IST 691: Deep Learning in Practice

Homework 3

Name: Benjamin Heindl

SUID:

Save this notebook into your Google Drive. The notebook has appropriate comments at the top of code cells to indicate whether you need to modify them or not. Answer your questions directly in the notebook. Remember to use the GPU as your runtime. Once finished, run ensure all code blocks are run, download the notebook and submit through Blackboard.

Setup

In [1]:
```
import tensorflow as tf
import numpy as np
from tensorflow.keras import layers
from tensorflow.keras.layers import TextVectorization
import string
import re
import pandas as pd
from sklearn.model_selection import train_test_split
import json
import tensorflow as tf
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
from tensorflow import keras
from tensorflow.keras.preprocessing.text import Tokenizer
from tensorflow.keras.preprocessing.sequence import pad_sequences

# to build nearest neighbor model
from sklearn.neighbors import NearestNeighbors
```

In this homework, we will perform **sarcasm detection** with Onion vs HuffPost headlines, using LSTM. We will first load the data and generate the training and testing input and labels.

In [2]:
```
! wget -nc -q https://github.com/mrech/NLP_TensorFlow/blob/master/0_Sentiment_in_Text/Sarcasm_Headlines_Dataset_v2.json?raw=true
```

```
zsh:1: no matches found: https://github.com/mrech/NLP_TensorFlow/blob/master/0_Sentiment_in_Text/Sarcasm_Headlines_Dataset_v2.json?raw=true
```

In [4]:
```
import pandas as pd

url = "https://raw.githubusercontent.com/mrech/NLP_TensorFlow/master/0_Sentiment_in_Text/Sarcasm_Headlines_Dataset_v2.json"
df = pd.read_json(url, lines=True)
```

In [5]:
```
# get information about the data frame
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 28619 entries, 0 to 28618
Data columns (total 3 columns):
 #   Column        Non-Null Count  Dtype 
---  ------        --------------  ----- 
 0   is_sarcastic  28619 non-null  int64 
 1   headline      28619 non-null  object
 2   article_link  28619 non-null  object
dtypes: int64(1), object(2)
memory usage: 670.9+ KB
```

In [6]:
```
# take a peek at the key data
df[['headline', 'is_sarcastic']].head(5).values
```

Out[6]:
```
array([['thirtysomething scientists unveil doomsday clock of hair loss',
        1],
       ['dem rep. totally nails why congress is falling short on gender, racial equality',
        0],
       ['eat your veggies: 9 deliciously different recipes', 0],
       ['inclement weather prevents liar from getting to work', 1],
       ["mother comes pretty close to using word 'streaming' correctly",
        1]], dtype=object)
```

In [7]:
```
# the training input sequence will be in variable seq_padd_train and the label in train_y
# The testing input sequence will be in variable seq_padd_test and the label in test_y
headlines = df['headline'].values.tolist()
sarcastic = df['is_sarcastic'].values.tolist()
```

In [8]:
```
training_size = 20000
test_size = 6709

train_x = headlines[:training_size]
test_x = headlines[training_size:]
train_y = np.array(sarcastic[:training_size])
test_y = np.array(sarcastic[training_size:])

# sequence of words input
max_len = 16

tokenizer = Tokenizer(oov_token = '<OOV>')
tokenizer.fit_on_texts(train_x)

word_index = tokenizer.word_index
index_word = {v: k for k, v in word_index.items()}
vocab_size = len(word_index)
sequence_train = tokenizer.texts_to_sequences(train_x)
seq_padd_train = pad_sequences(sequence_train,
                               padding = 'post',
                               truncating = 'post',
                               maxlen = max_len)


sequence_test = tokenizer.texts_to_sequences(test_x)
seq_padd_test = pad_sequences(sequence_test, padding = 'post',
                              truncating = 'post',
                              maxlen = max_len)
```

Q1 Calculating the Trainable Parameters of an LSTM

Below is the summary of an LSTM neural network with embeddings and three layers. Explain in detail, after this cell, the "why" of the number of parameters of each of the layers displayed by `model1.summary()`. Cite any sources you used to answer this question.

