МИНОБРНАУКИ РОССИИ
САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ
ЭЛЕКТРОТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ
«ЛЭТИ» ИМ. В.И. УЛЬЯНОВА (ЛЕНИНА)
Кафедра МО ЭВМ

ЛАБОРАТОРНАЯ РАБОТА №6

по дисциплине «Искусственные нейронные сети»

Тема: «Прогноз успеха фильмов по обзорам»

Студентка гр. 7381 _____ Машина Ю.Д.

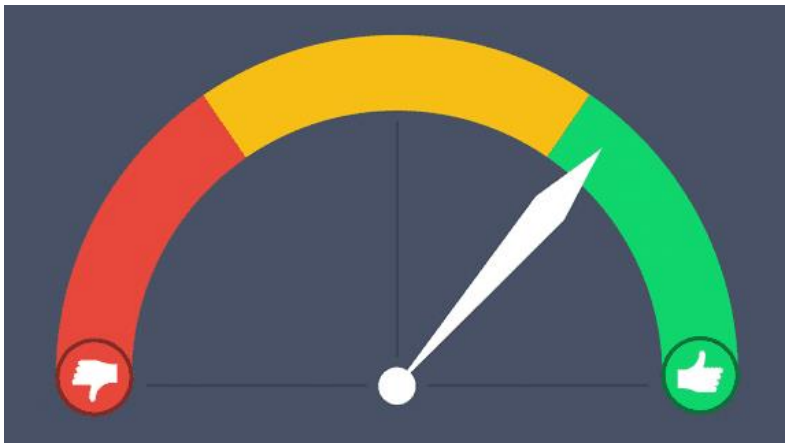
Преподаватель _____ Жукова Н. А.

Санкт-Петербург

2020

Цели.

Прогноз успеха фильмов по обзорам (Predict Sentiment From Movie Reviews)



Задачи.

- Ознакомиться с задачей регрессии
- Изучить способы представления текста для передачи в ИНС
- Достигнуть точность прогноза не менее 95%

Выполнение работы.

1. Построить и обучить нейронную сеть для обработки текста.

В поисках решения проблемы повышения точности до 95% я прочитала, что для этой цели подходит использование слоя Embedding, особенно совместно с РНС GRU, но его использование ни в модели такой простой архитектуры, как

```
model.add(Embedding(input_dim=max_features,
output_dim=embed_size, input_length=10000))
model.add(Flatten())
model.add(Dense(1, activation="sigmoid"))
```
(1)

, ни с использованием простой рекуррентной сети (SimpleRNN слой) или GRU/LSTM слои в том числе с параметром `return_sequences = True`. Точность даже обучения не поднималась выше 60% на первой эпохе, обучение длилось долго: больше 3-х часов на эпоху, после чего я прекращала процесс, в связи с чем графиков обучения нет.

`input_dim` я брала равным размеру словаря, то есть т.к. всего примерно 9999 уникальных слов, я брала 10000. `output_dim` я сначала брала равным 32, потом нашла совет при размере словаря в 10000 ставить 128-256, а вообще надо брать его настолько большим, насколько возможно, при этом сохраняя validation performance.

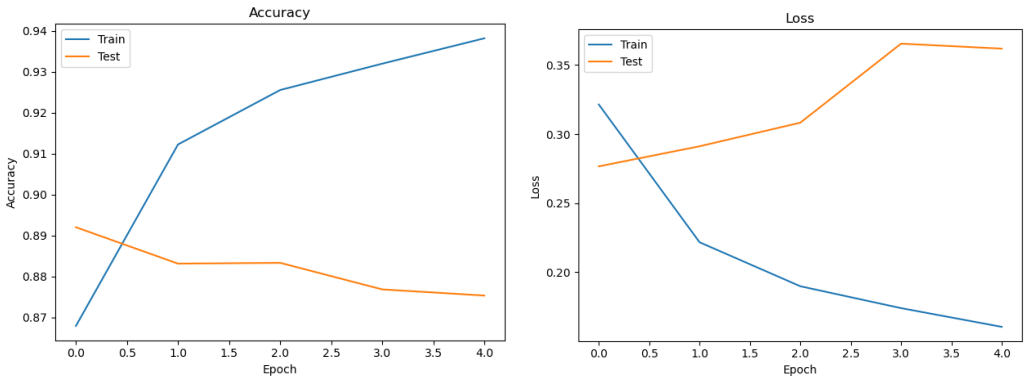


Рисунок 1 — Результаты (точность и потери на обучении и тестах) обучения модели архитектуры (1) при `output_dim=32`.

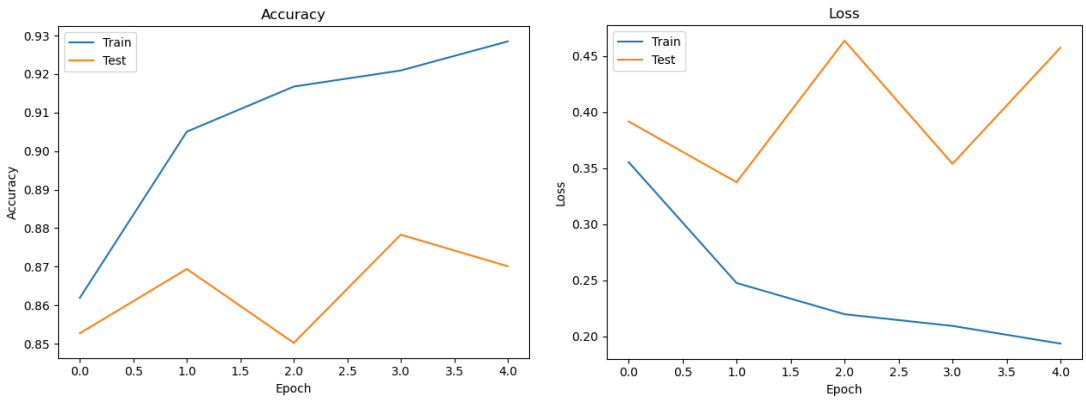


Рисунок 2 — Результаты обучения модели архитектуры (1) при `output_dim=256`.

Как видно, это не сработало, так что я убрала Embedding. Тут нужна нелинейная функция активации, и я нашла относительно новую, предложенную Google Brain Team, которая обещается работать так же, если не лучше ReLU в некоторых задачах. Пробую на следующей архитектуре (`batch_size=10`):

```
    model.add(Dense(64, activation=activation,
input_shape=(10000,)))
    model.add(Dropout(0.3, noise_shape=None, seed=13))
    model.add(Dense(64, activation=activation))
```

```
model.add(Dropout(0.3, noise_shape=None, seed=22))
model.add(Dense(64, activation=activation))
model.add(Dense(1, activation="sigmoid"))
```

