

November 4, 2020

## 1 Part 0 Python Environment Set-up

This project is completed using Python. There are two ways of setting up the Python environment.

1. Colab (on virtual machine)
  - Set up Google Colaboratory on lionmail drive, upload the notebook and open it with Colab.
  - If you choose to run our notebook on Colab, there is no need install any library.
  - Before training the model, please upload `fiducial_pt_full.pkl` and `label_full.pkl`.
2. Jupyter Notebook (on local machine)
  - Download Anaconda from <https://www.anaconda.com/products/individual>
  - Open the notebook with Jupyter Notebook
  - Install libraries that do not come with Anaconda: `!pip install tensorflow`, `!pip install scikit-learn`, `!pip install tensorflow_hub`. (or any other required libraries)

## 2 Part I Data Pre-Processing

### 2.1 1. Read fiducial point and save as pickle files

```
[1]: import scipy.io
import os
import numpy as np
import pandas as pd
import pickle
import tensorflow as tf
import warnings
warnings.filterwarnings('ignore')
```

```
[2]: def get_points(file):
    '''load matlab style file'''
    mat = scipy.io.loadmat(file)
    return mat[list(mat.keys())[3]]

def pickle_save(filename, content):
    '''save the file into python pickle object under output folder'''
    with open(filename, 'wb') as f:
        pickle.dump(content, f)
```

```
def pickle_open(filename):
    '''load the pickle file'''
    with open(filename, 'rb') as f:
        content = pickle.load(f)
    return content
```

```
[ ]: ## No need to run this cell if pickle files are uploaded
```

```
from google.colab import drive
drive.mount('/content/drive')
```

Mounted at /content/drive

```
[ ]: ## No need to run this cell if pickle files are uploaded
```

```
# read mat format file into pickle
dir_list = os.listdir('drive/My Drive/points')
dir_list.sort()

fiducial_pt_full = np.stack((get_points('drive/My Drive/points/'+filename) for
    ↪filename in dir_list))
label_full = pd.read_csv('label.csv')['label']

# save the data into pickle files, so that we don't need to read raw data
    ↪everytime
pickle_save('fiducial_pt_full', fiducial_pt_full)
pickle_save('label_full', label_full)
```

## 2.2 2. Train-test split

```
[3]: from sklearn.model_selection import train_test_split
```

```
# load data from pickle object
fiducial_pt_full = pickle_open('fiducial_pt_full.pkl')
label_full = pickle_open('label_full.pkl')

## Note: randomly split into training & test set with seed 42
X_train, X_test, y_train, y_test = train_test_split(fiducial_pt_full,
    ↪label_full, test_size=0.2, random_state=42)
```

```
[4]: X_train.shape, y_test.shape
```

```
[4]: ((2400, 78, 2), (600,))
```

## 2.3 3. Feature Construction

Keep the fiducial points on eyes, eyebrows, and lips. Compute the pairwise distances.

```
[10]: from sklearn.metrics import pairwise_distances
import time

# extract pairwise distance as features (78*77/2=3003 features)
# nrow=number of records of the dataset; ncol=3003
# keep 48 points on lips, eyes, and eyebrows, 48*47

start_time = time.time()
feature_train = np.stack((pairwise_distances(X_train[i])[np.triu_indices(78, k=
    ↳1)] for i in range(X_train.shape[0]))
print('Baseline training feature extraction takes %s seconds.'%round((time.
    ↳time()-start_time),3))

start_time = time.time()
feature_test = np.stack((pairwise_distances(X_test[i])[np.triu_indices(78, k =
    ↳1)] for i in range(X_test.shape[0]))
print('Baseline training feature extraction takes %s seconds.'%round((time.
    ↳time()-start_time),3))
```

Baseline training feature extraction takes 1.049 seconds.

Baseline training feature extraction takes 0.263 seconds.

