1　重要事項

各特徴量に正解/不正解のラベル 0/1 が付いている二値分類問題である。ただし、0 のラベルが 1 より圧倒的に多い不均衡データである。何も対処せずそのまま学習すると 0 しか出力しないモデルになって、accuracy は高いが全く使えないモデルになる。

実行した不均衡データへの対処法は次の通り。

- 評価指標を Precision, Recall を中心に見ることで 1 の予測の精度を見る。
- 学習時のパラメータ更新で 1 のデータによる更新を重くする。
- アンダーサンプリング

Precision は 1 と予測した中での正解した割合。Recall は正解が 1 の中で 1 と予測できた割合。詳しいことは検索してください。

パラメータ更新を重くするというのは、正解が 1 のデータが非常に少ないので、そのデータの学習時の逆伝播によるパラメータ更新を相対的に大きくするということ。

アンダーサンプリングは試しても精度の改善が見られなかったので、現在採用しているモデルの学習には使っていない。

zip ファイルを展開してできるフォルダが学習済みのモデルを保存したもの。keras の load_model() メソッドで読み込める。

以下学習結果に関して。

学習を始めると、Precision は比較的緩やかに減少していく。Recall は最初に大きく増加した後に増加が緩やかになる。

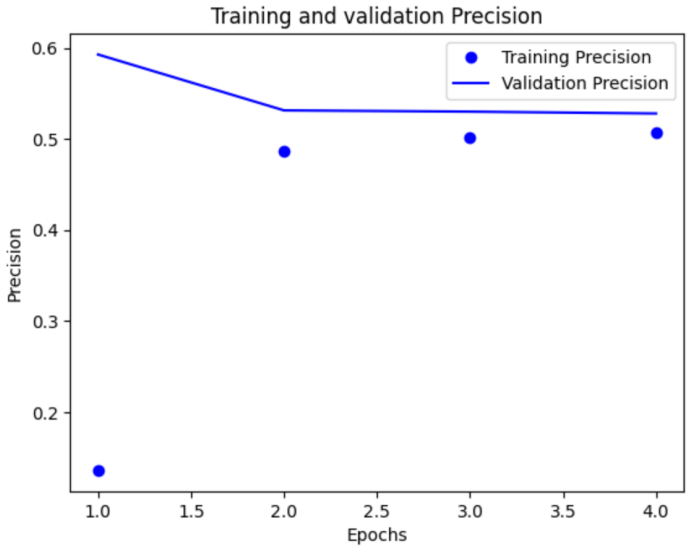

図 1　Precision

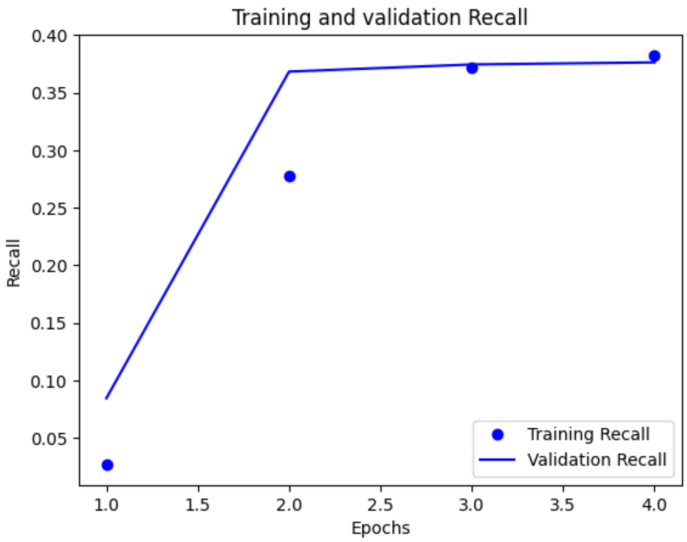

図 2　Recall

エポック数が少ない段階で学習は打ち切られる。最終的に Precision は 0.5 程度、Recall は 0.35 程度になった。データが 1 に分類された元でそれが正解である確率は 0.5 で、データのラベルが 1 である時にそれが 1 に分類される確率は 0.35 といった状態。

Recall は増加が続くので 1 の予測数自体は増えていることが分かるが、Precision は緩やかに減少し続けるので、1 の予測の正解率は低下していく。学習を継続しても雑な 1 の予測が増え続けるだけだと考えて、最初に Recall が大きく改善された段階で学習を打ち切った。(より正確には、Precision の改善が 3 エポック連続で起こらない場合に学習を止める設定にすることで、間接的に Recall の改善が弱まった段階で止まるようになった)

以下コードの説明が続くが、どうしてもモデルの学習を実行したい人以外読まなくていいと思う。

2　コードまとめ

全部一か所に張り付けるのではなく、分けて実行した方がいい。パスやファイル名は各自設定。各部分の詳細は次セクションから。

```
#XML ファイルの処理・データセットの保存
import dataset\_processor as dp

pr = dp.DatasetProcessor(
  dirpath='/content/drive/MyDrive/実験 2/prj-14/mj_files/2012',
  save_path='/content/drive/MyDrive/実験 2/prj-14/mj_files',
  max_file=200
)

pr.process_dataset()

pr.save_as_pkl('data200.pkl', 'label200.pkl')


#データの読み込み
import pickle
from sklearn.model_selection import train_test_split

filepath1 = '/content/drive/MyDrive/実験 2/prj-14/mj_files/data200.pkl'
filepath2 = '/content/drive/MyDrive/実験 2/prj-14/mj_files/label200.pkl'

f1 = open(filepath1, 'rb')
f2 = open(filepath2, 'rb')

mj_data = pickle.load(f1)#特徴量ベクトル
mj_target = pickle.load(f2)#ラベルベクトル

train_data, validation_data, train_label, validation_label = train_test_split(
  mj_data,
  mj_target,
  test_size = 0.2
```

```
)
```

#モデル定義

```
import numpy as np
import tensorflow as tf
from tensorflow import keras

model1 = keras.Sequential(
  [
      keras.layers.Dense(64, activation='relu', input_shape=(3209,)),
      keras.layers.Dropout(0.5),
      keras.layers.Dense(32, activation='relu'),
      keras.layers.Dropout(0.5),
      keras.layers.Dense(1, activation='sigmoid')
  ]
)

model1.compile(
  optimizer = 'adam',
  loss = 'binary_crossentropy',
  metrics = [
      'accuracy',
      #正解率
      keras.metrics.F1Score(average='weighted', threshold=0.5),
      #F1 値による評価
      keras.metrics.AUC(curve='PR', name='auc1'),
      #PR-AUC による評価
      keras.metrics.Precision(name='pre1'),
      #Precision による評価
      keras.metrics.Recall(name='rec1')
      #Recall による評価
  ]
)

print(model1.summary())
```

```
#学習
class_weight = {
  0: 1.0,
  1: 5.0
}

history1 = model1.fit(
  x = train_data,
  y = train_label,
  class_weight = class_weight,
  epochs = 100,
  batch_size = 2048,
  validation_data = (validation_data, validation_label),
  callbacks = [
      keras.callbacks.EarlyStopping(monitor='val_auc1', patience=3, mode='max'),
      keras.callbacks.EarlyStopping(monitor='val_pre1', patience=3, mode='max')
  ]
)
```

3 データ処理

3.1 コード全体

XML ファイルから numpy 配列を作成する部分。

```
import dataset\_processor as dp

pr = dp.DatasetProcessor(
  dirpath='/content/drive/MyDrive/実験 2/prj-14/mj_files/2012',
  save_path='/content/drive/MyDrive/実験 2/prj-14/mj_files',
  max_file=200
)

pr.process_dataset()
```

```
pr.save_as_pkl('data200.pkl', 'label200.pkl')
```