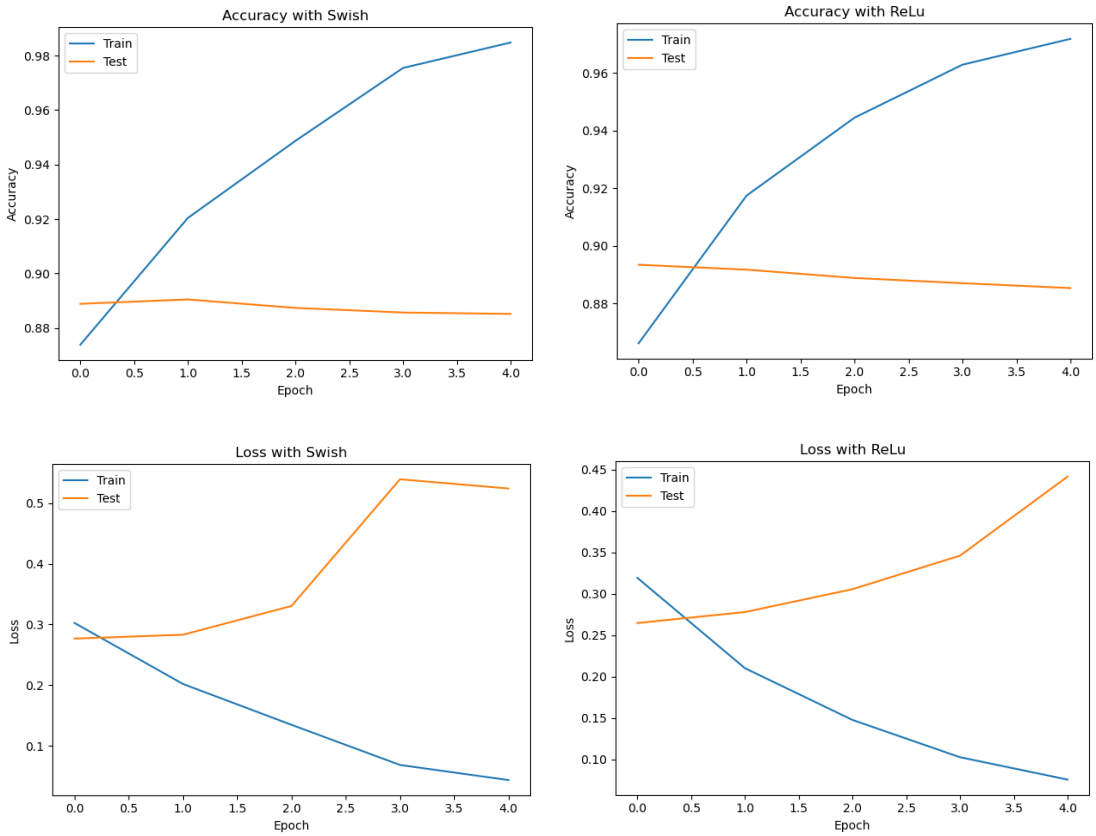


Рисунок 3 — Сравнение результатов обучения при функциях активации Swish и ReLU. Видна тенденция к переобучению на 2-й эпохе в обоих случаях.

После нескольких тестов мне стало видно, что точность обучения при Swish имеет тенденцию быть повыше при меньшем batch_size, но точность на тестах отличается слишком несущественно, а у ReLU она даже несколько выше.

Пробую на такой же архитектуре (но `batch_size=200`):

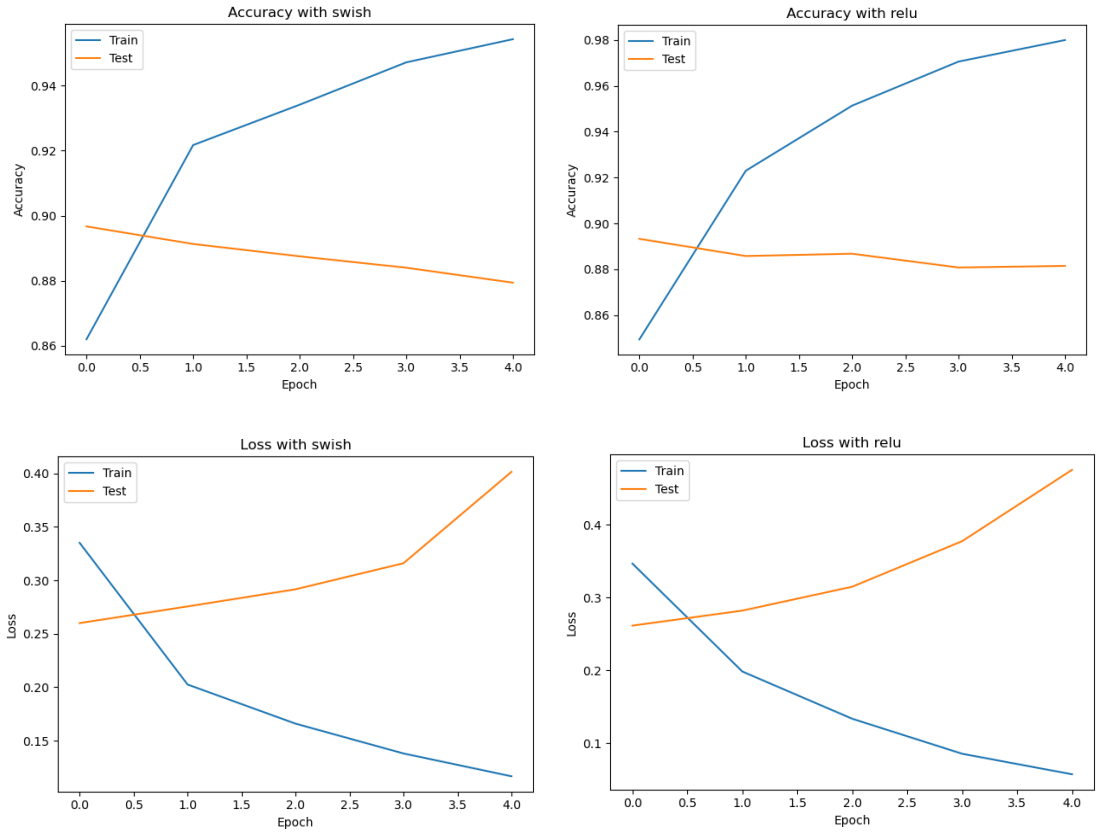


Рисунок 3 — Сравнение результатов обучения при функциях активации Swish (пиковая точность на тестах – 0.8967) и ReLU (пиковая точность на тестах – 0.8933). Видна тенденция к переобучению.

Как видно, точность обучения при Swish имеет тенденцию быть повыше при большем batch_size, но точность на тестах отличается слишком несущественно, но у ReLU она несколько ниже.

Теперь на этой же архитектуре попробую другие нелинейные функции активации (т.е. буду варьировать только `activation`).

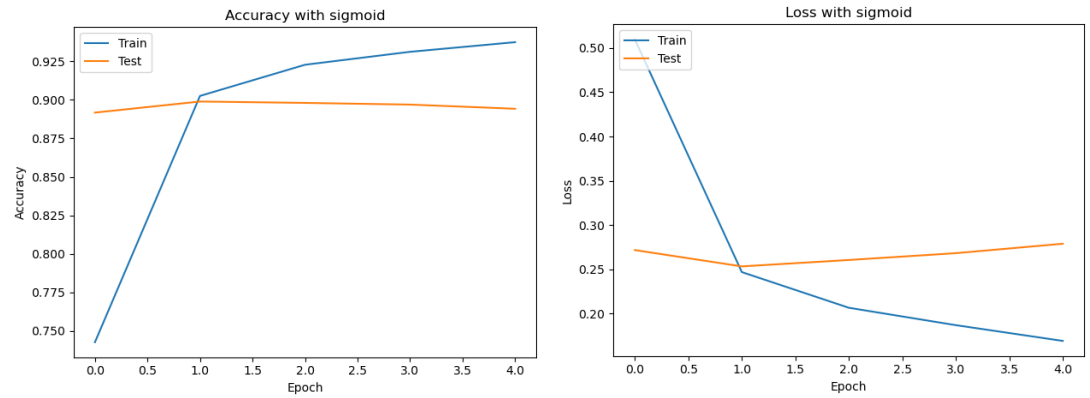


Рисунок 4 — Результаты обучения при функции активации Sigmoid в среднем выглядели так. Пиковая точность на тестах – 0.8993. Кажется, она эффективнее relu и swish при такой архитектуре.