## 3 Part II Baseline Model: GBT

We optimized the baseline model with tuned parameters. Referencing main\_draft.ipynb for more information.

```
[11]: from sklearn.ensemble import GradientBoostingClassifier
from sklearn.metrics import precision_score, recall_score, roc_auc_score,
    ↳accuracy_score
import time

# define function to report metrics for baseline model
def clf_metrics(y_true, y_pred, y_score):
    accuracy = accuracy_score(y_true, y_pred)

    # reweight dataset in order to estimate the accuracy of "balanced" data set
    weight_data = np.zeros(len(y_true))
    for v in np.unique(y_true):
        weight_data[y_true==v] = 0.5*len(y_true)/np.sum(y_true==v)
    weighted_acc = np.sum(weight_data * (y_pred==y_true)/np.sum(weight_data))

    precision = precision_score(y_true, y_pred)
    recall = recall_score(y_true, y_pred)
```

```

auc = roc_auc_score(y_true, y_score, average='weighted')

df = pd.DataFrame({'accuracy': [accuracy], 'weighted acc': [weighted_acc],
                    'precision': [precision], 'recall': [recall], 'auc':
→ [auc]})
print(df)

```

```

[34]: # assign weights for data to remedy the imbalanced training data
weights2 = np.zeros(len(y_train))
weights2[y_train == 0] = 1
weights2[y_train == 1] = 10

gbt_baseline = GradientBoostingClassifier(learning_rate=0.1, n_estimators=150,
                                         max_depth=4, min_samples_split=2,
→ min_samples_leaf=1,
                                         subsample=0.8, max_features='sqrt',
→ random_state=242)
start_baseline = time.time()
gbt_baseline.fit(feature_train, y_train, sample_weight = weights2)
end_baseline = time.time()
print('Training time cost {:.2f} s'.format(end_baseline-start_baseline))

```

Training time cost 7.13 s

```

[36]: # store the trained model
pickle_save('gbt_baseline.pkl', gbt_baseline)

```

```

[37]: # load the trained model from file
gbt_baseline = pickle_open('gbt_baseline.pkl')

```

```

[18]: gbt_baseline.get_params()

```

```

[18]: {'ccp_alpha': 0.0,
      'criterion': 'friedman_mse',
      'init': None,
      'learning_rate': 0.1,
      'loss': 'deviance',
      'max_depth': 4,
      'max_features': 'sqrt',
      'max_leaf_nodes': None,
      'min_impurity_decrease': 0.0,
      'min_impurity_split': None,
      'min_samples_leaf': 1,
      'min_samples_split': 2,
      'min_weight_fraction_leaf': 0.0,
      'n_estimators': 150,
      'n_iter_no_change': None,

```

```
'presort': 'deprecated',
'random_state': 299,
'subsample': 0.8,
'tol': 0.0001,
'validation_fraction': 0.1,
'verbose': 0,
'warm_start': False}
```

```
[90]: # Gradient boosting baseline model performance
pred_train = gbt_baseline.predict(feature_train)
score_train = gbt_baseline.decision_function(feature_train)
print('Training set:')
clf_metrics(y_train, pred_train, score_train)
print('\n')

test_start_baseline = time.time()
pred_test = gbt_baseline.predict(feature_test)
score_test = gbt_baseline.decision_function(feature_test)
test_end_baseline = time.time()
print('Test set:')
clf_metrics(y_test, pred_test, score_test)
print('\n')
print('Testing takes {:.2f} seconds'.
      ↪format(test_end_baseline-test_start_baseline))
```

Training set:

	accuracy	weighted acc	precision	recall	auc
0	0.964167	0.977846	0.842202	1.0	0.999891

Test set:

	accuracy	weighted acc	precision	recall	auc
0	0.778333	0.704997	0.519737	0.568345	0.789447

Testing takes 0.02 seconds

## 4 Part III Our Model: Densely Connected Neural Network

### 4.1 1. Feature engineering

```
[39]: import warnings
warnings.filterwarnings('ignore')
start_time = time.time()
feature_train = np.stack((pairwise_distances(X_train[i])[np.triu_indices(78, k_
↪= 1)] for i in range(X_train.shape[0])))
```

```

print('Training feature extraction takes %s seconds.'%round((time.
↪time()-start_time),3))

start_time = time.time()
feature_test = np.stack((pairwise_distances(X_test[i])[np.triu_indices(78, k =
↪1)] for i in range(X_test.shape[0])))
print('Test feature extraction takes %s seconds.'%round((time.
↪time()-start_time),3))

```

Training feature extraction takes 1.059 seconds.  
Test feature extraction takes 0.295 seconds.