`model1.summary()`

```
Model: "model"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
input_1 (InputLayer)         [(None, None)]            0
_________________________________________________________________
embedding (Embedding)        (None, None, 100)         2000100
_________________________________________________________________
lstm (LSTM)                  (None, None, 128)         117248
_________________________________________________________________
lstm_1 (LSTM)                (None, None, 96)          86400
_________________________________________________________________
lstm_2 (LSTM)                (None, 64)                41216
_________________________________________________________________
predictions (Dense)          (None, 1)                 65
=================================================================
Total params: 2,245,029
Trainable params: 2,245,029
Non-trainable params: 0
_________________________________________________________________
```

Why do we have the number of parameters after each of the layers?

The number of parameters at each layer represents the model's complexity and learning capacity

Q2: LSTM for Detecting Sarcasm

Modify the code below to create an embedding layer of dimension 50. The vocabulary size is in variable `vocab_size`, and remember to add one in the embedding for the "out of vocabulary" input. Define an LSTM with two layers, one with 64 memory size and the second with 32 memory size. Remember to use the suffix `2` for each of the variables you define (e.g., `x2`)

In [10]:
```
# an integer input for vocab indices
inputs2 = tf.keras.Input(shape = (None,), dtype = 'int32')

# define the layers below Embedding -> LSTM 1 -> LSTM 2
x2 = layers.Embedding(input_dim=vocab_size + 1, output_dim=50)(inputs2)

x2 = layers.LSTM(64, return_sequences=True)(x2)
x2 = layers.LSTM(32)(x2)

# we project onto a single unit output layer, and squash it with a sigmoid
predictions2 = layers.Dense(1, activation = 'sigmoid', name = 'predictions')(x2)

model2 = tf.keras.Model(inputs2, predictions2, name = 'lstm_simple')

# compile the model with binary crossentropy loss and an adam optimizer
model2.compile(loss = 'binary_crossentropy',
               optimizer = 'adam',
               metrics = ['accuracy'])
```

In [11]:
```
epochs = 10
# fit the model using the train and test datasets
model2.fit(seq_padd_train, train_y,
           validation_split = 0.1,
           epochs = epochs,
           verbose = 2,
           batch_size = 64)
```

```
Epoch 1/10
282/282 - 6s - loss: 0.4250 - accuracy: 0.7909 - val_loss: 0.3324 - val_accuracy: 0.8620 - 6s/epoch - 21ms/step
Epoch 2/10
282/282 - 5s - loss: 0.1791 - accuracy: 0.9343 - val_loss: 0.4261 - val_accuracy: 0.8460 - 5s/epoch - 17ms/step
Epoch 3/10
282/282 - 5s - loss: 0.0820 - accuracy: 0.9719 - val_loss: 0.5116 - val_accuracy: 0.8415 - 5s/epoch - 17ms/step
Epoch 4/10
282/282 - 5s - loss: 0.0427 - accuracy: 0.9869 - val_loss: 0.5516 - val_accuracy: 0.8385 - 5s/epoch - 17ms/step
Epoch 5/10
282/282 - 5s - loss: 0.0247 - accuracy: 0.9924 - val_loss: 0.7043 - val_accuracy: 0.8350 - 5s/epoch - 19ms/step
Epoch 6/10
282/282 - 6s - loss: 0.0151 - accuracy: 0.9954 - val_loss: 0.8018 - val_accuracy: 0.8395 - 6s/epoch - 20ms/step
Epoch 7/10
282/282 - 6s - loss: 0.0144 - accuracy: 0.9957 - val_loss: 0.7869 - val_accuracy: 0.8345 - 6s/epoch - 21ms/step
Epoch 8/10
282/282 - 6s - loss: 0.0078 - accuracy: 0.9978 - val_loss: 0.8244 - val_accuracy: 0.8380 - 6s/epoch - 20ms/step
Epoch 9/10
282/282 - 6s - loss: 0.0067 - accuracy: 0.9981 - val_loss: 0.8214 - val_accuracy: 0.8330 - 6s/epoch - 21ms/step
Epoch 10/10
282/282 - 6s - loss: 0.0057 - accuracy: 0.9984 - val_loss: 0.9389 - val_accuracy: 0.8370 - 6s/epoch - 21ms/step
```

Out[11]:
```
<keras.src.callbacks.History at 0x17f8e39a0>
```

In [12]:
```
# estimate the test performance
model2.evaluate(seq_padd_test, test_y)
```

```
270/270 [==============================] - 1s 2ms/step - loss: 0.9472 - accuracy: 0.8342
```

Out[12]:
```
[0.9472020864486694, 0.8342034816741943]
```

Q3: GloVe Word Embeddings

Use the code below to download the GloVe embeddings and create the matrix `embedding_matrix` corresponding to the vocabulary above. Define a layer `embedding_layer_glove` which will be use by the LSTM below. Evaluate the performance and compare to model above.

