Word Embedding and Positional Embedding using Recurrent Neural Network (RNN) and Transformers Architecture

Summary

A Sequence to Text approach has taken to the IMDB example using Recurrent Neural Network (RNN) by training the embedding layer on own and with a pretrained word embedding layer using different sample sizes and altering text lengths from 100 to 150. Test Accuracy for different observations are mentioned in Table 1 below:

Intrepration for Table 1: Embedding layer with masking performed well compared to Pretrained word embedding with all the different training samples (100, 200, 300, 400) by altering text length to cutoff the reviews after 150 or 300 words. Usually, it should be viceversa i.e., Pretrained word embedding should perform better compared to Embedding layer with masking.

The possible reason could be as follows:

1. **Small Training Data:** If training data size (100, 200, 300, 400 samples) is relatively small, the pre-trained embeddings might not have enough data to adapt effectively to the specific task of sentiment analysis on movie reviews.
2. **Domain Specificity:** Pre-trained word embeddings like GloVe or Word2Vec are trained on massive datasets that might not be specific to movie reviews. The embedding layer trained on your own IMDB data might capture the nuances of sentiment and vocabulary specific to movie reviews better.
3. **Masking Impact:** Padding sequences to a fixed length (150 or 300 words) with pre-trained embeddings might introduce noise, especially if many reviews are shorter. Masking removes these padding tokens, allowing the model to focus on the actual content.

Table 1: Comparision between Word Embedding and Positional Embedding using RNN

S.no	Model Name	Test Accuracy 100 Training samples, 150 Text length	Test Accuracy 200 Training samples, 150 Text length	Test Accuracy 400 Training samples, 150 Text length	Test Accuracy 100 Training samples, 300 Text length	Test Accuracy 300 Training samples, 300 Text length
1	Basic sequential	55.4	49.5	57.7	52.0	60.8
2	Embedded layer	50.7	50.7	49.5	60.6	56.8
3	Embedded layer with masking	61.1	59.9	61.0	62.9	61.8
4	Pretrained word embedding	54.4	55.3	50.0	53.3	53.1

In addition to this, the same IMDB dataset has passed through the "Transformer Architecture" for both word embedding and positional embedding. Given the 100 training sample size with 300 text length performed well using RNN, transformers architecture was implememted on the sample sample size.Apparently, the Test Accuracy is higher in Transformers compared to RNN however, the positional encoding is still lower compared to the base Transformer encoder.

Assuming its due to lower training sample size (100), it was increased to 1000 training samples but still the trend between positional encoding and base transformer encoder is same. Details of the observations are mentioned below:

Table 2: Comparision between Word Embedding and Positional Embedding using RNN and Transformers

S.no	Model Name	Test Accuracy 100 Training samples, 300 Text length	Test Accuracy 1000 Training samples, 300 Text length
1	Basic sequential	52.0	74.0
2	Embedded layer	60.6	71.8
3	Embedded layer with masking	62.9	77.6
4	Pretrained word embedding	53.3	66.5
5	Transformer Encoder based	68.2	80.5
6	Positional Embedding	61.9	76.9

Note: This file has the execution of 100 Training Samples with 150 Text Length. Other files with training samples are attached in [this github folder.]

Processing words as a sequence: The sequence model approach

A first practical example

```
Start coding or generate with AI.
```

Downloading the data

```
!curl -O https://ai.stanford.edu/~amaas/data/sentiment/aclImdb_v1.tar.gz
!tar -xf aclImdb_v1.tar.gz
!rm -r aclImdb/train/unsup
```

```
  % Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
                                 Dload  Upload   Total   Spent    Left  Speed
100 80.2M  100 80.2M    0     0  5081k      0  0:00:16  0:00:16 --:--:-- 7485k
```

