**Sentiment Analysis with IMDB dataset**

This notebook provides a simple straight-forward way to achieve 90% accuracy on IMDB dataset.

**Load Data**

In [1]:

**import** **utils**

*# Function for loading imdb dataset*

**def** load\_imdb():

train, test = utils.get\_imdb\_dataset()

TEXT\_COL, LABEL\_COL = 'text', 'sentiment'

**return** (

train[TEXT\_COL], train[LABEL\_COL],

test[TEXT\_COL], test[LABEL\_COL])

In [2]:

train\_text, train\_label, test\_text, test\_label = load\_imdb()

data already available, skip downloading.

imdb loaded successfully.

In [3]:

*# Check Shape, should not throw exceptions*

**for** data **in** train\_text, train\_label, test\_text, test\_label:

**assert** data.shape == (25000,)

**Prepare Data**

**Build Vectorizer**

In [4]:

**from** **sklearn.feature\_extraction.text** **import** TfidfVectorizer

tfidf\_vectorizer = TfidfVectorizer(

min\_df=2, *# ignore word that only appears in 1 document*

ngram\_range=(1, 2), *# consider both uni-gram and bi-gram*

)

In [5]:

*# Learn (fit) and transform text into vector*

train\_x = tfidf\_vectorizer.fit\_transform(train\_text)

*# Convert label to 0 and 1 (optional)*

train\_y = train\_label.apply(**lambda** x: 1 **if** x == 'pos' **else** 0)

In [6]:

*# Check the shape*

print('Training x shape:', train\_x.shape)

print('Training y shape:', train\_y.shape)

Training x shape: (25000, 438350)

Training y shape: (25000,)

In [7]:

*# Expect 12500 for 1 and 0, instead of pos and neg*

train\_y.value\_counts()

Out[7]:

1 12500

0 12500

Name: sentiment, dtype: int64

In [8]:

*# Apply the same transformer to validation set as well*

test\_x = tfidf\_vectorizer.transform(test\_text)

test\_y = test\_label.apply(**lambda** x: 1 **if** x == 'pos' **else** 0)

In [9]:

*# Sanity check*

**assert** test\_x.shape == train\_x.shape

**assert** test\_y.shape == train\_y.shape

**Dimensionality Reduction**

In this notebook, SelectKBest from sklearn is used to reduce dimensionality and using f\_classif to help up pick up k best features (word).

In [10]:

**from** **sklearn.feature\_selection** **import** SelectKBest

In [11]:

DIM = 20000 *# Dimensions to keep, a hyper parameter*

*# Create a feature selector*

*# By default, f\_classif algorithm is used*

*# Other available options include mutual\_info\_classif, chi2, f\_regression etc.*

selector = SelectKBest(k=20000)

In [12]:

*# The feature selector also requires information from labels*

*# Fit on training data*

selector.fit(train\_x, train\_y)

Out[12]:

SelectKBest(k=20000, score\_func=<function f\_classif at 0x00000229D46E3798>)

In [13]:

*# Apply to both training data and testing data*

train\_x = selector.transform(train\_x)

test\_x = selector.transform(test\_x)

In [14]:

*# Sanity check*

**assert** train\_x.shape == (25000, 20000)

**assert** test\_x.shape == (25000, 20000)

**Build a MLP Model**

Muti-Layer Perceptron model, aka Feed Forward Network, is the most basic neural network structure, but is used in quite a lot of place as it is very robust. It is true that deep networks are usually more powerful, but they are usually more data hungry. In this coding demostration, for local computation efficieny, I didn't use much data, hence a MLP model may works better.