In [18]:
```
import requests

url = "http://nlp.stanford.edu/data/glove.6B.zip"
response = requests.get(url, stream=True)

with open("glove.6B.zip", "wb") as file:
    for chunk in response.iter_content(chunk_size=1024):
        if chunk:
            file.write(chunk)
```

In [19]:
```
! unzip glove.6B.zip
```

```
Archive:  glove.6B.zip
  inflating: glove.6B.50d.txt
  inflating: glove.6B.100d.txt
  inflating: glove.6B.200d.txt
  inflating: glove.6B.300d.txt
```

In [20]:
```
import os
embeddings_index = {}
f = open('glove.6B.100d.txt')
for line in f:
    values = line.split()
    word = values[0]
    coefs = np.asarray(values[1:], dtype = 'float32')
    embeddings_index[word] = coefs
f.close()

print('Found %s word vectors.' % len(embeddings_index))
```

```
Found 400000 word vectors.
```

In [21]:
```
num_tokens = vocab_size + 2
embedding_dim3 = 100
hits = 0
misses = 0

# prepare embedding matrix
embedding_matrix = np.zeros((num_tokens, embedding_dim3))
for word, i in word_index.items():
    embedding_vector = embeddings_index.get(word)
    if embedding_vector is not None:
        # words not found in embedding index will be all-zeros.
        # this includes the representation for "padding" and "OOV"
        embedding_matrix[i] = embedding_vector
        hits += 1
    else:
        misses += 1
print("Converted %d words (%d misses)" % (hits, misses))
```

```
Converted 21242 words (4656 misses)
```

Create the embedding layer below:

In [22]:
```
# create the embedding layer using the embedding_matrix from above
embedding_layer_glove = layers.Embedding(
    num_tokens,
    embedding_dim3,
    input_length = max_len,
    embeddings_initializer = tf.keras.initializers.Constant(embedding_matrix),
    trainable = False,
)
```

In [24]:
```
# an integer input for vocab indices
inputs3 = tf.keras.Input(shape = (None,), dtype = 'int32')

# next, we add a layer to map those vocab indices into a space of dimensionality
x3 = embedding_layer_glove(inputs3)

x3 = layers.LSTM(32)(x3)

# we project onto a single unit output layer, and squash it with a sigmoid
predictions3 = layers.Dense(1, activation = 'sigmoid', name = 'predictions')(x3)

model3 = tf.keras.Model(inputs3, predictions3)

# compile the model with binary crossentropy loss and an adam optimizer.
model3.compile(loss = 'binary_crossentropy',
               optimizer = 'adam',
               metrics = ['accuracy'])
```

In [25]:
```
# fit the model using the train and test datasets
epochs = 10
model3.fit(seq_padd_train, train_y,
           validation_split = 0.1,
           epochs = epochs,
           verbose = 2,
           batch_size = 64)
```

```
Epoch 1/10
282/282 - 2s - loss: 0.5379 - accuracy: 0.7191 - val_loss: 0.4413 - val_accuracy: 0.8030 - 2s/epoch - 8ms/step
Epoch 2/10
282/282 - 2s - loss: 0.4042 - accuracy: 0.8201 - val_loss: 0.3930 - val_accuracy: 0.8235 - 2s/epoch - 6ms/step
Epoch 3/10
282/282 - 2s - loss: 0.3561 - accuracy: 0.8471 - val_loss: 0.3476 - val_accuracy: 0.8520 - 2s/epoch - 6ms/step
Epoch 4/10
282/282 - 1s - loss: 0.3276 - accuracy: 0.8621 - val_loss: 0.3364 - val_accuracy: 0.8590 - 1s/epoch - 4ms/step
Epoch 5/10
282/282 - 1s - loss: 0.3018 - accuracy: 0.8742 - val_loss: 0.3400 - val_accuracy: 0.8535 - 1s/epoch - 5ms/step
Epoch 6/10
282/282 - 1s - loss: 0.2817 - accuracy: 0.8835 - val_loss: 0.3243 - val_accuracy: 0.8630 - 1s/epoch - 4ms/step
Epoch 7/10
282/282 - 1s - loss: 0.2659 - accuracy: 0.8917 - val_loss: 0.3422 - val_accuracy: 0.8625 - 1s/epoch - 4ms/step
Epoch 8/10
282/282 - 1s - loss: 0.2504 - accuracy: 0.8981 - val_loss: 0.3330 - val_accuracy: 0.8635 - 1s/epoch - 4ms/step
Epoch 9/10
282/282 - 1s - loss: 0.2341 - accuracy: 0.9050 - val_loss: 0.3529 - val_accuracy: 0.8545 - 1s/epoch - 4ms/step
Epoch 10/10
282/282 - 1s - loss: 0.2239 - accuracy: 0.9106 - val_loss: 0.3454 - val_accuracy: 0.8610 - 1s/epoch - 5ms/step
```