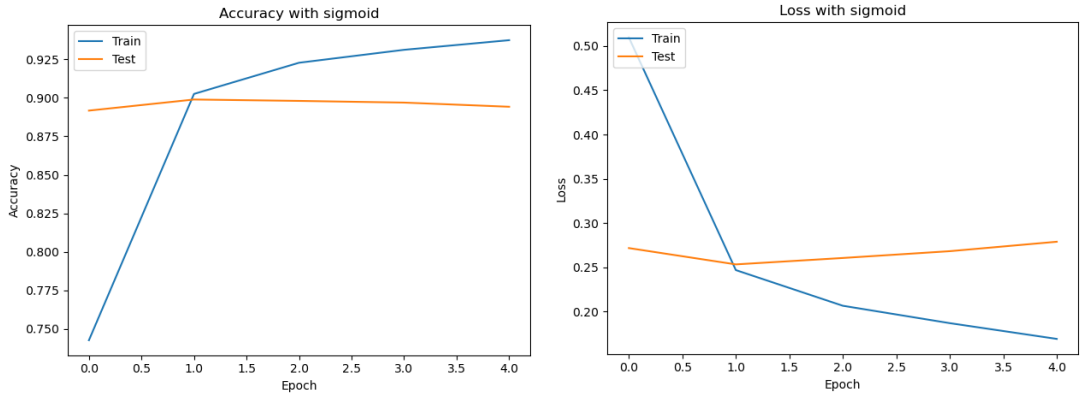


Рисунок 5 — Результаты обучения при функции активации Sigmoid в среднем выглядели так. Пиковая точность на тестах – 0.8993. Видна тенденция к переобучению после 2-й эпохи.Кажется, она эффективнее ReLU и Swish при такой архитектуре.

Следующей будет LeakyReLU, это advanced activation, а такие в Keras доступны только в качестве слоев. Архитектура будет выглядеть так:

```
model.add(Dense(64, activation=None, input_shape=(10000,)))
model.add(LeakyReLU(alpha=0.3))
model.add(Dropout(0.3))
model.add(Dense(64, activation=None))
model.add(LeakyReLU(alpha=0.3))
model.add(Dropout(0.3))
model.add(Dense(64, activation=None))
model.add(LeakyReLU(alpha=0.3))
model.add(Dense(1, activation="sigmoid"))
```

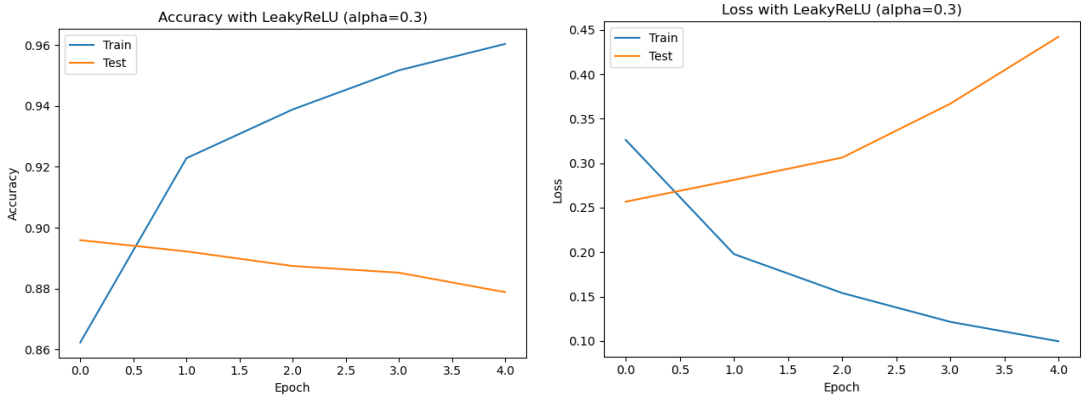


Рисунок 6 — Результаты обучения при функции активации LeakyReLU в среднем выглядели так. Пиковая точность на тестах – 0.8959. Кажется, она эффективнее ReLU при такой архитектуре.

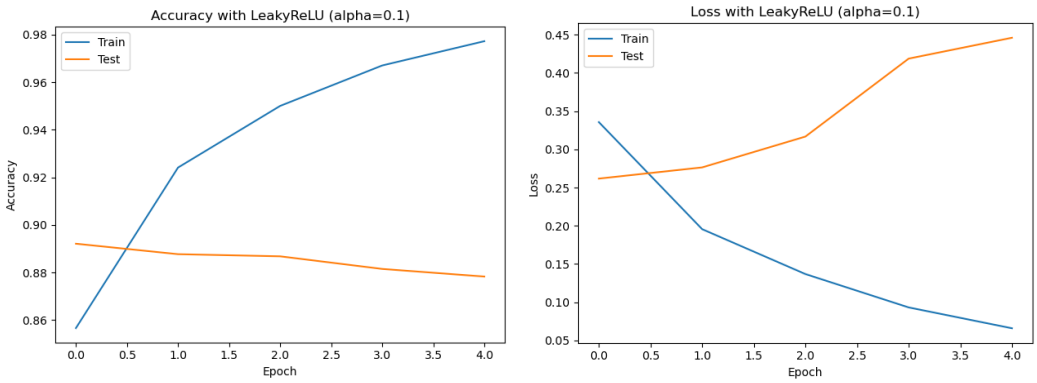


Рисунок 7 — Результаты обучения при функции активации LeakyReLU в среднем выглядели так. Пиковая точность на тестах – 0.8921. Видна тенденция к переобучению.

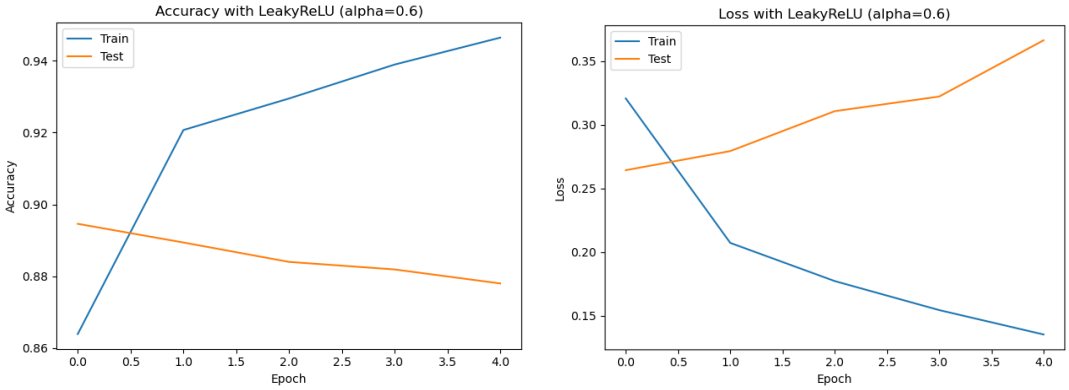


Рисунок 8 — Результаты обучения при функции активации LeakyReLU в среднем выглядели так. Пиковая точность на тестах – 0.8946. Кажется, она эффективнее LeakyReLU при alpha=0.1 при такой архитектуре. Видна тенденция к переобучению.

