Machine Learning Project Serie 1:

IMDB Movie Review Sentiment Classification

Episode 3: Embedding Word Matrix

This episode focuses on fitting and testing the data with embedding word matrix usually used in NLP and a densely connected network.

I. Importing Libraries

In [1]:
```
import numpy as np
import os
import pathlib
import tensorflow as tf
from tensorflow.keras import regularizers
from keras.layers import Bidirectional, Concatenate, Permute, Dot, Input, LSTM, Multiply, Embedding, Reshape, D
from keras.layers import RepeatVector, Dense, Activation, Lambda, Softmax, ConvID
from keras.optimizers import Adam, SGD
from keras.utils import to_categorical
from keras.models import load_model, Model
import keras.backend as K
import keras
```

II. Extracting Data

In [2]:
```
(x_train, y_train), (x_test, y_test) = tf.keras.datasets.imdb.load_data()
```

```
<__array_function__ internals>:5: VisibleDeprecationWarning: Creating an ndarray from ragged nested sequences
(which is a list-or-tuple of lists-or-tuples-or ndarrays with different lengths or shapes) is deprecated. If yo
meant to do this, you must specify 'dtype=object' when creating the ndarray
/Users/tieubinh03/opt/anaconda3/lib/python3.8/site-packages/tensorflow/python/keras/datasets/imdb.py:159: Visib
leDeprecationWarning: Creating an ndarray from ragged nested sequences (which is a list-or-tuple of lists-or-tu
ples-or ndarrays with different lengths or shapes) is deprecated. If you meant to do this, you must specify 'dt
ype=object' when creating the ndarray
  x_train, y_train = np.array(xs[:idx]), np.array(labels[:idx])
/Users/tieubinh03/opt/anaconda3/lib/python3.8/site-packages/tensorflow/python/keras/datasets/imdb.py:160: Visib
leDeprecationWarning: Creating an ndarray from ragged nested sequences (which is a list-or-tuple of lists-or-tu
ples-or ndarrays with different lengths or shapes) is deprecated. If you meant to do this, you must specify 'dt
ype=object' when creating the ndarray
  x_test, y_test = np.array(xs[idx:]), np.array(labels[idx:])
```

In [3]:
```
word_dict = tf.keras.datasets.imdb.get_word_index(path="imdb_word_index.json")
```

In [4]:
```
vocab_len = len(word_dict)
print("Total words count:", vocab_len)
```

```
Total words count: 88584
```

III. Data Preprocessing

In [51]:
```
chosen_cmt_len = 2000
max_index = 25000

def padding(initial_x):
    output = np.zeros((chosen_cmt_len))
    for i in range(chosen_cmt_len):
        if i < len(initial_x) and initial_x[i] < max_index:
            output[i] = initial_x[i]
        else:
            output[i] = 0
    return output
```

In [52]:
```
x_train_padded = np.zeros((len(x_train), chosen_cmt_len))
for i in range(len(x_train)):
    x_train_padded[i] = padding(x_train[i])
```

In [53]:
```
x_test_padded = np.zeros((len(x_test), chosen_cmt_len))
for i in range(len(x_test)):
    x_test_padded[i] = padding(x_test[i])
```

IV. Machine Learning Model:

In [54]:
```
e_s = 20
```

In [55]:
```
# Creating model:
def model():

    # Retrieving inputs
    X_input = Input(shape=(chosen_cmt_len,))

    # Embedding meanings
    embedding = Embedding(max_index, e_s)(X_input)

    drop = Dropout(0.9)(embedding)

    flatten = Flatten()(drop)

    output = Dense(1, activation='sigmoid',
                   kernel_regularizer=regularizers.l1_l2(l1=1e-5, l2=1e-4),
                   bias_regularizer=regularizers.l2(1e-4),
                   activity_regularizer=regularizers.l2(1e-5)
                   )(flatten)

    model = Model(inputs = X_input, outputs = output)

    return model
```