## 4.2 2. Oversampling the minority class

```

[40]: y_train = np.array(y_train)
      y_test = np.array(y_test)

      bool_train_labels = y_train != 0
      pos_features = feature_train[bool_train_labels]
      neg_features = feature_train[~bool_train_labels]

      pos_labels = y_train[bool_train_labels]
      neg_labels = y_train[~bool_train_labels]

```

```

[41]: pos_labels.shape, neg_labels.shape

```

```

[41]: ((459,), (1941,))

```

```

[78]: # Balance the dataset manually by choosing the right number of random indices
      ↪from the positive examples
      RANDOM_SEED = 111
      np.random.seed(RANDOM_SEED)

      ids = np.arange(len(pos_features))
      choices = np.random.choice(ids, len(neg_features))

      res_pos_features = pos_features[choices]
      res_pos_labels = pos_labels[choices]

      res_pos_features.shape

```

```

[78]: (1941, 3003)

```

```

[79]: resampled_features = np.concatenate([res_pos_features, neg_features], axis=0)
      resampled_labels = np.concatenate([res_pos_labels, neg_labels], axis=0)

      order = np.arange(len(resampled_labels))

```

```

np.random.shuffle(order)
resampled_features = resampled_features[order]
resampled_labels = resampled_labels[order]

resampled_features.shape, resampled_labels.shape

```

[79]: ((3882, 3003), (3882,))

### 4.3 3. Model Training

```

[80]: from tensorflow import keras

METRICS = [
    keras.metrics.BinaryAccuracy(name='accuracy'),
    keras.metrics.AUC(name='auc'),
    keras.metrics.Precision(name='precision'),
    keras.metrics.Recall(name='recall')
]
INPUT_SHAPE=[3003]
model1 = keras.Sequential([
    keras.layers.
    ↪BatchNormalization(input_shape=INPUT_SHAPE,
                        momentum=0.80),
    keras.layers.Dense(1024, activation='relu',
                        kernel_initializer=keras.
    ↪initializers.glorot_normal(seed=99)),
    keras.layers.Dropout(0.3, seed=4),
    keras.layers.Dense(512, activation='relu',
                        kernel_initializer=keras.
    ↪initializers.glorot_normal(seed=99)),
    keras.layers.Dropout(0.3, seed=4),
    keras.layers.Dense(256, activation='relu',
                        kernel_initializer=keras.
    ↪initializers.glorot_normal(seed=99)),
    keras.layers.Dense(128, activation='relu',
                        kernel_initializer=keras.
    ↪initializers.glorot_normal(seed=99)),
    keras.layers.Dropout(0.3, seed=4),
    keras.layers.Dense(64, activation='relu',
                        kernel_initializer=keras.
    ↪initializers.glorot_normal(seed=99)),
    keras.layers.Dropout(0.25, seed=4),
    keras.layers.Dense(32, activation='relu',
                        kernel_initializer=keras.
    ↪initializers.glorot_normal(seed=99)),
    keras.layers.Dropout(0.1, seed=4),

```

```

        keras.layers.Dense(16, activation='relu',
                             kernel_initializer=keras.
→initializers.glorot_normal(seed=99)),
        keras.layers.Dense(1, activation='sigmoid')
])

model1.compile(
    #optimizer=keras.optimizers.RMSprop(),
    optimizer=keras.optimizers.Adam(lr=0.001),
    loss=keras.losses.BinaryCrossentropy(),
    metrics=METRICS)

early_stopping = tf.keras.callbacks.EarlyStopping(
    monitor='val_auc',
    verbose=1,
    patience=5,
    mode='max',
    restore_best_weights=True)

model1.summary()

```

Model: "sequential\_7"