Следующей будет Parametric ReLU, это advanced activation, а такие в Keras доступны только в качестве слоев. Архитектура будет выглядеть так:

```
model.add(Dense(64, activation=None, input_shape=(10000,)))
model.add(PReLU())
model.add(Dropout(0.3))
model.add(Dense(64, activation=None))
model.add(PReLU())
```

```
model.add(Dropout(0.3))
model.add(Dense(64, activation=None))
model.add(PReLU())
model.add(Dense(1, activation="sigmoid"))
```

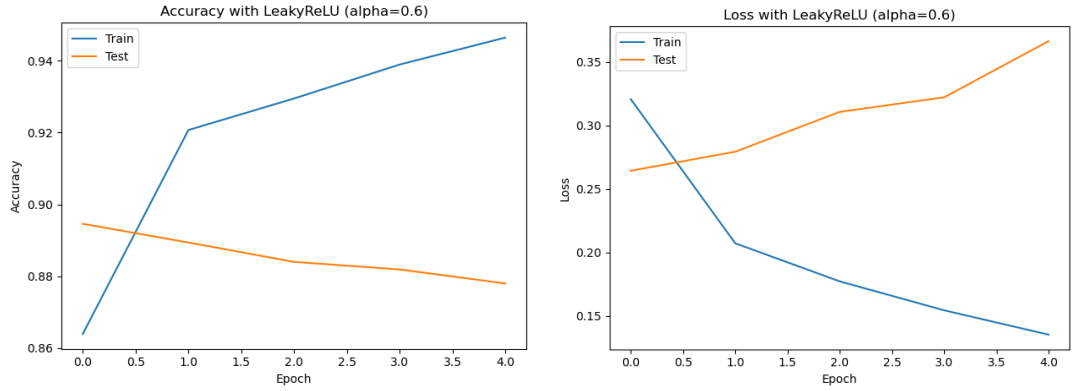


Рисунок 9 — Результаты обучения при функции активации LeakyReLU в среднем выглядели так. Пиковая точность на тестах – 0.8946. Кажется, она эффективнее LeakyReLU при alpha=0.1 при такой архитектуре. Видна тенденция к переобучению.

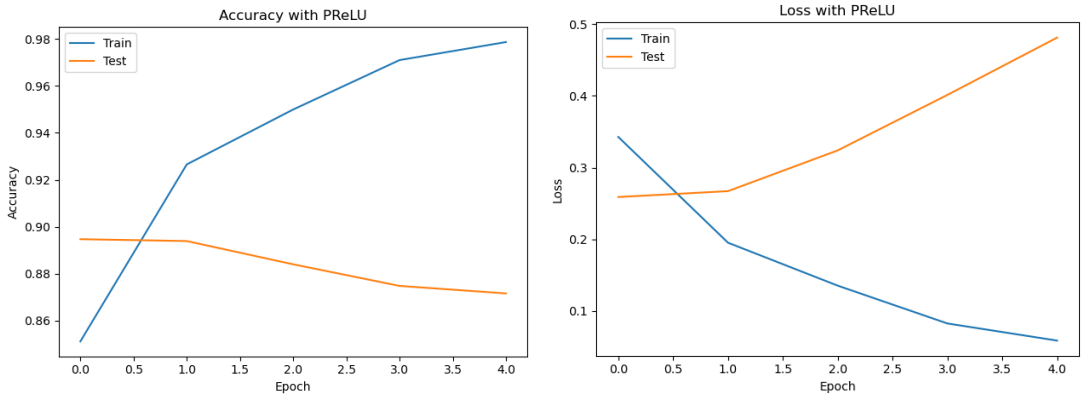


Рисунок 10 — Результаты обучения при функции активации LeakyReLU в среднем выглядели так. Пиковая точность на тестах – 0.8947. Кажется, она эффективнее LeakyReLU при alpha=0.6 и при alpha=0.1 при такой архитектуре. Видна тенденция к переобучению.

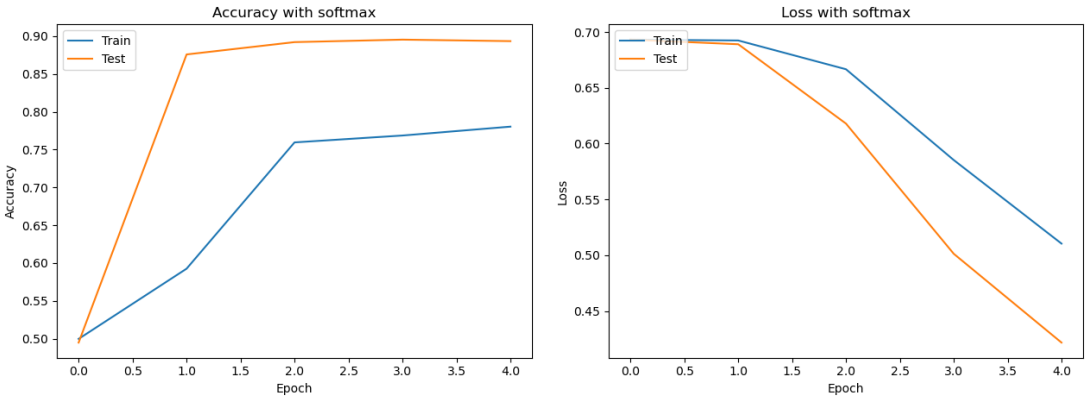


Рисунок 11 — Результаты обучения при функции активации Softmax в среднем выглядели так. Пиковая точность на тестах – 0.8954. Кажется, она справляется лучше прочих, уступая только Sigmoid, LeakyReLU и Swish при такой архитектуре. Меня смущают эти графики. Я не уверена, что такой модели можно доверять, потому что точность на обучении сильно ниже точности на тестах, мне кажется, это значит, что ей везет на тестах, и при этом ей немного помогает обученность.