Preparing the data

```
import os, pathlib, shutil, random
from tensorflow import keras
batch_size = 32
base_dir = pathlib.Path("aclImdb")
val_dir = base_dir / "val"
train_dir = base_dir / "train"
excess_dir = base_dir / "excess"
for category in ("neg", "pos"):
    os.makedirs(val_dir / category)
    os.makedirs(excess_dir / category)
    files = os.listdir(train_dir / category)
    random.Random(1337).shuffle(files)
    num_val_samples = 5000
    val_files = files[-num_val_samples:]
    for fname in val_files:
        shutil.move(train_dir / category / fname,
                    val_dir / category / fname)

    files = os.listdir(train_dir / category)
    random.Random(1338).shuffle(files)
    num_ex_samples = 50
    ex_files = files[-num_ex_samples:]
    for fname in ex_files:
        shutil.move(train_dir / category / fname,
                    excess_dir / category / fname)

train_ds = keras.utils.text_dataset_from_directory(
    "aclImdb/excess", batch_size=batch_size
)
val_ds = keras.utils.text_dataset_from_directory(
    "aclImdb/val", batch_size=batch_size
)
test_ds = keras.utils.text_dataset_from_directory(
    "aclImdb/test", batch_size=batch_size
)
text_only_train_ds = train_ds.map(lambda x, y: x)
```

```
Found 100 files belonging to 2 classes.
Found 10000 files belonging to 2 classes.
Found 25000 files belonging to 2 classes.
```

```
!ls -al /content/aclImdb/excess/neg/ | wc
!ls -al /content/aclImdb/excess/pos/ | wc
!ls -al /content/aclImdb/val/pos/ | wc
!rm -rf /content/aclImdb1/
!rm -rf /content/aclImdb/
```

```
     53     470    2702
     53     470    2722
  10003   90020  545011
```

Preparing integer sequence datasets

```
from tensorflow.keras import layers

max_length = 150
max_tokens = 10000
text_vectorization = layers.TextVectorization(
    max_tokens=max_tokens,
    output_mode="int",
    output_sequence_length=max_length,
)
text_vectorization.adapt(text_only_train_ds)

int_train_ds = train_ds.map(
    lambda x, y: (text_vectorization(x), y),
    num_parallel_calls=4)
int_val_ds = val_ds.map(
    lambda x, y: (text_vectorization(x), y),
    num_parallel_calls=4)
int_test_ds = test_ds.map(
    lambda x, y: (text_vectorization(x), y),
    num_parallel_calls=4)
```

A sequence model built on one-hot encoded vector sequences

```
import tensorflow as tf
inputs = keras.Input(shape=(None,), dtype="int64")
embedded = tf.one_hot(inputs, depth=max_tokens)
x = layers.Bidirectional(layers.LSTM(32))(embedded)
x = layers.Dropout(0.5)(x)
outputs = layers.Dense(1, activation="sigmoid")(x)
model = keras.Model(inputs, outputs)
model.compile(optimizer="adam",
              loss="binary_crossentropy",
              metrics=["accuracy"])
model.summary()
```

```
Model: "model"
_________________________________________________________________
 Layer (type)                Output Shape              Param #   
=================================================================
 input_1 (InputLayer)        [(None, None)]            0         
                                                                 
 tf.one_hot (TFOpLambda)     (None, None, 10000)       0         
                                                                 
 bidirectional (Bidirection  (None, 64)                2568448   
 al)                                                             
                                                                 
 dropout (Dropout)           (None, 64)                0         
                                                                 
 dense (Dense)               (None, 1)                 65        
                                                                 
=================================================================
Total params: 2568513 (9.80 MB)
Trainable params: 2568513 (9.80 MB)
Non-trainable params: 0 (0.00 Byte)
_________________________________________________________________
```