Layer (type)	Output Shape	Param #
batch_normalization_7 (Batch Normalization)	(None, 3003)	12012
dense_56 (Dense)	(None, 1024)	3076096
dropout_35 (Dropout)	(None, 1024)	0
dense_57 (Dense)	(None, 512)	524800
dropout_36 (Dropout)	(None, 512)	0
dense_58 (Dense)	(None, 256)	131328
dense_59 (Dense)	(None, 128)	32896
dropout_37 (Dropout)	(None, 128)	0
dense_60 (Dense)	(None, 64)	8256
dropout_38 (Dropout)	(None, 64)	0
dense_61 (Dense)	(None, 32)	2080



dropout_39 (Dropout)	(None, 32)	0
-----		
dense_62 (Dense)	(None, 16)	528
-----		
dense_63 (Dense)	(None, 1)	17
=====		
Total params: 3,788,013		
Trainable params: 3,782,007		
Non-trainable params: 6,006		
-----		

```
[81]: start_NN = time.time()

EPOCHS=100
resampled_history = model1.fit(
    resampled_features, resampled_labels,
    epochs=EPOCHS,
    batch_size=64,
    callbacks = [early_stopping],
    validation_split=0.2
)

end_NN = time.time()
print("training model takes %s seconds" % round((end_NN-start_NN),3))
```

```
Epoch 1/100
49/49 [=====] - 3s 71ms/step - loss: 0.8279 - accuracy:
0.4979 - auc: 0.4973 - precision: 0.4945 - recall: 0.4333 - val_loss: 0.6949 -
val_accuracy: 0.5084 - val_auc: 0.5181 - val_precision: 0.5097 - val_recall:
0.9924
Epoch 2/100
49/49 [=====] - 3s 68ms/step - loss: 0.7238 - accuracy:
0.5089 - auc: 0.5164 - precision: 0.5058 - recall: 0.5376 - val_loss: 0.6904 -
val_accuracy: 0.5508 - val_auc: 0.5670 - val_precision: 0.5373 - val_recall:
0.8715
Epoch 3/100
49/49 [=====] - 3s 68ms/step - loss: 0.7157 - accuracy:
0.4957 - auc: 0.5006 - precision: 0.4937 - recall: 0.5544 - val_loss: 0.6913 -
val_accuracy: 0.5187 - val_auc: 0.5410 - val_precision: 0.5228 - val_recall:
0.6650
Epoch 4/100
49/49 [=====] - 3s 68ms/step - loss: 0.7082 - accuracy:
0.5192 - auc: 0.5149 - precision: 0.5147 - recall: 0.5784 - val_loss: 0.6911 -
val_accuracy: 0.5135 - val_auc: 0.5684 - val_precision: 0.5629 - val_recall:
0.2141
Epoch 5/100
49/49 [=====] - 3s 68ms/step - loss: 0.7004 - accuracy:
0.5069 - auc: 0.5004 - precision: 0.5039 - recall: 0.5389 - val_loss: 0.6919 -
```

val\_accuracy: 0.5354 - val\_auc: 0.5555 - val\_precision: 0.5308 - val\_recall: 0.7809

Epoch 6/100

49/49 [=====] - 3s 70ms/step - loss: 0.6962 - accuracy: 0.5176 - auc: 0.5260 - precision: 0.5145 - recall: 0.5278 - val\_loss: 0.6947 - val\_accuracy: 0.5225 - val\_auc: 0.5110 - val\_precision: 0.5489 - val\_recall: 0.3678

Epoch 7/100

49/49 [=====] - 3s 69ms/step - loss: 0.6937 - accuracy: 0.5488 - auc: 0.5683 - precision: 0.5467 - recall: 0.5421 - val\_loss: 0.6682 - val\_accuracy: 0.6165 - val\_auc: 0.6574 - val\_precision: 0.6410 - val\_recall: 0.5668

Epoch 8/100

49/49 [=====] - 3s 69ms/step - loss: 0.6789 - accuracy: 0.5820 - auc: 0.6086 - precision: 0.5712 - recall: 0.6392 - val\_loss: 0.6428 - val\_accuracy: 0.6281 - val\_auc: 0.6757 - val\_precision: 0.6280 - val\_recall: 0.6675