Out[25]:
```
<keras.src.callbacks.History at 0x289883ac0>
```

In [26]:
```
model3.evaluate(seq_padd_test, test_y)
```

```
270/270 [==============================] - 0s 1ms/step - loss: 0.3415 - accuracy: 0.8561
```

Out[26]:
```
[0.34147176146507263, 0.8561317920684814]
```

Is it better or worse performance compared to `model2`? Why?

Accuracy: model3 outperforms model2 in terms of both validation and test accuracy, indicating better generalization to unseen data.

Loss: model3 also shows lower loss on the validation and test sets, suggesting better fit and generalization.

Overfitting in model2: The training trajectory of model2 shows signs of overfitting, looking at the high training accuracy and increasing validation loss over epochs. It seems as though while model2 learns the training data very well, it struggles to generalize this learning to new data.

Consistency in model3: model3 demonstrates more consistent performance between training and validation, meaning better generalization

model3 demonstrates better performance in terms of generalization to unseen data, looking at both higher validation and test accuracies and lower loss. It seems as though the pre-trained embeddings provide an advantage, particularly in capturing complex language features.

model2 shows signs of overfitting, indicating a need for better regularization or a more complex model architecture to improve its generalization capabilities

In [ ]:

In [ ]:

In [ ]:

In [ ]:

In [ ]:

In [ ]:

Q4: Word Analogies

Above, we created the matrix `embedding_matrix` for the vocabulary in the sarcasm dataset. Use the code below to find the word analogy to "`germany` is to `berlin` as `uk` is to *blank*"

In [32]:
```
# we will first create the nearest neighbor model
nbrs_glove = NearestNeighbors(n_neighbors = 5, metric = 'cosine').fit(embedding_matrix)
```

In [33]:
```
# let's check if it works
embedding_man = embedding_matrix[word_index['man']]
```

In [34]:
```
# closest words to `man`
dist, idx = nbrs_glove.kneighbors([embedding_man])
[index_word[i] for i in idx[0]]
```

Out[34]:
```
['man', 'woman', 'boy', 'one', 'person']
```

In [35]:
```
# Retrieve embeddings for 'germany', 'berlin', and 'uk'
embedding_germany = embedding_matrix[word_index['germany']] if 'germany' in word_index else None
embedding_berlin = embedding_matrix[word_index['berlin']] if 'berlin' in word_index else None
embedding_uk = embedding_matrix[word_index['uk']] if 'uk' in word_index else None

# Calculate the vector for the analogy and find the closest words
if embedding_germany is not None and embedding_berlin is not None and embedding_uk is not None:
    # berlin - germany + uk
    blank_embedding = embedding_berlin - embedding_germany + embedding_uk

    # Find the closest words to the analogy vector
    dist, idx = nbrs_glove.kneighbors([blank_embedding])
    nearest_words = [index_word[i] for i in idx[0]]
else:
    nearest_words = ["Embedding not found for one or more words"]

nearest_words
```

Out[35]:
```
['uk', 'london', 'theatre', '2013', '2011']
```

In [36]:
```
# find the closest to blank_embedding
# closest words to `man`
#dist, idx = nbrs_glove.kneighbors([blank_embedding])
#[index_word[i] for i in idx[0]]
```

Q5: Biases

As we discussed in class, there might be several biases in word embeddings. Use the list of occupations below and for each of them find whether `man` or `woman` is closest to it. In particular, first list all occupations that are closer to `man` than `woman`, and then all occupations that are closer to `woman` than `man`.