In [56]:
```
model = model()
model.summary()
```

```
Model: "model_7"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
input_8 (InputLayer)         [(None, 2000)]            0
_________________________________________________________________
embedding_7 (Embedding)      (None, 2000, 20)          500000
_________________________________________________________________
dropout_7 (Dropout)          (None, 2000, 20)          0
_________________________________________________________________
flatten_7 (Flatten)          (None, 40000)             0
_________________________________________________________________
dense_7 (Dense)              (None, 1)                 40001
=================================================================
Total params: 540,001
Trainable params: 540,001
Non-trainable params: 0
_________________________________________________________________
```

In [57]:
```
# Optimizer for the model
learning_rate = 5e-3
opt = Adam(lr=learning_rate, decay=1e-5)
model.compile(optimizer=opt,
              loss='binary_crossentropy',
              metrics=[tf.keras.metrics.BinaryAccuracy(name="binary_accuracy", threshold=0.5)])
```

In [58]:
```
# Storing histories
histories = []
testings = []

# Track testing accuracy
prev_acc = 0
curr_acc = 0.01

# Max testing accuracy
max_acc = 0

# Fitting and evaluating the model after epochs
epoch = 1

# Keep training as long as testing accuracy on testing set is still increasing
while epoch < 21:
    # Fitting
    print("Epoch:", epoch)
    print("Fitting data:")
    history = model.fit(x = x_train_padded, y = np.array(y_train).reshape(25000, 1), epochs=1, batch_size=1000)

    # Evaluating
    print("Testing data:")
    testing = model.evaluate(x_test_padded, np.array(y_test).reshape(25000, 1))

    # Assigning max accuracy
    if testing[1] > max_acc:
        max_acc = testing[1]

    # Assigning test accuracy
    prev_acc = curr_acc
    curr_acc = testing[1]

    # Adjust learning rate
    if prev_acc > curr_acc:
        learning_rate /= 10
        opt = Adam(lr=learning_rate, decay=1e-5)
        model.compile(optimizer=opt,
                      loss='binary_crossentropy',
                      metrics=[tf.keras.metrics.BinaryAccuracy(name="binary_accuracy", threshold=0.5)])

    # Storing
    histories.append(history)
    testings.append(testing)

    epoch += 1
    print('\n')
```

```
print("Optimal testing accuracy is: {:.2f}%".format(max_acc * 100))
```

```
Epoch: 1
Fitting data:
25/25 [==============================] - 11s 415ms/step - loss: 0.7443 - binary_accuracy: 0.4992
Testing data:
782/782 [==============================] - 2s 2ms/step - loss: 0.6885 - binary_accuracy: 0.5351


Epoch: 2
Fitting data:
25/25 [==============================] - 10s 411ms/step - loss: 0.6308 - binary_accuracy: 0.6651
Testing data:
782/782 [==============================] - 2s 2ms/step - loss: 0.5485 - binary_accuracy: 0.7708


Epoch: 3
Fitting data:
25/25 [==============================] - 10s 399ms/step - loss: 0.4599 - binary_accuracy: 0.8007
Testing data:
782/782 [==============================] - 2s 2ms/step - loss: 0.3946 - binary_accuracy: 0.8506


Epoch: 4
Fitting data:
25/25 [==============================] - 10s 399ms/step - loss: 0.3594 - binary_accuracy: 0.8498
Testing data:
782/782 [==============================] - 2s 2ms/step - loss: 0.3469 - binary_accuracy: 0.8667


Epoch: 5
Fitting data:
25/25 [==============================] - 10s 396ms/step - loss: 0.3081 - binary_accuracy: 0.8791
Testing data:
782/782 [==============================] - 2s 2ms/step - loss: 0.3262 - binary_accuracy: 0.8763


Epoch: 6
Fitting data:
25/25 [==============================] - 10s 397ms/step - loss: 0.2744 - binary_accuracy: 0.8964
Testing data:
782/782 [==============================] - 2s 2ms/step - loss: 0.3131 - binary_accuracy: 0.8804


Epoch: 7