Epoch 9/100

49/49 [=====] - 3s 68ms/step - loss: 0.6507 - accuracy: 0.6006 - auc: 0.6450 - precision: 0.5836 - recall: 0.6872 - val\_loss: 0.6228 - val\_accuracy: 0.6757 - val\_auc: 0.7364 - val\_precision: 0.6436 - val\_recall: 0.8186

Epoch 10/100

49/49 [=====] - 3s 69ms/step - loss: 0.6237 - accuracy: 0.6319 - auc: 0.6931 - precision: 0.6206 - recall: 0.6684 - val\_loss: 0.5970 - val\_accuracy: 0.6834 - val\_auc: 0.7505 - val\_precision: 0.6332 - val\_recall: 0.9043

Epoch 11/100

49/49 [=====] - 3s 69ms/step - loss: 0.6018 - accuracy: 0.6631 - auc: 0.7374 - precision: 0.6479 - recall: 0.7066 - val\_loss: 0.5691 - val\_accuracy: 0.6602 - val\_auc: 0.7758 - val\_precision: 0.6157 - val\_recall: 0.8917

Epoch 12/100

49/49 [=====] - 3s 68ms/step - loss: 0.5616 - accuracy: 0.6953 - auc: 0.7781 - precision: 0.6728 - recall: 0.7539 - val\_loss: 0.5432 - val\_accuracy: 0.7079 - val\_auc: 0.8079 - val\_precision: 0.7607 - val\_recall: 0.6247

Epoch 13/100

49/49 [=====] - 3s 69ms/step - loss: 0.5282 - accuracy: 0.7192 - auc: 0.8069 - precision: 0.7005 - recall: 0.7604 - val\_loss: 0.5991 - val\_accuracy: 0.7143 - val\_auc: 0.7621 - val\_precision: 0.6606 - val\_recall: 0.9068

Epoch 14/100

49/49 [=====] - 3s 70ms/step - loss: 0.4969 - accuracy: 0.7456 - auc: 0.8329 - precision: 0.7116 - recall: 0.8212 - val\_loss: 0.4748 - val\_accuracy: 0.7838 - val\_auc: 0.8576 - val\_precision: 0.7657 - val\_recall: 0.8312

Epoch 15/100

49/49 [=====] - 3s 68ms/step - loss: 0.4696 - accuracy: 0.7778 - auc: 0.8582 - precision: 0.7485 - recall: 0.8329 - val\_loss: 0.4545 - val\_accuracy: 0.8018 - val\_auc: 0.8613 - val\_precision: 0.7845 - val\_recall: 0.8438

Epoch 16/100

49/49 [=====] - 3s 69ms/step - loss: 0.4723 - accuracy: 0.7646 - auc: 0.8551 - precision: 0.7634 - recall: 0.7630 - val\_loss: 0.4309 - val\_accuracy: 0.8069 - val\_auc: 0.8805 - val\_precision: 0.7826 - val\_recall: 0.8615

Epoch 17/100

49/49 [=====] - 3s 68ms/step - loss: 0.4494 - accuracy: 0.7942 - auc: 0.8732 - precision: 0.7698 - recall: 0.8361 - val\_loss: 0.4209 - val\_accuracy: 0.8198 - val\_auc: 0.8827 - val\_precision: 0.7800 - val\_recall: 0.9018

Epoch 18/100

49/49 [=====] - 3s 68ms/step - loss: 0.4163 - accuracy: 0.8068 - auc: 0.8865 - precision: 0.7763 - recall: 0.8588 - val\_loss: 0.3918 - val\_accuracy: 0.8366 - val\_auc: 0.8980 - val\_precision: 0.8111 - val\_recall: 0.8866

Epoch 19/100

49/49 [=====] - 3s 67ms/step - loss: 0.4010 - accuracy: 0.8161 - auc: 0.8946 - precision: 0.7901 - recall: 0.8582 - val\_loss: 0.4333 - val\_accuracy: 0.7773 - val\_auc: 0.8922 - val\_precision: 0.8237 - val\_recall: 0.7179