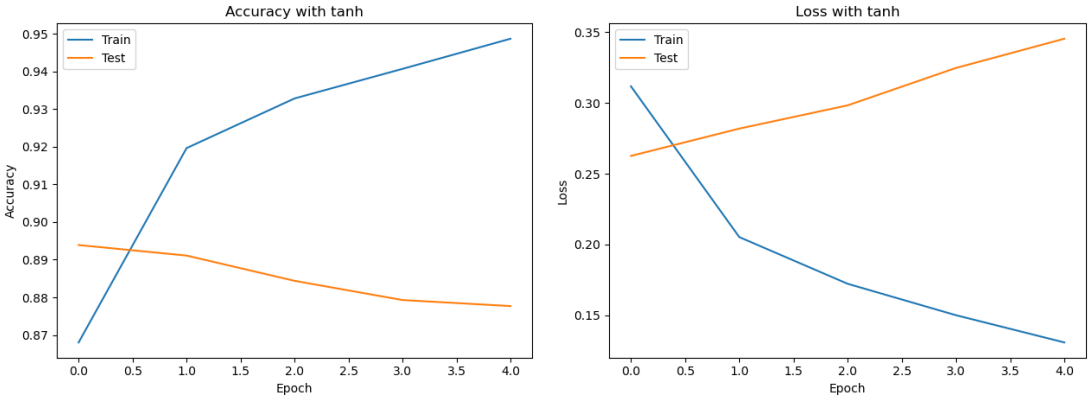
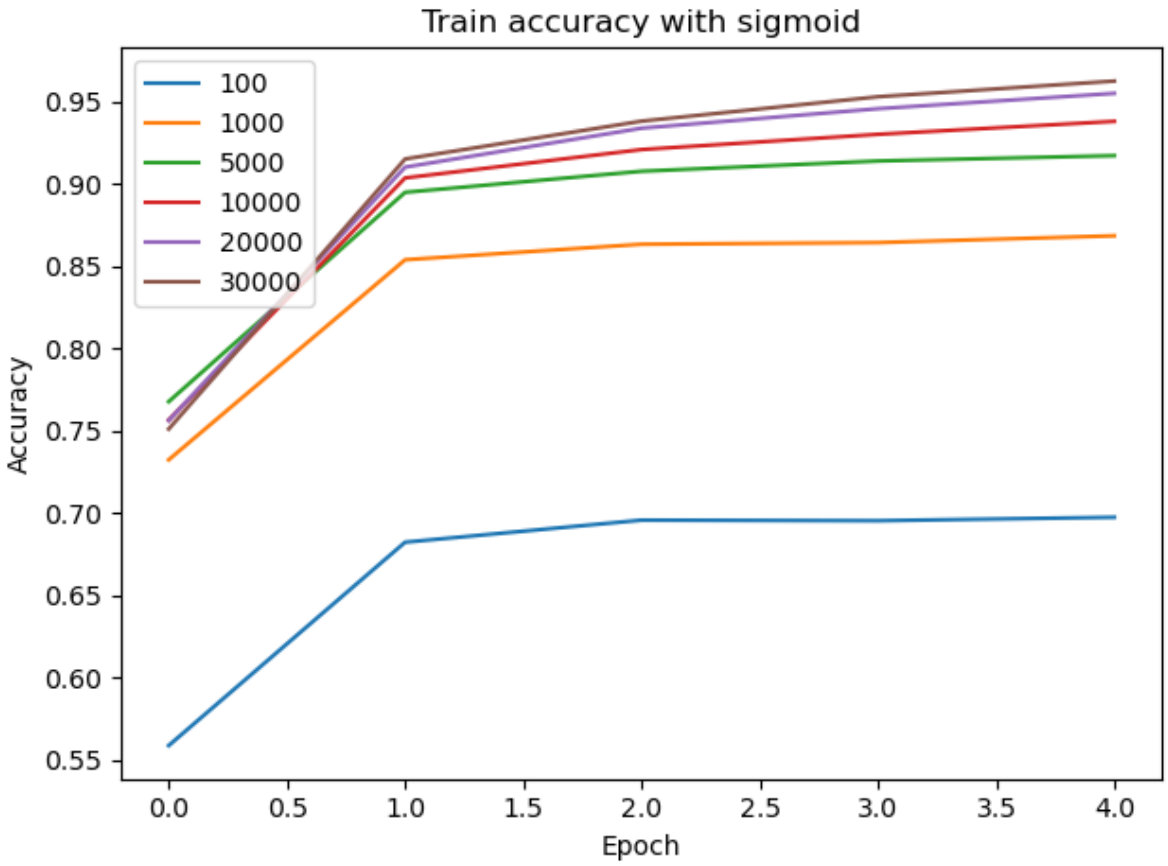
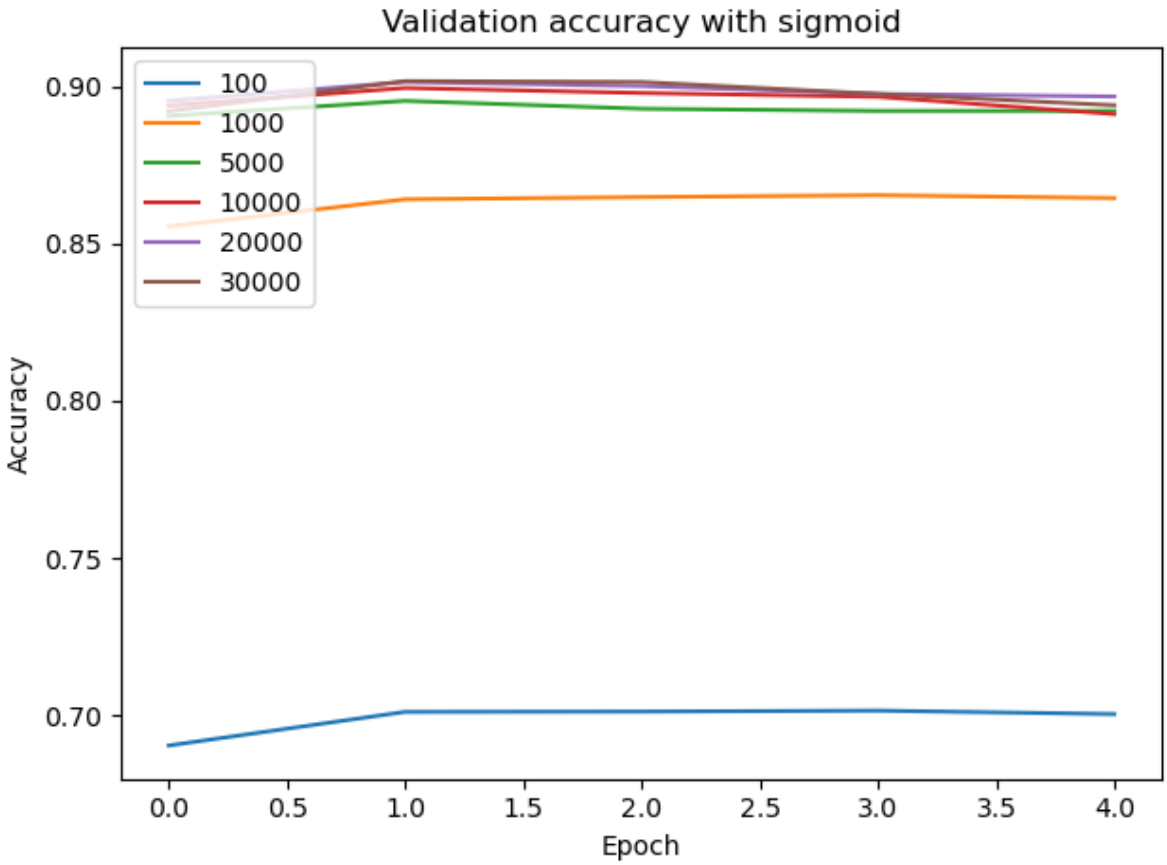
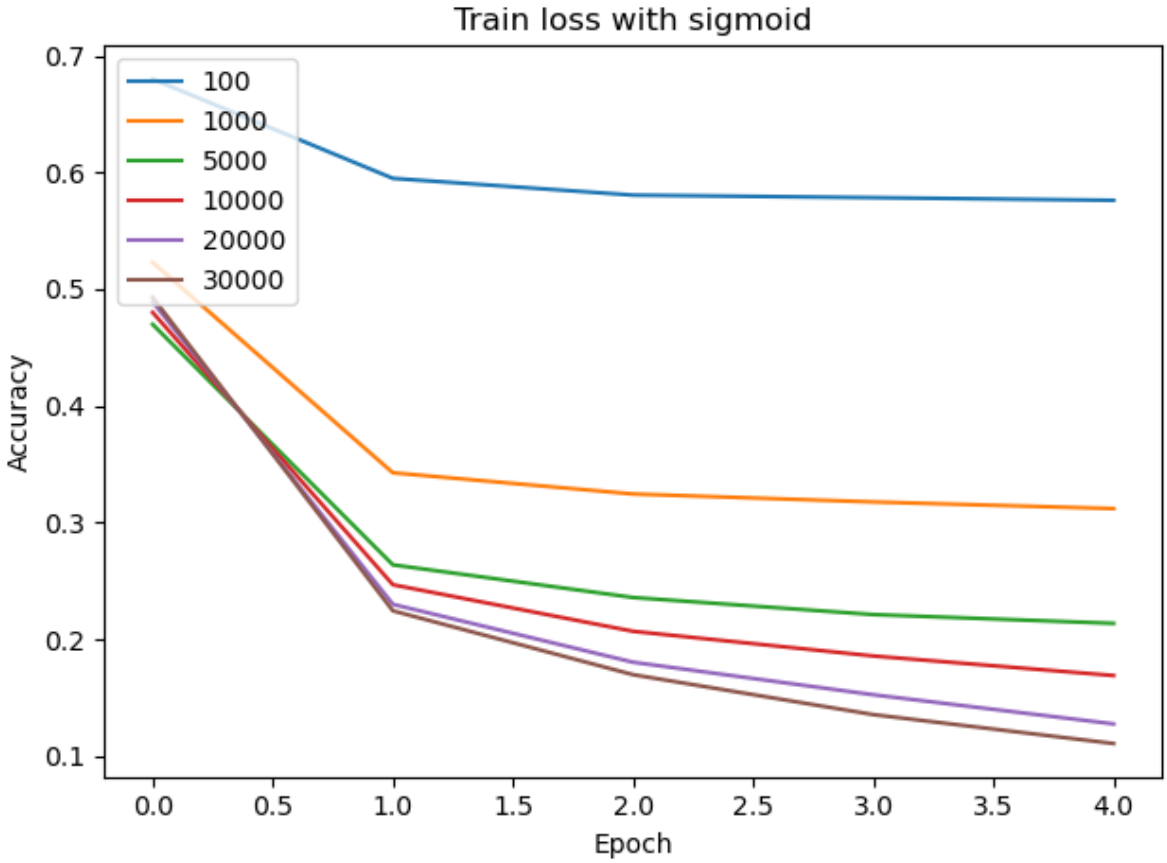