Training a first basic sequence model

```
callbacks = [
    keras.callbacks.ModelCheckpoint("one_hot_bidir_lstm.keras",
                                    save_best_only=True)
]
model.fit(int_train_ds, validation_data=int_val_ds, epochs=10, callbacks=callbacks)
model = keras.models.load_model("one_hot_bidir_lstm.keras")
print(f"Test acc: {model.evaluate(int_test_ds)[1]:.3f}")
```

```
Epoch 1/10
4/4 [==============================] - 19s 4s/step - loss: 0.6934 - accuracy: 0.4800 - val_loss: 0.6931 - val_accuracy: 0.5056
Epoch 2/10
4/4 [==============================] - 11s 4s/step - loss: 0.6878 - accuracy: 0.7400 - val_loss: 0.6931 - val_accuracy: 0.5065
Epoch 3/10
4/4 [==============================] - 10s 3s/step - loss: 0.6821 - accuracy: 0.8300 - val_loss: 0.6931 - val_accuracy: 0.5105
Epoch 4/10
4/4 [==============================] - 10s 3s/step - loss: 0.6752 - accuracy: 0.8400 - val_loss: 0.6933 - val_accuracy: 0.5118
Epoch 5/10
4/4 [==============================] - 10s 3s/step - loss: 0.6679 - accuracy: 0.8400 - val_loss: 0.6934 - val_accuracy: 0.5116
```

```
Epoch 6/10
4/4 [==============================] - 10s 3s/step - loss: 0.6580 - accuracy: 0.8500 - val_loss: 0.6938 - val_accuracy: 0.5094
Epoch 7/10
4/4 [==============================] - 6s 2s/step - loss: 0.6445 - accuracy: 0.8400 - val_loss: 0.6946 - val_accuracy: 0.5021
Epoch 8/10
4/4 [==============================] - 10s 3s/step - loss: 0.6102 - accuracy: 0.8100 - val_loss: 0.7130 - val_accuracy: 0.5000
Epoch 9/10
4/4 [==============================] - 11s 4s/step - loss: 0.5630 - accuracy: 0.6300 - val_loss: 0.6895 - val_accuracy: 0.5463
Epoch 10/10
4/4 [==============================] - 11s 4s/step - loss: 0.4978 - accuracy: 0.8800 - val_loss: 0.6833 - val_accuracy: 0.5569
782/782 [==============================] - 17s 20ms/step - loss: 0.6843 - accuracy: 0.5538
Test acc: 0.554
```

Understanding word embeddings

Learning word embeddings with the Embedding layer

Instantiating an `Embedding` layer

```
embedding_layer = layers.Embedding(input_dim=max_tokens, output_dim=256)
```

Model that uses an `Embedding` layer trained from scratch

```
inputs = keras.Input(shape=(None,), dtype="int64")
embedded = layers.Embedding(input_dim=max_tokens, output_dim=256)(inputs)
x = layers.Bidirectional(layers.LSTM(32))(embedded)
x = layers.Dropout(0.5)(x)
outputs = layers.Dense(1, activation="sigmoid")(x)
model = keras.Model(inputs, outputs)
model.compile(optimizer="adam",
              loss="binary_crossentropy",
              metrics=["accuracy"])
model.summary()

callbacks = [
    keras.callbacks.ModelCheckpoint("embeddings_bidir_gru.keras",
                                    save_best_only=True)
]
model.fit(int_train_ds, validation_data=int_val_ds, epochs=10, callbacks=callbacks)
model = keras.models.load_model("embeddings_bidir_gru.keras")
print(f"Test acc: {model.evaluate(int_test_ds)[1]:.3f}")
```

```
Model: "model_2"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 input_3 (InputLayer)        [(None, None)]            0

 embedding_2 (Embedding)     (None, None, 256)         2560000

 bidirectional_2 (Bidirecti  (None, 64)                73984
 onal)

 dropout_2 (Dropout)         (None, 64)                0

 dense_2 (Dense)             (None, 1)                 65

=================================================================
Total params: 2634049 (10.05 MB)
Trainable params: 2634049 (10.05 MB)
Non-trainable params: 0 (0.00 Byte)
_________________________________________________________________
Epoch 1/10
4/4 [==============================] - 10s 1s/step - loss: 0.6922 - accuracy: 0.4800 - val_loss: 0.6941 - val_accuracy: 0.5062
Epoch 2/10
4/4 [==============================] - 3s 886ms/step - loss: 0.6727 - accuracy: 0.7600 - val_loss: 0.6973 - val_accuracy: 0.5085
Epoch 3/10
4/4 [==============================] - 3s 976ms/step - loss: 0.6528 - accuracy: 0.8500 - val_loss: 0.7008 - val_accuracy: 0.5095
Epoch 4/10
4/4 [==============================] - 3s 1s/step - loss: 0.6299 - accuracy: 0.8300 - val_loss: 0.7049 - val_accuracy: 0.5024
Epoch 5/10
4/4 [==============================] - 3s 974ms/step - loss: 0.6059 - accuracy: 0.8600 - val_loss: 0.7068 - val_accuracy: 0.4976
Epoch 6/10
4/4 [==============================] - 3s 1s/step - loss: 0.5468 - accuracy: 0.8600 - val_loss: 0.7119 - val_accuracy: 0.4970
Epoch 7/10
```

```
4/4 [==============================] - 3s 796ms/step - loss: 0.4761 - accuracy: 0.8800 - val_loss: 0.7166 - val_accuracy: 0.4966
Epoch 8/10
4/4 [==============================] - 3s 819ms/step - loss: 0.3532 - accuracy: 0.9500 - val_loss: 0.7499 - val_accuracy: 0.5485
Epoch 9/10
4/4 [==============================] - 3s 845ms/step - loss: 0.2428 - accuracy: 0.9700 - val_loss: 0.7512 - val_accuracy: 0.5686
Epoch 10/10
4/4 [==============================] - 4s 1s/step - loss: 0.1557 - accuracy: 1.0000 - val_loss: 0.8265 - val_accuracy: 0.5828
782/782 [==============================] - 8s 8ms/step - loss: 0.6944 - accuracy: 0.5066
Test acc: 0.507
```

Understanding padding and masking

Using an `Embedding` layer with masking enabled

```
inputs = keras.Input(shape=(None,), dtype="int64")
embedded = layers.Embedding(
    input_dim=max_tokens, output_dim=256, mask_zero=True)(inputs)
x = layers.Bidirectional(layers.LSTM(32))(embedded)
x = layers.Dropout(0.5)(x)
outputs = layers.Dense(1, activation="sigmoid")(x)
model = keras.Model(inputs, outputs)
model.compile(optimizer="adam",
              loss="binary_crossentropy",
              metrics=["accuracy"])
model.summary()

callbacks = [
    keras.callbacks.ModelCheckpoint("embeddings_bidir_gru_with_masking.keras",
                                    save_best_only=True)
]
model.fit(int_train_ds, validation_data=int_val_ds, epochs=10, callbacks=callbacks)
model = keras.models.load_model("embeddings_bidir_gru_with_masking.keras")
print(f"Test acc: {model.evaluate(int_test_ds)[1]:.3f}")
```

```
Model: "model_3"
_________________________________________________________________
 Layer (type)                Output Shape              Param #   
=================================================================
 input_4 (InputLayer)        [(None, None)]            0         
                                                                 
 embedding_3 (Embedding)     (None, None, 256)         2560000   
                                                                 
 bidirectional_3 (Bidirecti  (None, 64)                73984     
 onal)                                                           
                                                                 
 dropout_3 (Dropout)         (None, 64)                0         
                                                                 
 dense_3 (Dense)             (None, 1)                 65        
                                                                 
=================================================================
Total params: 2634049 (10.05 MB)
Trainable params: 2634049 (10.05 MB)
Non-trainable params: 0 (0.00 Byte)
_________________________________________________________________
Epoch 1/10
4/4 [==============================] - 17s 3s/step - loss: 0.6939 - accuracy: 0.4800 - val_loss: 0.6935 - val_accuracy: 0.4934
Epoch 2/10
4/4 [==============================] - 6s 2s/step - loss: 0.6789 - accuracy: 0.8600 - val_loss: 0.6932 - val_accuracy: 0.4983
Epoch 3/10
4/4 [==============================] - 4s 1s/step - loss: 0.6652 - accuracy: 0.9600 - val_loss: 0.6928 - val_accuracy: 0.5042
Epoch 4/10
4/4 [==============================] - 6s 2s/step - loss: 0.6490 - accuracy: 0.9800 - val_loss: 0.6924 - val_accuracy: 0.5151
Epoch 5/10
4/4 [==============================] - 6s 2s/step - loss: 0.6191 - accuracy: 1.0000 - val_loss: 0.6916 - val_accuracy: 0.5263
Epoch 6/10
4/4 [==============================] - 4s 1s/step - loss: 0.5794 - accuracy: 1.0000 - val_loss: 0.6901 - val_accuracy: 0.5337
Epoch 7/10
4/4 [==============================] - 3s 1s/step - loss: 0.5200 - accuracy: 1.0000 - val_loss: 0.6865 - val_accuracy: 0.5516
Epoch 8/10
4/4 [==============================] - 6s 2s/step - loss: 0.4125 - accuracy: 1.0000 - val_loss: 0.6834 - val_accuracy: 0.5507
Epoch 9/10
4/4 [==============================] - 6s 2s/step - loss: 0.2656 - accuracy: 0.9900 - val_loss: 0.6640 - val_accuracy: 0.5987
Epoch 10/10
4/4 [==============================] - 6s 2s/step - loss: 0.1340 - accuracy: 1.0000 - val_loss: 0.8027 - val_accuracy: 0.5499
782/782 [==============================] - 13s 10ms/step - loss: 0.6593 - accuracy: 0.6110
Test acc: 0.611
```

Using pretrained word embeddings

```
!wget http://nlp.stanford.edu/data/glove.6B.zip
!unzip -q glove.6B.zip
```

```
--2024-05-03 03:35:02--  http://nlp.stanford.edu/data/glove.6B.zip
Resolving nlp.stanford.edu (nlp.stanford.edu)... 171.64.67.140
Connecting to nlp.stanford.edu (nlp.stanford.edu)|171.64.67.140|:80... connected.
HTTP request sent, awaiting response... 302 Found
Location: https://nlp.stanford.edu/data/glove.6B.zip [following]
--2024-05-03 03:35:03--  https://nlp.stanford.edu/data/glove.6B.zip
Connecting to nlp.stanford.edu (nlp.stanford.edu)|171.64.67.140|:443... connected.
HTTP request sent, awaiting response... 301 Moved Permanently
Location: https://downloads.cs.stanford.edu/nlp/data/glove.6B.zip [following]
--2024-05-03 03:35:04--  https://downloads.cs.stanford.edu/nlp/data/glove.6B.zip
Resolving downloads.cs.stanford.edu (downloads.cs.stanford.edu)... 171.64.64.22
Connecting to downloads.cs.stanford.edu (downloads.cs.stanford.edu)|171.64.64.22|:443... connected.
HTTP request sent, awaiting response... 200 OK
Length: 862182613 (822M) [application/zip]
Saving to: ‘glove.6B.zip’

glove.6B.zip        100%[===================>] 822.24M  2.31MB/s    in 4m 46s

2024-05-03 03:39:50 (2.88 MB/s) - ‘glove.6B.zip’ saved [862182613/862182613]
```

Parsing the GloVe word-embeddings file

```
import numpy as np
path_to_glove_file = "glove.6B.100d.txt"

embeddings_index = {}
with open(path_to_glove_file) as f:
    for line in f:
        word, coefs = line.split(maxsplit=1)
        coefs = np.fromstring(coefs, "f", sep=" ")
        embeddings_index[word] = coefs

print(f"Found {len(embeddings_index)} word vectors.")
```

```
Found 400000 word vectors.
```

Preparing the GloVe word-embeddings matrix

```
embedding_dim = 100

vocabulary = text_vectorization.get_vocabulary()
word_index = dict(zip(vocabulary, range(len(vocabulary))))

embedding_matrix = np.zeros((max_tokens, embedding_dim))
for word, i in word_index.items():
    if i < max_tokens:
        embedding_vector = embeddings_index.get(word)
    if embedding_vector is not None:
        embedding_matrix[i] = embedding_vector
```

```
embedding_layer = layers.Embedding(
    max_tokens,
    embedding_dim,
    embeddings_initializer=keras.initializers.Constant(embedding_matrix),
    trainable=False,
    mask_zero=True,
)
```

Model that uses a pretrained Embedding layer

```
inputs = keras.Input(shape=(None,), dtype="int64")
embedded = embedding_layer(inputs)
x = layers.Bidirectional(layers.LSTM(32))(embedded)
x = layers.Dropout(0.5)(x)
outputs = layers.Dense(1, activation="sigmoid")(x)
model = keras.Model(inputs, outputs)
model.compile(optimizer="adam",
              loss="binary_crossentropy",
              metrics=["accuracy"])
model.summary()

callbacks = [
    keras.callbacks.ModelCheckpoint("glove_embeddings_sequence_model.keras",
                                    save_best_only=True)
]
model.fit(int_train_ds, validation_data=int_val_ds, epochs=10, callbacks=callbacks)
model = keras.models.load_model("glove_embeddings_sequence_model.keras")
print(f"Test acc: {model.evaluate(int_test_ds)[1]:.3f}")
```

```
Model: "model_4"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 input_5 (InputLayer)        [(None, None)]            0

 embedding_4 (Embedding)     (None, None, 100)         1000000

 bidirectional_4 (Bidirecti  (None, 64)                34048
 onal)

 dropout_4 (Dropout)         (None, 64)                0

 dense_4 (Dense)             (None, 1)                 65

=================================================================
Total params: 1034113 (3.94 MB)
Trainable params: 34113 (133.25 KB)
Non-trainable params: 1000000 (3.81 MB)
_________________________________________________________________
Epoch 1/10
4/4 [==============================] - 32s 8s/step - loss: 0.7077 - accuracy: 0.5500 - val_loss: 0.6932 - val_accuracy: 0.5115
Epoch 2/10
4/4 [==============================] - 19s 6s/step - loss: 0.6852 - accuracy: 0.5700 - val_loss: 0.6926 - val_accuracy: 0.5191
Epoch 3/10
4/4 [==============================] - 14s 5s/step - loss: 0.6732 - accuracy: 0.5900 - val_loss: 0.6914 - val_accuracy: 0.5262
Epoch 4/10
4/4 [==============================] - 15s 5s/step - loss: 0.6747 - accuracy: 0.6200 - val_loss: 0.6905 - val_accuracy: 0.5303
Epoch 5/10
4/4 [==============================] - 19s 6s/step - loss: 0.6498 - accuracy: 0.6500 - val_loss: 0.6903 - val_accuracy: 0.5320
Epoch 6/10
4/4 [==============================] - 14s 5s/step - loss: 0.6345 - accuracy: 0.6300 - val_loss: 0.6895 - val_accuracy: 0.5356
Epoch 7/10
4/4 [==============================] - 5s 2s/step - loss: 0.6369 - accuracy: 0.6700 - val_loss: 0.6903 - val_accuracy: 0.5333
Epoch 8/10
4/4 [==============================] - 23s 8s/step - loss: 0.6366 - accuracy: 0.6700 - val_loss: 0.6888 - val_accuracy: 0.5378
Epoch 9/10
4/4 [==============================] - 5s 2s/step - loss: 0.5820 - accuracy: 0.7200 - val_loss: 0.6895 - val_accuracy: 0.5384
Epoch 10/10
4/4 [==============================] - 5s 2s/step - loss: 0.6075 - accuracy: 0.7100 - val_loss: 0.6915 - val_accuracy: 0.5380
782/782 [==============================] - 14s 13ms/step - loss: 0.6882 - accuracy: 0.5436
Test acc: 0.544
```