Epoch 20/100

49/49 [=====] - 3s 68ms/step - loss: 0.4033 - accuracy: 0.8148 - auc: 0.8951 - precision: 0.7739 - recall: 0.8867 - val\_loss: 0.4041 - val\_accuracy: 0.8069 - val\_auc: 0.8934 - val\_precision: 0.7947 - val\_recall: 0.8388

Epoch 21/100

49/49 [=====] - 3s 67ms/step - loss: 0.3890 - accuracy: 0.8209 - auc: 0.9025 - precision: 0.7930 - recall: 0.8659 - val\_loss: 0.4003 - val\_accuracy: 0.8237 - val\_auc: 0.9038 - val\_precision: 0.7664 - val\_recall: 0.9421

Epoch 22/100

49/49 [=====] - 3s 68ms/step - loss: 0.3613 - accuracy: 0.8309 - auc: 0.9184 - precision: 0.8097 - recall: 0.8627 - val\_loss: 0.3636 - val\_accuracy: 0.8430 - val\_auc: 0.9131 - val\_precision: 0.8146 - val\_recall: 0.8967

Epoch 23/100

49/49 [=====] - 3s 67ms/step - loss: 0.3459 - accuracy: 0.8409 - auc: 0.9214 - precision: 0.8121 - recall: 0.8847 - val\_loss: 0.3595 - val\_accuracy: 0.8507 - val\_auc: 0.9247 - val\_precision: 0.8022 - val\_recall: 0.9395

Epoch 24/100

49/49 [=====] - 3s 69ms/step - loss: 0.3443 - accuracy: 0.8464 - auc: 0.9260 - precision: 0.8166 - recall: 0.8912 - val\_loss: 0.4439 - val\_accuracy: 0.7928 - val\_auc: 0.8605 - val\_precision: 0.7389 - val\_recall:

0.9194

Epoch 25/100

49/49 [=====] - 3s 70ms/step - loss: 0.3540 - accuracy: 0.8303 - auc: 0.9187 - precision: 0.8065 - recall: 0.8666 - val\_loss: 0.3627 - val\_accuracy: 0.8237 - val\_auc: 0.9156 - val\_precision: 0.8009 - val\_recall: 0.8715

Epoch 26/100

49/49 [=====] - 3s 69ms/step - loss: 0.3199 - accuracy: 0.8548 - auc: 0.9334 - precision: 0.8314 - recall: 0.8880 - val\_loss: 0.3590 - val\_accuracy: 0.8288 - val\_auc: 0.9162 - val\_precision: 0.7661 - val\_recall: 0.9572

Epoch 27/100

49/49 [=====] - 3s 69ms/step - loss: 0.3082 - accuracy: 0.8612 - auc: 0.9389 - precision: 0.8322 - recall: 0.9028 - val\_loss: 0.3082 - val\_accuracy: 0.8674 - val\_auc: 0.9416 - val\_precision: 0.8789 - val\_recall: 0.8589

Epoch 28/100

49/49 [=====] - 3s 66ms/step - loss: 0.3053 - accuracy: 0.8567 - auc: 0.9407 - precision: 0.8332 - recall: 0.8899 - val\_loss: 0.3589 - val\_accuracy: 0.8546 - val\_auc: 0.9123 - val\_precision: 0.8034 - val\_recall: 0.9471

Epoch 29/100

49/49 [=====] - 3s 67ms/step - loss: 0.3077 - accuracy: 0.8576 - auc: 0.9398 - precision: 0.8352 - recall: 0.8892 - val\_loss: 0.3262 - val\_accuracy: 0.8739 - val\_auc: 0.9363 - val\_precision: 0.8453 - val\_recall: 0.9219

Epoch 30/100

49/49 [=====] - 3s 66ms/step - loss: 0.2815 - accuracy: 0.8776 - auc: 0.9508 - precision: 0.8665 - recall: 0.8912 - val\_loss: 0.3431 - val\_accuracy: 0.8468 - val\_auc: 0.9267 - val\_precision: 0.8697 - val\_recall: 0.8237