Рисунок 12 — Результаты обучения при функции активации TanH в среднем выглядели так. Пиковая точность на тестах - 0.8939. Кажется, она эффективнее LeakyReLU при alpha=0.1 при такой архитектуре. Сразу же тоже видна тенденция к переобучению.

2. Исследовать результаты при различном размере вектора представления текста.




Validation accuracy with sigmoid
100
1000
5000
10000
20000
30000
0.90
0.85
0.80
0.75
0.70
Accuracy
0.0
0.5
1.0
1.5
2.0
2.5
3.0
3.5
4.0
Epoch


Train loss with sigmoid
100
1000
5000
10000
20000
30000
0.7
0.6
0.5
0.4
0.3
0.2
0.1
Accuracy
0.0
0.5
1.0
1.5
2.0
2.5
3.0
3.5
4.0
Epoch

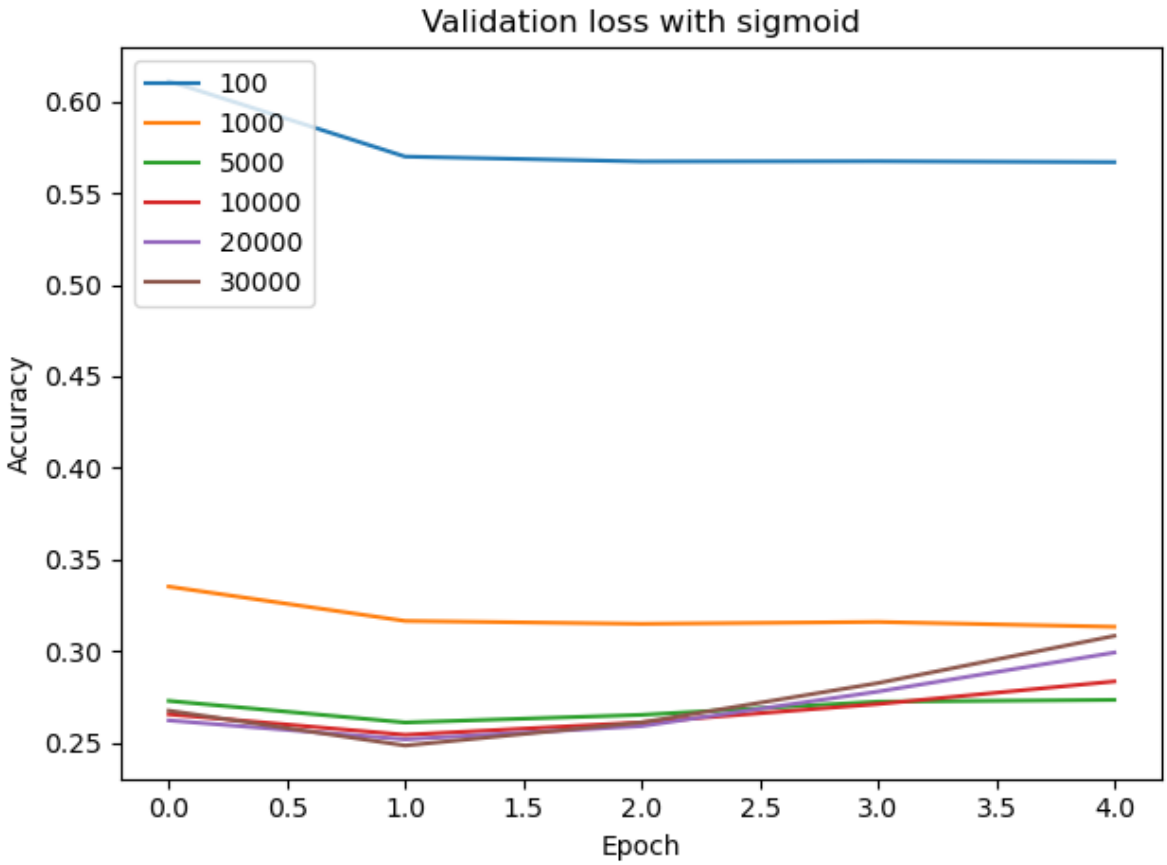
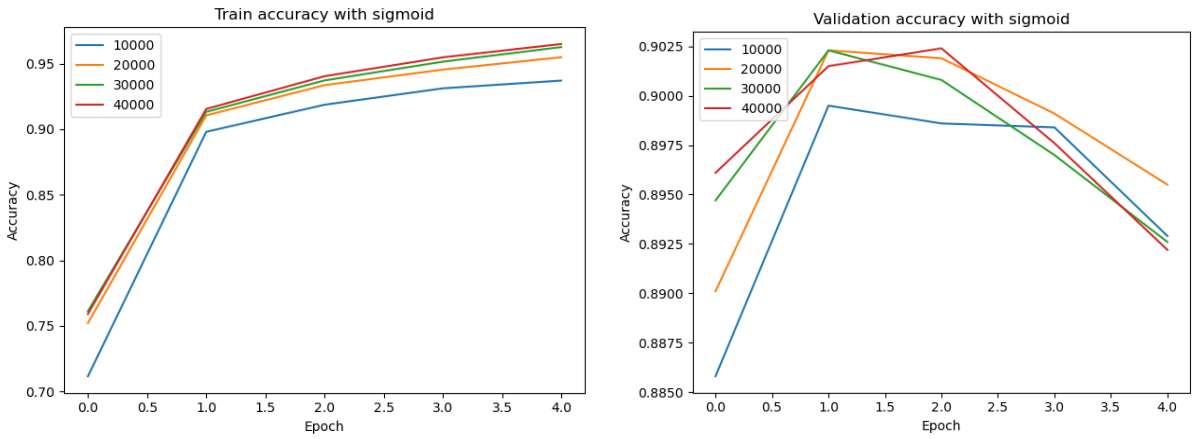


Рисунок 13 — Результаты расследования различных размеров вектора представления текста.