3.2 詳細

「dataset_processor.py」を import する。このファイルに特徴量やラベルを処理して numpy 配列を作成するインターフェース部分が入っている。

```
import dataset_processor as dp
```

dataset_processor に記述されているクラス「DatasetProcessor」をインスタンス化する。
以下引数の説明。

- dirpath
  XML ファイルのあるディレクトリのパスを与える。必須。
- save_path
  作成した numpy 配列をファイルに保存するときの保存先ディレクトリのパスを与える。任意。デフォルトでは dirpath を指定したことになる。
- max_file
  処理するファイルの数を指定する。任意。デフォルトでは 100。Google Colab で実行するとき、あまり数字を大きくしすぎると RAM がパンクするので注意。現状だと 200 程度が限度と思われる。これ以上だとデータが RAM に収まっても、モデルがメモリ不足で実行できない可能性がある。

```
pr = dp.DatasetProcessor(
  dirpath='/content/drive/MyDrive/実験 2/prj-14/mj_files/2012',
  save_path='/content/drive/MyDrive/実験 2/prj-14/mj_files',
  max_file=200
)
```

DatasetProcessor の「process_dataset()」メソッドを実行する。この中で arrannge_dataset.py, make_dataset.py, make_label_vec.py の処理が行われる。

```
pr.process_dataset()
```

実行が正常に終わったことの確認。データセットのアクセス方法は、入力データは「インスタンス変数.data」、ラベルは「インスタンス変数.label」。以下の例ではデータ数 191548 となっていることが分かる。

```
  print(pr.data.shape)
  print(pr.label.shape)
---------------------------------------
  (191548, 3209)
  (191548,)
```

作成した numpy 配列をファイルに保存するには、「save_as_pkl()」を実行する。pickle ファイルで保存するようになっている。

以下引数の説明。

- data_file
  特徴量ベクトルを保存するファイル名 (パスは不要)。
- label_file
  ラベルベクトルを保存するファイル名 (パスは不要)。

```
  pr.save_as_pkl('data200.pkl', 'label200.pkl')
```

4 データ読み込み

4.1 コード全体

ファイルに保存したデータセットをメモリに読み出す部分。

```
  import pickle
  from sklearn.model_selection import train_test_split

  filepath1 = '/content/drive/MyDrive/実験 2/prj-14/mj_files/data200.pkl'
  filepath2 = '/content/drive/MyDrive/実験 2/prj-14/mj_files/label200.pkl'

  f1 = open(filepath1, 'rb')
  f2 = open(filepath2, 'rb')

  mj_data = pickle.load(f1)#特徴量ベクトル
  mj_target = pickle.load(f2)#ラベルベクトル
```

```
train_data, validation_data, train_label, validation_label = train_test_split(
  mj_data,
  mj_target,
  test_size = 0.2
)
```

4.2 詳細

始めに必要なものを import する。

```
import pickle
from sklearn.model_selection import train_test_split
```

pickle ファイルから numpy 配列を復元する。変数 filepath1, filepath2 は特徴量とラベルそれぞれの pickle ファイルのパスを入れている。

それぞれのファイルを open() で開く。バイナリなので第二引数には'rb' を指定。

numpy 配列の復元には「pickle.load()」を使う。以下の例では mj_data に特徴量、mj_target にはラベルが復元されている。

```
filepath1 = '/content/drive/MyDrive/実験 2/prj-14/mj_files/data200.pkl'
filepath2 = '/content/drive/MyDrive/実験 2/prj-14/mj_files/label200.pkl'

f1 = open(filepath1, 'rb')
f2 = open(filepath2, 'rb')

mj_data = pickle.load(f1)#特徴量ベクトル
mj_target = pickle.load(f2)#ラベルベクトル
```

読み込んだ numpy 配列のデータセットを学習用と検証用に分割する。データセット全体の 2 割を検証データにあてている。

```
train_data, validation_data, train_label, validation_label = train_test_split(
  mj_data,
  mj_target,
  test_size = 0.2
)
```