Epoch 31/100

49/49 [=====] - 3s 66ms/step - loss: 0.2816 - accuracy: 0.8741 - auc: 0.9504 - precision: 0.8556 - recall: 0.8983 - val\_loss: 0.3289 - val\_accuracy: 0.8430 - val\_auc: 0.9389 - val\_precision: 0.8917 - val\_recall: 0.7884

Epoch 32/100

49/49 [=====] - ETA: 0s - loss: 0.2933 - accuracy: 0.8680 - auc: 0.9464 - precision: 0.8607 - recall: 0.8763

Restoring model weights from the end of the best epoch.

49/49 [=====] - 3s 67ms/step - loss: 0.2933 - accuracy: 0.8680 - auc: 0.9464 - precision: 0.8607 - recall: 0.8763 - val\_loss: 0.3452 - val\_accuracy: 0.8237 - val\_auc: 0.9185 - val\_precision: 0.8186 - val\_recall: 0.8413

Epoch 00032: early stopping

training model takes 110.419 seconds

```
[82]: # calculate weighted accuracy on test data
pred_prob = model1.predict(feature_test).reshape(len(y_test))
pred_test = np.zeros(len(y_test))
pred_test[pred_prob>0.5] = 1

weight_test = np.zeros(len(y_test))
for v in np.unique(y_test):
    weight_test[y_test==v] = 0.5*len(y_test)/np.sum(y_test==v)
weighted_acc = np.sum(weight_test * (pred_test==y_test))/np.sum(weight_test)
```

```
[94]: test_start = time.time()
eval = model1.evaluate(feature_test, y_test)
test_end = time.time()
print('Test accuracy: {:.2f}'.format(eval[1]))
print('Test weighted accuracy: {:.3f}'.format(weighted_acc))
print('Test auc: {:.3f}'.format(eval[2]))
print('Testing takes {:.2f} seconds'.format(test_end-test_start))
```

```
19/19 [=====] - 0s 11ms/step - loss: 0.4495 - accuracy:
0.8050 - auc: 0.8306 - precision: 0.5764 - recall: 0.5971
Test accuracy: 0.81
Test weighted accuracy: 0.732
Test auc: 0.831
Testing takes 0.30 seconds
```

#### 4.4 4. Visualization

```
[48]: import matplotlib as mpl
from matplotlib import pyplot as plt
mpl.rcParams['figure.figsize'] = (12, 10)
colors = plt.rcParams['axes.prop_cycle'].by_key()['color']

def plot_metrics(history):
    metrics = ['auc', 'accuracy', 'precision', 'recall']
    for n, metric in enumerate(metrics):
        name = metric.replace("_", " ").capitalize()
        plt.subplot(2,2,n+1)
        plt.plot(history.epoch, history.history[metric], color=colors[0],
        label='Train')
        plt.plot(history.epoch, history.history['val_'+metric],
        color=colors[0], linestyle="--", label='Val')
        plt.xlabel('Epoch')
        plt.ylabel(name)
        if metric == 'loss':
            plt.ylim([0, plt.ylim()[1]])
        elif metric == 'auc':
            plt.ylim([0.8,1])
```