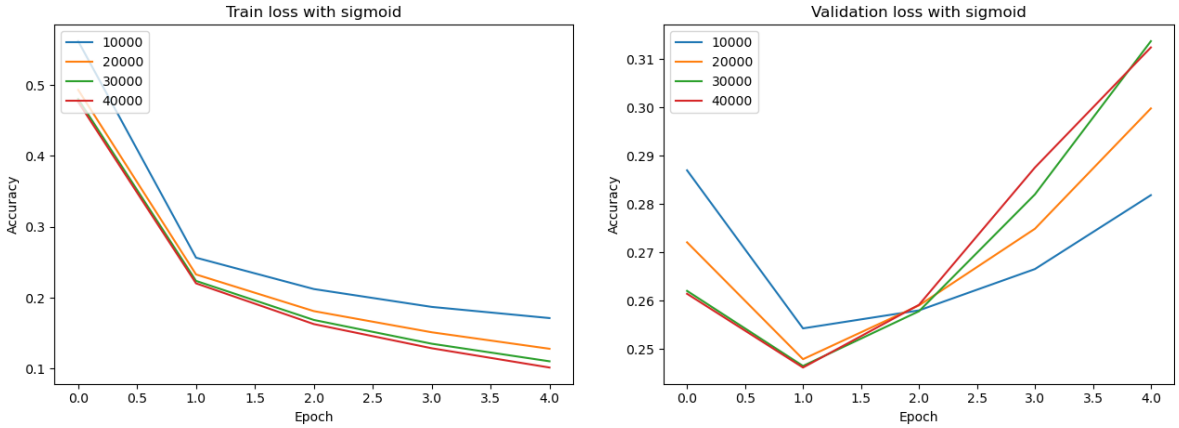


Рисунок 14 — Результаты исследования различных размеров вектора представления текста вблизи размеров 10000, 20000, 30000, 40000. Как видно, наибольшая пиковая точность на тестах принадлежит вектору с размером 40000.

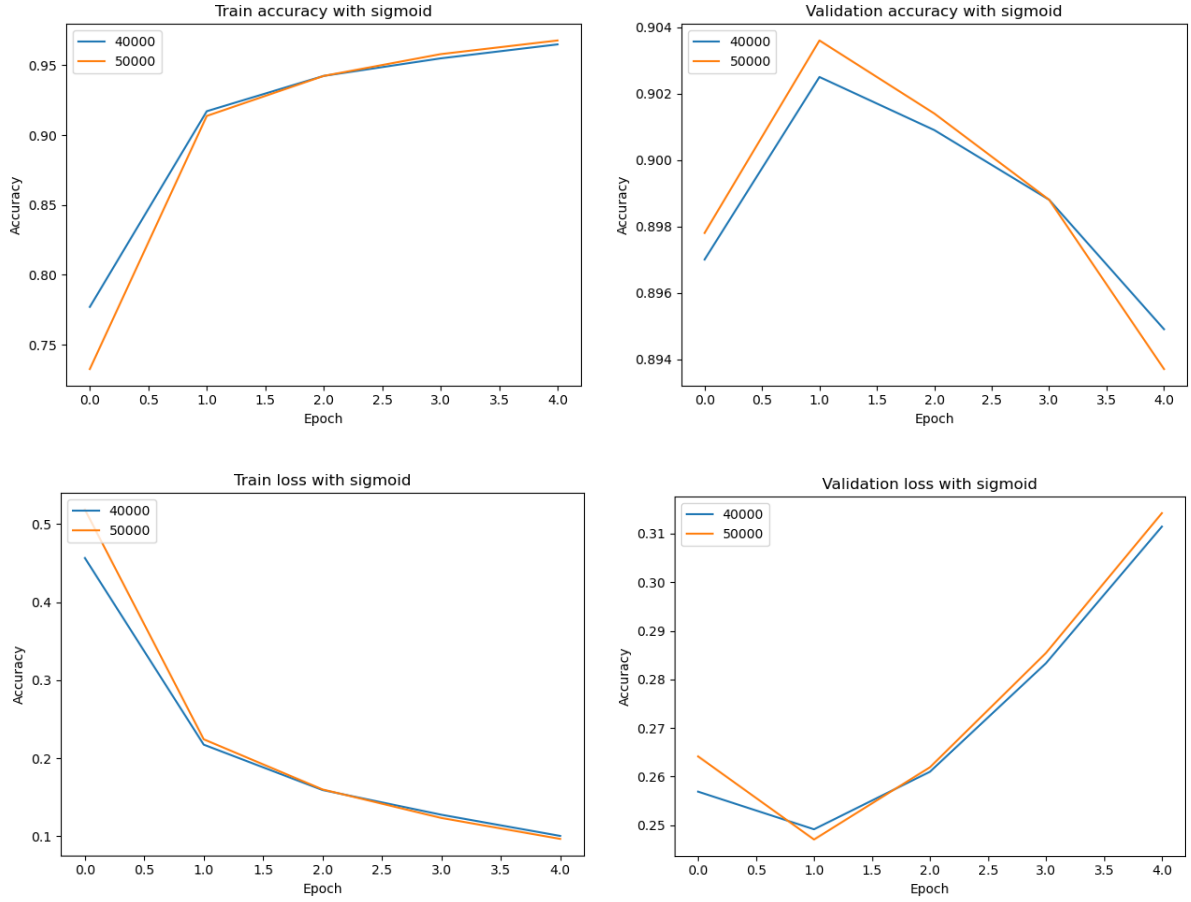


Рисунок 15 — Сравнение результатов для моделей с векторами размером 40000 и 50000.

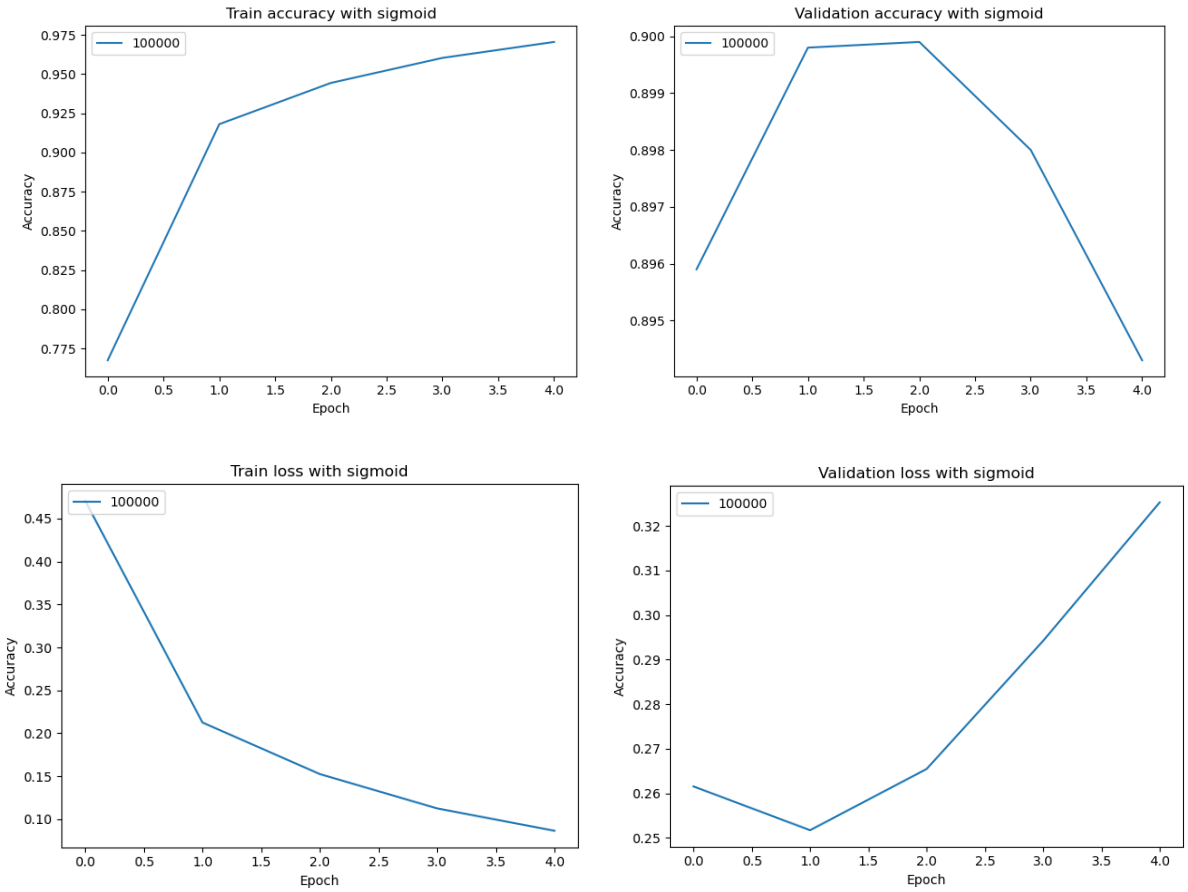
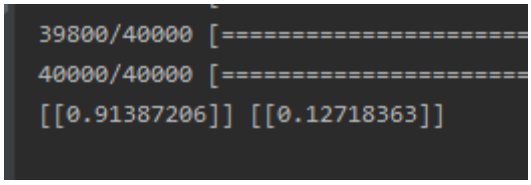


Рисунок 16 — Результаты для вектора размером 100000. Как видно, они не превышают точности на тестировании в 90%, тогда как моделям с вектором размером 40000 и 50000 это удавалось.