Hint: Use the `cosine` distance between pairs of embeddings from the `SciPy` package. If the ocupation does not exist in the embedding matrix, skip it. Also, remember that the cosine distance is smaller when the embeddings are more similar.

In [37]:
```
from scipy.spatial.distance import cosine
print('cosine([1,1], [1,1]): ', cosine([1,1], [1,1]))
print('cosine([1,1], [0,1]): ', cosine([1,1], [0,1]))
```

```
cosine([1,1], [1,1]):  0
cosine([1,1], [0,1]):  0.29289321881345254
```

In [38]:
```
occupation_list = """technician, accountant, supervisor, engineer, worker, educator, clerk, counselor,
inspector, mechanic, manager, therapist, administrator, salesperson, receptionist, librarian,
advisor, pharmacist, janitor, psychologist, physician, carpenter, nurse, investigator,
bartender, specialist, electrician, officer, pathologist, teacher, lawyer, planner, practitioner,
plumber, instructor, surgeon, veterinarian, paramedic, examiner, chemist, machinist,
appraiser, nutritionist, architect, hairdresser, baker, programmer, paralegal, hygienist,
scientist""".replace('\n', '').replace(' ', '').split(',')
```

In [39]:
```
man_embedding = embedding_matrix[word_index['man']]
woman_embedding = embedding_matrix[word_index['woman']]
```

In [41]:
```
occupations_closer_to_man = []
occupations_closer_to_woman = []

for occupation in occupation_list:
    if occupation in word_index:
        occupation_embedding = embedding_matrix[word_index[occupation]]
        distance_to_man = cosine(occupation_embedding, man_embedding)
        distance_to_woman = cosine(occupation_embedding, woman_embedding)

        if distance_to_man < distance_to_woman:
            occupations_closer_to_man.append(occupation)
        else:
            occupations_closer_to_woman.append(occupation)
```

In [42]:
```
# first print the ocupations that are for a man, as perceived by GloVe
print("Occupations closer to 'man':")
print(occupations_closer_to_man)

# second print the ocupations that are for a woman, as perceived by GloVe
print("\nOccupations closer to 'woman':")
print(occupations_closer_to_woman)
```

```
Occupations closer to 'man':
['engineer', 'inspector', 'mechanic', 'manager', 'advisor', 'carpenter', 'investigator', 'officer', 'lawyer', 'planner', 'plumber', 'instructor', 'architect', 'scientist']

Occupations closer to 'woman':
['technician', 'supervisor', 'worker', 'educator', 'clerk', 'counselor', 'therapist', 'administrator', 'receptionist', 'librarian', 'pharmacist', 'janitor', 'psychologist', 'physician', 'nurse', 'bartender', 'teach
er', 'practitioner', 'surgeon', 'veterinarian', 'paramedic', 'examiner', 'nutritionist', 'hairdresser', 'hygienist']
```

Do you see a pattern in the results? Do you think there are biases?

Looking at the results, there's a pretty clear pattern that shows some traditional thinking about which jobs are for men and which are for women. The jobs listed closer to 'man' are typically the kind you might think of as 'guy roles' historically, like engineers, lawyers, or carpenters. On the other hand, the jobs closer to 'woman' are often those traditionally seen as 'women's work', like nursing, teaching, or being a librarian.

This isn't about what jobs men and women can actually do, of course. It's more about the kind of language that gets used on the internet and in books, which is where the GloVe model seems to have learned its stuff. If people talk more about men being engineers and women being nurses, that's what the AI is going to pick up.

Q6: Sequence to Sequence Embedding

What is the problem with LSTM models, and why do we need **attention** to fix them? Give as an example of what happens with sequence to sequence models for translation.

LSTMs have a problem with long sequences. It seems like they tend to forget earlier information when dealing with a lot of data. This is a big issue in tasks like translation, where every piece of a sentence matters.

Attention mechanisms help by allowing the model to focus on specific parts of the input sequence as needed, rather than relying on a single, fixed representation of the entire sequence. This way, the model can handle longer sentences more effectively.

In translation, for example, with attention, the model doesn't just convert an entire English sentence into a French one in one go. Instead, it looks at different parts of the English sentence while translating each word or phrase, ensuring that the context is maintained throughout the sentence. This leads to more accurate translations, especially for longer sentences.

In [ ]: