Учреждение образования
«БЕЛОРУССКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ
ИНФОРМАТИКИ И РАДИОЭЛЕКТРОНИКИ»

Кафедра информатики

Отчет по предмету:
«Машинное обучение»
По лабораторной работе №7
«Рекуррентные нейронные сети для анализа текста»

Выполнил: Сенькович Дмитрий Сергеевич
магистрант кафедры информатики
группы №858642

Проверил: Стержанов Максим Валерьевич
доцент, кандидат технических наук

Минск 2020

Оглавление

1 Постановка задачи

Имеется 50000 отзывов с IMDB, разделенные поровну на обучающую и контрольную выборки. Каждая выборка содержит поровну положительных и негативных отзывов. Требуется обучить рекуррентную нейронную сеть для распознавание отзыва как положительного либо отрицательного.

2 Чтение данных и формирование выборок

Покажем, как будем преобразовывать, считывать данные:

```
def read_dataset(dataset_directory):
    dataset = []
    labels = []
    positive_reviews_directory = dataset_directory + POSITIVE_DATASET_DIRECTORY
    for review_file_name in os.listdir(positive_reviews_directory):
        full_file_name = positive_reviews_directory + review_file_name
        dataset.append(open(full_file_name, 'r').read())
        labels.append(1)
    negative_reviews_directory = dataset_directory + NEGATIVE_DATASET_DIRECTORY
    for review_file_name in os.listdir(negative_reviews_directory):
        full_file_name = negative_reviews_directory + review_file_name
        dataset.append(open(full_file_name, 'r').read())
        labels.append(0)
    return np.array(dataset), np.array(labels)



def load_vocabulary():
    vocabulary = {}
    i = 0
    with open('imdb.vocab', 'r') as vocabulary_file:
        for line in vocabulary_file:
            vocabulary[line.replace('\n', '')] = i
            i += 1
    return vocabulary
```

```
def find_words_from_vocabulary_preserving_order(text, vocabulary):
    found_words_codes = []
    for word in text.split(' '):
        if word in vocabulary:
            found_words_codes.append(vocabulary[word])
    return pd.Series(found_words_codes).drop_duplicates().tolist()



def dataset_to_indices(dataset, vocabulary):
    mapped_dataset = []
    for sample in dataset:
        found_words_codes = find_words_from_vocabulary_preserving_order(sample,
vocabulary)
        mapped_dataset.append(found_words_codes)
    return np.array(mapped_dataset)



def save_indices_processed_as_csv(mapped_dataset, labels, output):
    mapped_dataset_in_string = []
    for sample in mapped_dataset:
        mapped_dataset_in_string.append(' '.join(map(str, sample)) if sample
else '')
    dataset = np.column_stack([labels, np.array(mapped_dataset_in_string)])
    np.savetxt(output, dataset, delimiter=',', fmt='%s')
```

```
def load_indices_processed_as_csv(output):

    csv = pd.read_csv(output)

    labels = csv.iloc[:, 0].to_numpy()

    dataset = []

    for sample in csv.iloc[:, 1:].to_numpy():

        dataset.append(list(map(int, sample[0].split(' '))) if
isinstance(sample[0], str) else [])

    return np.array(dataset), labels
```

Чтение и обработка данных включается в себя чтение отзывов как строк, после чего каждый отзыв трансформируется в список индексов слов из словаря. Далее, списки сохраняются в csv файлы для дальнейшего переиспользования.

3 Архитектура и результаты для собственной сети

Приведем нейронную сеть, не использующую pretrained embeddings:

```
    print('Pad sequences (samples x time)')

    x_train = sequence.pad_sequences(mapped_train_dataset, maxlen=MAX_LENGTH)

    x_test = sequence.pad_sequences(mapped_test_dataset, maxlen=MAX_LENGTH)

    print('x_train shape:', x_train.shape)

    print('x_test shape:', x_test.shape)


    resolver = tf.distribute.cluster_resolver.TPUClusterResolver(tpu='grpc://'
+ os.environ['COLAB_TPU_ADDR'])

    tf.config.experimental_connect_to_cluster(resolver)

    tf.tpu.experimental.initialize_tpu_system(resolver)

    strategy = tf.distribute.experimental.TPUStrategy(resolver)

    with strategy.scope():

        print('Build model...')

        model = Sequential()

        model.add(Embedding(VOCABULARY_SIZE, EMBEDDINE_VECTOR_SIZE))

        model.add(Bidirectional(LSTM(LSTM_UNITS_COUNT, dropout=0.2,
recurrent_dropout=0.2)))

        model.add(Dense(1, activation='sigmoid'))


        # try using different optimizers and different optimizer configs

        model.compile(loss='binary_crossentropy',

                      optimizer='adam',

                      metrics=['accuracy'])


        print('Train...')
```

```
        model.fit(x_train, train_labels, batch_size=TPU_BATCH_SIZE,
epochs=EPOCHS_NUMBER)

        score, acc = model.evaluate(x_test, test_labels)

        print('Test accuracy:', acc)
```

Сначала мы дополняем все получившиеся списки индексов до одинаковой длины. Первый слой нейронной сети - embedding - нужен для “кодирования” каждого слова некоторым вектором для изучения сетью взаимоотношений между словами. Второй слой - двунаправленный LSTM. Выходной слой как обычно слой с 1 нейроном и функцией активации sigmoid.

Такая сеть обучается очень быстро до 99% на обучающей выборке, но дает всего 79.89% точности на контрольной выборке.

4 Использование pretrained embeddings

В лабораторной работе предлагается использоваться два pretrained embeddings: glove и word2vec.

Glove embeddings получим следующим образом:

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

!unzip glove.6B.zip
```

Далее, приведем, как будем использовать embeddings:

```
    embeddings_index = {}

    f = open(os.path.join(BASE_PATH, 'glove.6B.300d.txt'))

    for line in f:

        values = line.split()

        word = values[0]

        coefs = np.asarray(values[1:], dtype='float32')

        embeddings_index[word] = coefs

    f.close()


    embedding_matrix = np.zeros((VOCABULARY_SIZE, EMBEDDING_VECTOR_SIZE))

    asd = 0

    for word, i in vocabulary.items():

        embedding_vector = embeddings_index.get(word)

        if embedding_vector is not None:

            # words not found in embedding index will be all-zeros.

            embedding_matrix[i] = embedding_vector
```

Здесь каждому слову будет соответствовать вектор из 300 чисел, строится матрица embeddings, в которой каждому слову будем соответствовать вектор и скачанных embeddings.

Покажем, как будет теперь выглядеть embedding слой:

```
model.add(Embedding(VOCABULARY_SIZE,
                    EMBEDDING_VECTOR_SIZE,
                    weights=[embedding_matrix],
                    input_length=MAX_LENGTH,
                    trainable=False))
```

Единственная разница - использование pretrained embedding matrix. Такое решение дало 85.31% точности на контрольной выборке.

Теперь покажем использование word2vec. Получать их будем следующим образом:

```
!wget
http://wikipedia2vec.s3.amazonaws.com/models/en/2018-04-20/enwiki_20180420_300d
.txt.bz2

!bzip2 -dk enwiki_20180420_300d.txt.bz2
```

Здесь также на каждое слово отводится вектор размерности 300. Embedding layer остается таким же, но немного другим образом формируется embedding matrix:

```
embeddings_index = {}
f = open(os.path.join(BASE_PATH, 'enwiki_20180420_300d.txt'))
next(f)
for line in f:
    values = line.split()
    split_index = len(values) - 300
    word = ' '.join(values[:split_index])
    coefs = np.asarray(values[split_index:], dtype='float32')
```

```
        embeddings_index[word] = coefs
    f.close()


    embedding_matrix = np.zeros((VOCABULARY_SIZE,
EMBEDDING_VECTOR_SIZE))
```

Такое решение дает 85.23% точности.

5 Эксперименты

Для экспериментов было решено выбрать glove embedding, так как их обработка гораздо быстрее, чем word2vec (размер word2vec намного больше).

В экспериментах использовались и несколько слоев lstm и conv1d слои, и полносвязные слои и т.д. Ниже представлены результаты экспериментов с описанием архитектуры сети:

Accuracy (test, %)	Кол-во эпох	Кол-во LSTM слоев	Кол-во нейронов LSTM	LSTM dropout	Кол-во Dense слоев	Кол-во нейронов Dense	Dense dropout	Кол-во Conv1d слоев	Кол-во фильтров	Kernel size	Conv1d до LSTM
84.62	100	1	128	0.2	1	128	0.2	0	-	-	-
84.63	100	1	128	0.2	1	128	0.5	0	-	-	-
84.01	100	1	128	0.2	1	256	0.5	0	-	-	-
84.85	100	1	128	0.2	1	512	0.5	0	-	-	-
84.89	100	1	128	0.2	1	1024	0.5	0	-	-	-
84.08	100	1	256	0.3	1	1024	0.5	0	-	-	-
84.66	100	1	256	0.4	1	1024	0.5	0	-	-	-
84.99	100	1	256	0.5	1	1024	0.5	0	-	-	-

84.71	100	1	256	0.6	1	1024	0.5	0	-	-	-
85.03	100	1	256	0.7	1	1024	0.5	0	-	-	-
85.62	100	1	256	0.7	1	1024	0.5	0	-	-	-
87.07	100	2	256	0.5	1	1024	0.5	0	-	-	-
86.33	100	2	256	0.5	1	1024	0.5	0	-	-	-
86.53	100	2	256	0.5	2	1024	0.5	0	-	-	-
81.18	100	2	256	0.5	2	1024	0.5	1	32	8	да
81.58	100	2	256	0.5	2	1024	0.5	1	64	8	да
81.32	100	2	256	0.5	2	1024	0.5	1	128	8	да
86.97	100	2	256	0.5	2	1024	0.5	1	128	8	нет
86.85	100	2	256	0.5	2	1024	0.5	1	128	10	нет
86.43	100	2	256	0.5	2	1024	0.5	1	128	11	нет
86.05	100	1	256	0.5	2	1024	0.5	1	128	11	нет
86.23	100	1	256	0.5	2	1024	0.5	1	48	11	нет
86.5	100	1	256	0.5	2	1024	0.5	пр	-	-	нет

0								лр			
85.57	200	1	256	0.5	2	1024	0.5	пр лр	-	-	нет
85.95	200	1	256	0.7	2	1024	0.5	пр лр	-	-	нет
85.62	300	1	256	0.7	2	1024	0.5	пр лр	-	-	нет
84.60	300	1	256	0.4	2	1024	0.5	пр лр	-	-	нет
85.27	200	1	512	0.5	2	1024	0.5	пр лр	-	-	нет
86.27	200	2	512	0.5	2	1024	0.5	пр лр	-	-	нет

Как видим, наилучший результат получен в без сверточных слоев с двумя LSTM слоями по 256 нейронов с dropout 0.5 с 2 полносвязными слоями по 1024 нейронов с dropout 0.5 с обучением в течение 100 эпох. Обозначение “пр лр” означает, что был использован фрагмент сверточных сетей из предыдущих 3 лабораторных работ (8 сверточных слоев с различными кол-во нейронов и т.д.).

6 Готовые сети

В лабораторной работе предлагается использовать и сравнить результаты с какой-нибудь готовой сетью, например, DeepMoji.

Было интересно получить какой-нибудь хороший результат, поэтому изначально были попытки воспроизвести обучение AWD сети из курса Fast AI на Torch, которая дала более 94% правильных классификаций. Но из-за того, что коду уже 3 года, а конкретных версий используемых зависимостей нет, несколько часов не увенчались успехами из-за очередной ошибки в коде в самой модели.

Покажем установки сети DeepMoji:

```
!git clone https://github.com/bfelbo/DeepMoji.git

%cd DeepMoji/

!python scripts/download_weights.py

!pip install numpy==1.16.2
```

И приведем код использования самой сети:

```
nb_tokens = 20000

maxlen = 80

batch_size = 512


print('Loading data...')

(X_train, y_train), (X_test, y_test) = imdb.load_data(num_words=nb_tokens)

print(len(X_train), 'train sequences')

print(len(X_test), 'test sequences')


print('Pad sequences (samples x time)')

X_train = sequence.pad_sequences(X_train, maxlen=maxlen)
```

```
X_test = sequence.pad_sequences(X_test, maxlen=maxlen)
print('X_train shape:', X_train.shape)
print('X_test shape:', X_test.shape)

print('Build model...')
model = deepmoji_emojis(maxlen, PRETRAINED_PATH)
model.summary()
print(model.layers)
# model.trainable = False
# for layer in model.layers:
#     layer.trainable = False
x = model.layers[-2].output
x = Dense(1024)(x)
x = Dropout(1024)(x)
x = Dense(1024)(x)
x = Dropout(1024)(x)
predictions = Dense(1, activation = "sigmoid")(x)
model = Model(inputs = model.input, outputs = predictions)
model.summary()

model.compile(loss='binary_crossentropy',
              optimizer='adam',
              metrics=['accuracy'])

model.fit(X_train, y_train, batch_size=batch_size, epochs=EPOCHS_NUMBER)

score, acc = model.evaluate(X_test, y_test, batch_size=batch_size)
```

```
print('Test score:', score)

print('Test accuracy:', acc)
```

К сожалению, пришлось использовать обучение на Python 2 на GPU, поэтому как следует поэкспериментировать не получилось (одна эпоха обучается в течение порядка 90 секунд). Обучение без изменений весов за не более 10 эпох дает порядка 67% точности, поэтому обучение происходило с дообучением весов оригинальной сети. Сеть уже на 3 эпохе обучается до 99-100% точности, потом точность падает до 70% и далее снова увеличивается. Максимальная полученная точность около 83%.