In [1]:

**from** **tensorflow.keras.models** **import** Model

**from** **tensorflow.python.keras.layers** **import** Input, Dense, Dropout

In [16]:

**def** build\_mlp\_model(input\_dim, layers, output\_dim, dropout\_rate=0.2):

*# Input layer*

X = Input(shape=(input\_dim,))

*# Hidden layer(s)*

H = X

**for** layer **in** layers:

H = Dense(layer, activation='relu')(H)

H = Dropout(rate=dropout\_rate)(H)

*# Output layer*

activation\_func = 'softmax' **if** output\_dim > 1 **else** 'sigmoid'

Y = Dense(output\_dim, activation=activation\_func)(H)

**return** Model(inputs=X, outputs=Y)

In [17]:

hyper\_params = {

'learning\_rate': 1e-3, *# default for Adam*

'epochs': 1000,

'batch\_size': 64,

'layers': [64, 32],

'dim': DIM,

'dropout\_rate': 0.5,

}

In [18]:

mlp\_model = build\_mlp\_model(

input\_dim=hyper\_params['dim'],

layers=hyper\_params['layers'],

output\_dim=1,

dropout\_rate=hyper\_params['dropout\_rate'],

)

mlp\_model.summary()

Model: "model"

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Layer (type) Output Shape Param #

=================================================================

input\_1 (InputLayer) [(None, 20000)] 0

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

dense (Dense) (None, 64) 1280064

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

dropout (Dropout) (None, 64) 0

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

dense\_1 (Dense) (None, 32) 2080

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

dropout\_1 (Dropout) (None, 32) 0

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

dense\_2 (Dense) (None, 1) 33

=================================================================

Total params: 1,282,177

Trainable params: 1,282,177

Non-trainable params: 0

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**Compile the Model**

In [19]:

**from** **tensorflow.keras.optimizers** **import** Adam

In [20]:

mlp\_model.compile(

optimizer=Adam(lr=hyper\_params['learning\_rate']),

loss='binary\_crossentropy',

metrics=['acc'],

)

**Callbacks**

Two common callbacks were used here: EarlyStopping and ModelCheckpoint. The first is used to prevent overfitting and the second is used to keep track of the best models we got so far.

In [21]:

**from** **tensorflow.keras.callbacks** **import** EarlyStopping

**from** **tensorflow.keras.callbacks** **import** ModelCheckpoint

In [22]:

early\_stoppping\_hook = EarlyStopping(

monitor='val\_loss', *# what metrics to track*

patience=2, *# maximum number of epochs allowed without imporvement on monitored metrics*

)

CPK\_PATH = 'model\_cpk.hdf5' *# path to store checkpoint*

model\_cpk\_hook = ModelCheckpoint(

CPK\_PATH,

monitor='val\_loss',

save\_best\_only=**True**, *# Only keep the best model*

)

**Train the Model, Hope for the Best**

In [23]:

his = mlp\_model.fit(

train\_x,

train\_y,

epochs=10,

validation\_data=[test\_x, test\_y],

batch\_size=hyper\_params['batch\_size'],

callbacks=[early\_stoppping\_hook, model\_cpk\_hook],

)

print('Training finished')

Train on 25000 samples, validate on 25000 samples

Epoch 1/10

25000/25000 [==============================] - 9s 356us/sample - loss: 0.3634 - acc: 0.8491 - val\_loss: 0.2378 - val\_acc: 0.9017

Epoch 2/10

25000/25000 [==============================] - 9s 343us/sample - loss: 0.1422 - acc: 0.9508 - val\_loss: 0.2493 - val\_acc: 0.8998

Epoch 3/10

25000/25000 [==============================] - 9s 344us/sample - loss: 0.0838 - acc: 0.9740 - val\_loss: 0.2944 - val\_acc: 0.8957

Training finished

**Evaluation**

Load the best model and do evaluation:

In [24]:

*# Load the model checkpoint*

mlp\_model.load\_weights(CPK\_PATH)

*# Accuracy on validation*

mlp\_model.evaluate(test\_x, test\_y)

25000/25000 [==============================] - 4s 155us/sample - loss: 0.2378 - acc: 0.9017

Out[24]:

[0.23776556309223176, 0.90168]