    def call(self, input):
        ・・・・
```

- __init__はコンストラクタと呼ばれるPython特有のメソッド
- __build__はパラメータを定義するためのメソッド
- __call__はレイヤーを定義するためのメソッド

コンストラクタ

```
class MyLayer(tf.keras.layers.Layer):
    def __init__(self, units, init_bias):
        super().__init__()
        self.units = units
        self.init_bias = init_bias
```

units: MyLayerのユニット数を表す独自のパラメータ

init_bias: バイアスの初期値を指定する長さunisのリスト

super()を使って、サブ（子）クラスであるMyLayerから、スーパー（親）クラスであるtf.keras.layers.Layerのメソッドを呼び出している。

selfとは何か？

ユニット数が10のインスタンスlayer = MyLayer(10, init_bias)を生成した際のインスタンスlayerのことだと思えばよい。

buildメソッド

入力　input_shape = (ミニバッチサイズ, 入力データ次元)

各入力を行ベクトルで扱うことに注意

機能（入力次元）×unitsの重み行列と次元がunitsに等しいバイアスとを定義する。重み行列のみを学習対象とする。

```
def build(self, input_shape):
    #重みの定義
    self.kernel = self.add_weight(
      "kernel", shape = (int(input_shape[-1]), self.units),
      initializer = 'glorot_uniform', trainable=True)
    #バイアスの定義
    self.bias = self.add_weight(
      "bias", shape = (self.units),
      initializer = tf.keras.initializers.Constant(self.init_bias),
      trainable = False)
```

Input_shape[-1]はタプルの最後の要素（入力データ次元）を表す。

callメソッド

入力行ベクトル$\boldsymbol{x}$=input、重み行列$\boldsymbol{W}$=self.kernel、バイアス行ベクトル$\boldsymbol{b}$=self.biasに対して、以下を計算する。

$$f_{\mathrm{ReLU}}(\boldsymbol{x}\boldsymbol{W}+\boldsymbol{b})$$

f_{ReLU}は正規化線形関数を表す。

```
def call(self, input):
    output = tf.matmul(input, self.kernel)
    output += self.bias
    return tf.nn.relu(output)
```

活性化関数は要素ごとに適用される。

変数はテンソルなので、基本演算でもTensorFlowのメソッドを使用する必要がある。

確認

MyLayerクラスを定義した後に、以下を実行せよ。

```
input = tf.constant([[1., 0., 0.], [0., 1., 0.]])
UNITS = 5
init_bias = [0, 0, 0, 0, 0]
layer = MyLayer(UNITS, init_bias)
print(input)
print(layer(input))
print(layer.variables)
```

入力データベクトル(1, 0, 0)と(0, 1, 0)に対する出力が、それぞれ重み行列の1行目と2行目に正規化線形関数を施したものであることを確認せよ。

出力結果の例

```
#input
tf.Tensor(
[[1. 0. 0.]
 [0. 1. 0.]], shape=(2, 3), dtype=float32)

#layer(input)
tf.Tensor(
[[0.46303207 0.         0.         0.         0.68312186]
 [0.         0.         0.         0.         0.56542426]], shape=(2, 5), dtype=float32)

#layer.variables
[<tf.Variable 'my_layer/kernel:0' shape=(3, 5) dtype=float32, numpy=
array([[ 0.46303207, -0.47431585, -0.77928585, -0.64642787,  0.68312186],
       [-0.37205958, -0.48499542, -0.31127828, -0.22867483,  0.56542426],
       [-0.30075848,  0.14197868, -0.4037789 ,  0.31804734,  0.49709147]],
      dtype=float32)>, <tf.Variable 'my_layer/bias:0' shape=(5,) dtype=float32,
numpy=array([0., 0., 0., 0., 0.], dtype=float32)>]
```

中間層の初期バイアス

```
LAYERS = 3
UNITS = 128

init_bias = []
for t in range(LAYERS - 1):
    init_bias_t = []
    for n in range(UNITS):
        init_bias_t.append(t*n)
    init_bias += [init_bias_t]

for t in range(LAYERS - 1):
    print(init_bias[t])
```

どんな初期バイアスを指定したかを確認せよ。

Kerasのfunctional API

前回のmodel = tf.keras.models.Sequentialを以下に置き換えよ。

```
inputs = tf.keras.Input(shape = (28, 28))
#入力データのフォーマットを指定
x = tf.keras.layers.Flatten()(inputs)

for t in range(LAYERS - 1):
    x = MyLayer(UNITS, init_bias[t])(x)
#層数がLAYERSになるように、中間層を定義

outputs = tf.keras.layers.Dense(10, activation = 'softmax')(x)

model = tf.keras.Model(inputs = inputs, outputs = outputs)
```

ネットワークの定義にfor文が使える。

学習後も中間層のバイアスは変更されないことを確認せよ。

model.summary()の実行結果

```
Model: "model"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
input_1 (InputLayer)         [(None, 28, 28)]          0
_________________________________________________________________
flatten (Flatten)            (None, 784)               0
_________________________________________________________________
my_layer (MyLayer)           (None, 128)               100480
_________________________________________________________________
my_layer_1 (MyLayer)         (None, 128)               16512
_________________________________________________________________
dense (Dense)                (None, 10)                1290
=================================================================
Total params: 118,282
Trainable params: 118,026
Non-trainable params: 256
_________________________________________________________________
```