5　モデルの学習

5.1　コード全体

仕様の詳細は Keras のドキュメント参照。Google Colab でセルを分けて順に実行することを推奨する。

```
import numpy as np
import tensorflow as tf
from tensorflow import keras

model1 = keras.Sequential(
  [
      keras.layers.Dense(64, activation='relu', input_shape=(3209,)),
      keras.layers.Dropout(0.5),
      keras.layers.Dense(32, activation='relu'),
      keras.layers.Dropout(0.5),
      keras.layers.Dense(1, activation='sigmoid')
  ]
)

model1.compile(
  optimizer = 'adam',
  loss = 'binary_crossentropy',
  metrics = [
      'accuracy',
      #正解率
      keras.metrics.F1Score(average='weighted', threshold=0.5),
      #F1 値による評価
      keras.metrics.AUC(curve='PR', name='auc1'),
      #PR-AUC による評価
      keras.metrics.Precision(name='pre1'),
      #Precision による評価
      keras.metrics.Recall(name='rec1')
      #Recall による評価
  ]
```

```
)

print(model1.summary())


class_weight = {
  0: 1.0,
  1: 5.0
}

history1 = model1.fit(
  x = train_data,
  y = train_label,
  class_weight = class_weight,
  epochs = 100,
  batch_size = 2048,
  validation_data = (validation_data, validation_label),
  callbacks = [
      keras.callbacks.EarlyStopping(monitor='val_auc1', patience=3, mode='max'),
      keras.callbacks.EarlyStopping(monitor='val_pre1', patience=3, mode='max')
  ]
)
```

5.2 詳細

始めに Keras の Sequential() でモデルのインスタンスを定義する。中間層は 2 層で、ノード数がそれぞれ 64, 32。活性化関数は ReLU。確率 0.5 で Dropout を入れている。出力層はノード数 1 で活性化関数はシグモイド関数。

```
model1 = keras.Sequential(
  [
      keras.layers.Dense(64, activation='relu', input_shape=(3209,)),
      keras.layers.Dropout(0.5),
      keras.layers.Dense(32, activation='relu'),
      keras.layers.Dropout(0.5),
      keras.layers.Dense(1, activation='sigmoid')
  ]
```

```
)
```

↑で定義したモデルの学習アルゴリズムや評価指標を指定する。インスタンスから compile() を実行する。重要な部分は metrics で、keras の評価関数のリストを与える。以下評価関数の説明。Precision, Recall の詳細は検索してください。

- accuracy 普通の正解率。データが不均衡なのであまり使えない。
- F1Score Precision と Recall の調和平均で、両方を総合して値を出す。閾値を 0.5 にしている。
- AUC PR-AUC の値。当初はこれをメインで使おうと考えていたが、PR 曲線を描く処理を作るのが非常に手間なのであまり使えないと思う。
- Precision 適合率。1 と予測した中で正解している割合。予測の閾値は 0.5 にしている。
- Recall 再現率。正解が 1 である中で、1 と予測できている割合。予測の閾値は 0.5 にしている。

model.summary() で作成したモデルの内容を表示できる。

```
model1.compile(
  optimizer = 'adam',
  loss = 'binary_crossentropy',
  metrics = [
      'accuracy',
      #正解率
      keras.metrics.F1Score(average='weighted', threshold=0.5),
      #F1 値による評価
      keras.metrics.AUC(curve='PR', name='auc1'),
      #PR-AUC による評価
      keras.metrics.Precision(name='pre1'),
      #Precision による評価
      keras.metrics.Recall(name='rec1')
      #Recall による評価
  ]
)

print(model1.summary())
```

モデルの学習部分。class_weight には、学習時のパラメータ更新に重みをかけるが、その割合を指定している。ラベルが 1 のデータの逆伝播によるパラメータ更新はラベルが 0 のデータのパラメータ更新より 5 倍大きくなる。

インスタンスの fit() メソッドで学習が始まる。評価指標の動きなどの結果は history1 に格納さ

れる。

重要な部分は引数の callbacks で、EarlyStopping() を与えている。これは指定した評価指標が変動しなくなった時に学習を終了させる機能。監視する評価指標は AUC と Precision になっているが、Precision の方がメインになる。

```
class_weight = {
  0: 1.0,
  1: 5.0
}

history1 = model1.fit(
  x = train_data,
  y = train_label,
  class_weight = class_weight,
  epochs = 100,
  batch_size = 2048,
  validation_data = (validation_data, validation_label),
  callbacks = [
      keras.callbacks.EarlyStopping(monitor='val_auc1', patience=3, mode='max'),
      keras.callbacks.EarlyStopping(monitor='val_pre1', patience=3, mode='max')
  ]
)
```

6 モデルの評価

model4.py の matplotlib の plot をたくさん使っているところでグラフが描ける。ここはそのままコピペしていいと思う。詳細は matplotlib のドキュメント参照。