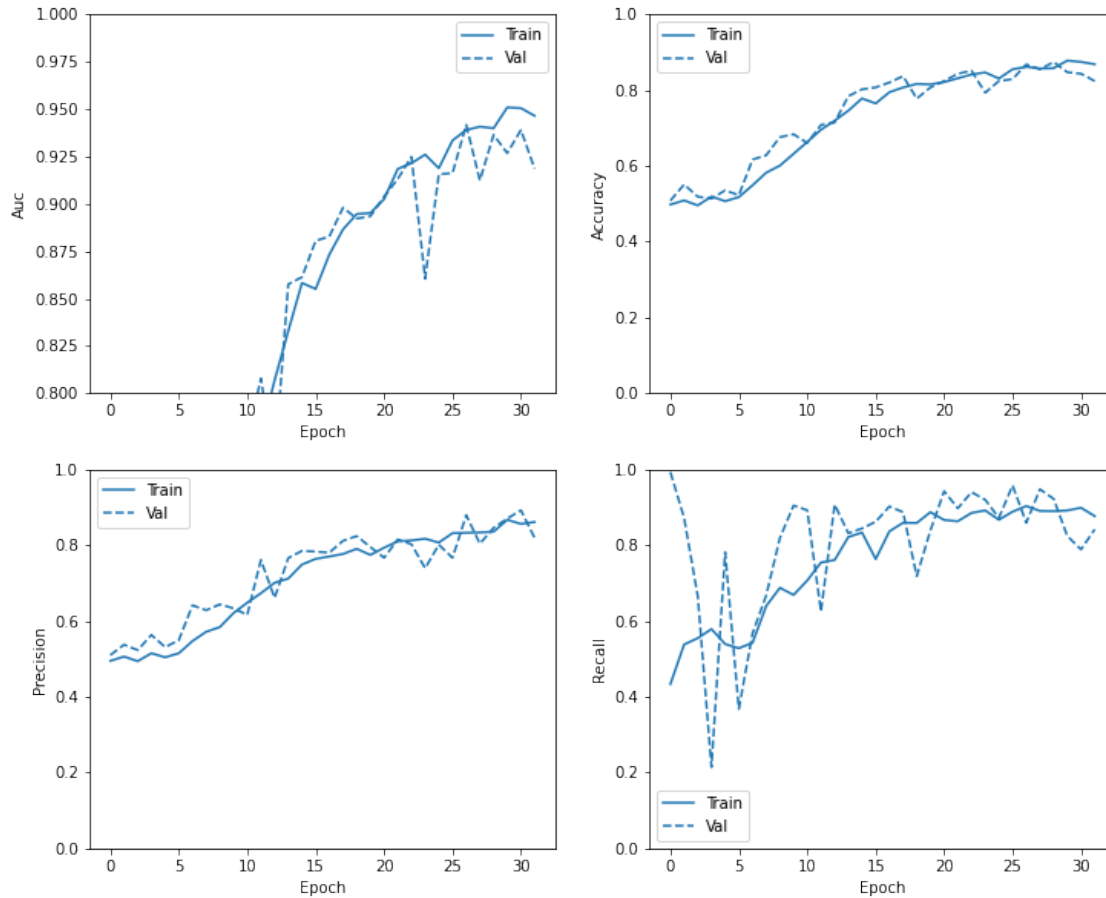
```

else:
    plt.ylim([0,1])

plt.legend()

```

```
[84]: plot_metrics(resampled_history)
```



## 4.5 5. Export the model

```

[96]: export_path_keras = "NNmodel.h5"
model1.save(export_path_keras)

# load the exported model

# NN_model = tf.keras.models.load_model(
#     export_path_keras,
#     custom_objects={'KerasLayer': hub.KerasLayer})

```

## 5 Part IV Comparison

The densely connected neural network is chosen from all the models we experimented for its running time and performance on accuracy and auc. The results and the links for other models we tried are shown below.

Model	Accuracy	Weighted Accuracy	AUC	Train Time	Test time	Link
Baseline Model (GBT)	0.78	0.70	0.79	7.13 s	0.02 s	<a href="#">GBT</a>
XGBoost	0.81	0.72	0.83	37.21 s	0.06 s	<a href="#">XGBoost</a>
Random Forest	0.80	0.58	0.81	8.38 s	0.23 s	<a href="#">Random Forest</a>
SVM	0.66	0.71	0.79	51.37 s	7.87 s	<a href="#">SVM</a>
Neural Networks	0.81	0.73	0.83	110.419 s	0.3 s	<a href="#">NN</a>
CNN	0.52	0.52	0.51	278 s	25 s	<a href="#">CNN</a>
KNN	0.76		0.51	73.33 s	15.2 s	<a href="#">KNN</a>
LDA	0.70		0.68	20.34 s		<a href="#">LDA</a>
LDA with PCA	0.72		0.8	0.02s		<a href="#">LDA with PCA</a>