Fitting data:
25/25 [==============================] - 12s 455ms/step - loss: 0.2632 - binary_accuracy: 0.9004
Testing data:
782/782 [==============================] - 1s 2ms/step - loss: 0.3134 - binary_accuracy: 0.8803


Epoch: 8
Fitting data:
25/25 [==============================] - 10s 375ms/step - loss: 0.2395 - binary_accuracy: 0.9108
Testing data:
782/782 [==============================] - 2s 2ms/step - loss: 0.3093 - binary_accuracy: 0.8813


Epoch: 9
Fitting data:
25/25 [==============================] - 10s 397ms/step - loss: 0.2342 - binary_accuracy: 0.9138
Testing data:
782/782 [==============================] - 1s 2ms/step - loss: 0.3081 - binary_accuracy: 0.8816


Epoch: 10
Fitting data:
25/25 [==============================] - 10s 403ms/step - loss: 0.2359 - binary_accuracy: 0.9142
Testing data:
782/782 [==============================] - 2s 2ms/step - loss: 0.3074 - binary_accuracy: 0.8814


Epoch: 11
Fitting data:
25/25 [==============================] - 10s 384ms/step - loss: 0.2246 - binary_accuracy: 0.9165
Testing data:
782/782 [==============================] - 2s 1ms/step - loss: 0.3066 - binary_accuracy: 0.8816


Epoch: 12
Fitting data:
25/25 [==============================] - 9s 372ms/step - loss: 0.2343 - binary_accuracy: 0.9136
Testing data:
782/782 [==============================] - 1s 2ms/step - loss: 0.3075 - binary_accuracy: 0.8819


Epoch: 13
Fitting data:
25/25 [==============================] - 10s 380ms/step - loss: 0.2333 - binary_accuracy: 0.9138
Testing data:
782/782 [==============================] - 1s 2ms/step - loss: 0.3071 - binary_accuracy: 0.8817


Epoch: 14
Fitting data:
25/25 [==============================] - 10s 370ms/step - loss: 0.2318 - binary_accuracy: 0.9143
Testing data:
782/782 [==============================] - 2s 2ms/step - loss: 0.3072 - binary_accuracy: 0.8815


Epoch: 15
Fitting data:
25/25 [==============================] - 10s 404ms/step - loss: 0.2298 - binary_accuracy: 0.9154
Testing data:
782/782 [==============================] - 2s 2ms/step - loss: 0.3072 - binary_accuracy: 0.8816


Epoch: 16
Fitting data:
25/25 [==============================] - 10s 397ms/step - loss: 0.2327 - binary_accuracy: 0.9142
Testing data:
782/782 [==============================] - 2s 2ms/step - loss: 0.3071 - binary_accuracy: 0.8816


Epoch: 17
Fitting data:
25/25 [==============================] - 9s 379ms/step - loss: 0.2278 - binary_accuracy: 0.9181
Testing data:
782/782 [==============================] - 2s 2ms/step - loss: 0.3071 - binary_accuracy: 0.8818


Epoch: 18
Fitting data:
25/25 [==============================] - 10s 397ms/step - loss: 0.2300 - binary_accuracy: 0.9185
Testing data:
782/782 [==============================] - 2s 2ms/step - loss: 0.3071 - binary_accuracy: 0.8817


Epoch: 19
Fitting data:
25/25 [==============================] - 10s 381ms/step - loss: 0.2399 - binary_accuracy: 0.9107
Testing data:
782/782 [==============================] - 2s 2ms/step - loss: 0.3071 - binary_accuracy: 0.8817


Epoch: 20
Fitting data:
25/25 [==============================] - 10s 403ms/step - loss: 0.2292 - binary_accuracy: 0.9139
Testing data:
782/782 [==============================] - 2s 2ms/step - loss: 0.3071 - binary_accuracy: 0.8817


Optimal testing accuracy is: 88.19%
```

In [59]:
```
model.evaluate(x_train_padded, np.array(y_train).reshape(25000, 1))
```

```
782/782 [==============================] - 2s 2ms/step - loss: 0.1575 - binary_accuracy: 0.9411
```

Out[59]:
```
[0.15848954021930695, 0.9401200222969055]
```

V. Summary:

Embedding matrix performed quite well given the number of parameters trained was relatively small compared to the model used in previous episode.

-	Loss	Accuracy	Sample size
Training	0.16	96.1%	25,000
Testing	0.31	88.2%	25,000

VIII. Thank you:

Thank you for viewing my project. See you in the next episode.