Разницу в точности для вектора размером 100 и размером в 10000 можно объяснить тем, что 10000 — это кол-во слов в самом большом обзоре из датасета. Представив данные вектором в 100 раз, меньше мы теряем возможность целостно изучить все ревью размером от 100 до 10000 слов. Поэтому точность сильно ниже. Я не уверена, что увеличение размера вектора на самом деле улучшает результат, учитывая, что все последующие за 10000 размеры вектора очень близки по точности валидации, скорее всего это совпадение.

3. Написать функцию, которая позволяет ввести пользовательский текст (в отчете привести пример работы сети на пользовательском тексте).

Для тестирования собственного примера была написана функция `test_my_text()`. В качестве ревью были выбраны два настоящих ревью новых фильмов с imdb. Первое (в файле test.txt) принадлежит человеку, оценившему фильм на 10 из 10, второе (в файле test2.txt) человеку, оценившему фильм на 1 из 10.



Вывод.

В ходе выполнения данной работы было произведено ознакомление с задачей регрессии, изучены способы представления текста для передачи в ИНС, но не получилось достигнуть точности прогноза выше 95%.

ПРИЛОЖЕНИЕ А
ИСХОДНЫЙ КОД

```
import string

import matplotlib.pyplot as plt
import numpy as np
from keras.utils import to_categorical
from keras import models
#from keras import layers
from tensorflow.keras.models import Sequential
from keras.regularizers import l2, l1
from keras.datasets import imdb
from tensorflow.keras.layers import Embedding, Dropout, Conv1D,
MaxPool1D, GRU, LSTM, Dense, SimpleRNN, Flatten, Activation,
LeakyReLU, PReLU
from keras.layers import Bidirectional, GlobalMaxPool1D
import tensorflow.keras.activations
import efficientnet.tfkeras
from tensorflow.keras.models import load_model
# For adding new activation function
from keras import backend as K
from keras.utils.generic_utils import get_custom_objects


def swish(x):
    return (K.sigmoid(x) * x)

def vectorize(sequences, dimension=10000):
    # Create an all-zero matrix of shape (len(sequences), dimension)
    results = np.zeros((len(sequences), dimension))
    for i, sequence in enumerate(sequences):
        results[i, sequence] = 1 # set specific indices of
results[i] to 1s
    return results

def to_str(array):
    result = []
    for element in array:
        result.append(str(element))
    return result
```

```
def singleModelPlots( histories, dimensions = [10000] ):
    #print(histories.history)
    #plt.plot(histories.history['acc'])
    #plt.show()
    for history in histories:
        plt.plot(history.history['acc'])
    plt.title('Train accuracy with ' + activation )
    plt.ylabel('Accuracy')
    plt.xlabel('Epoch')
    plt.legend(to_str(dimensions), loc='upper left')
    plt.show()

    for history in histories:
        plt.plot(history.history['val_acc'])
    plt.title('Validation accuracy with ' + activation )
    plt.ylabel('Accuracy')
    plt.xlabel('Epoch')
    plt.legend(to_str(dimensions), loc='upper left')
    plt.show()

    for history in histories:
        plt.plot(history.history['loss'])
    plt.title('Train loss with ' + activation )
    plt.ylabel('Accuracy')
    plt.xlabel('Epoch')
    plt.legend(to_str(dimensions), loc='upper left')
    plt.show()

    for history in histories:
        plt.plot(history.history['val_loss'])
    plt.title('Validation loss with ' + activation )
    plt.ylabel('Accuracy')
    plt.xlabel('Epoch')
    plt.legend(to_str(dimensions), loc='upper left')
    plt.show()
    return


def build_model(dimension = 10000):
 #   model = Sequential()
  #  model.add(Embedding(input_dim=10000,output_dim=32,
input_length=10000))
    #model.add(Conv1D(filters=32, kernel_size=3, padding='same',
activation='relu'))
```

```
    #model.add(MaxPool1D(pool_size=2))
    #model.add(LSTM(100))
    #model.add(GRU(32))
    #model.add(GRU(32, return_sequences=True))
    #model.add(SimpleRNN(16))
    #model.add(Dropout(0.2))
    #model.add(LSTM(64, dropout=0.2, recurrent_dropout=0.2))
    max_features = 9999
    embed_size = 256
    model = Sequential()

   # get_custom_objects().update({'swish': Activation(swish)})
    model.add(Dense(64, activation=activation,
input_shape=(dimension,)))
    #model.add(PReLU())
    model.add(Dropout(0.3))
    model.add(Dense(64, activation=activation))
    #model.add(PReLU())
    model.add(Dropout(0.3))
    model.add(Dense(64, activation=activation))
    #model.add(PReLU())
    model.add(Dense(1, activation="sigmoid"))


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

def test_dimensions(dimensions=[10000]):
    results = []
    for dimension in dimensions:
        test_x, test_y, train_x, train_y = load_data(dimension)
        results.append(build_model(dimension).fit(
            train_x, train_y,
            epochs=epochs,
            batch_size=200,
            validation_data=(test_x, test_y)
        ))
    singleModelPlots(results, dimensions)
    return

def load_data(dimension=10000):
```

```
    (training_data, training_targets), (testing_data,
testing_targets) = imdb.load_data(num_words=dimension)
    data = np.concatenate((training_data, testing_data), axis=0)
    targets = np.concatenate((training_targets, testing_targets),
axis=0)

    #print("Categories:", np.unique(targets))
    #print("Number of unique words:",
len(np.unique(np.hstack(data))))

    length = [len(i) for i in data]

    #print("Average Review length:", np.mean(length))
    #print("Standard Deviation:", round(np.std(length)))
    #print("Label:", targets[0])
   # print(data[0])

    index = imdb.get_word_index()
    reverse_index = dict([(value, key) for (key, value) in
index.items()])
    decoded = " ".join([reverse_index.get(i - 3, "#") for i in
data[0]])

   # print(decoded)

    data = vectorize(data, dimension)
    targets = np.array(targets).astype("float32")


    test_x = data[:10000]
    test_y = targets[:10000]
    train_x = data[10000:]
    train_y = targets[10000:]
    return (test_x, test_y, train_x, train_y)

def test_my_text(filename, dimension=10000):

    text = []
    with open(filename, 'r') as f:
        for line in f.readlines():
            text+=line.translate(str.maketrans('', '',
string.punctuation)).lower().split()
    indexes = imdb.get_word_index() # use ready indexes
    print(indexes)
```

```
    print(text)
    encoded = []
    for word in text:
        if word in indexes and indexes[word] < 10000: # <10000 to
avoid out of bounds error
            print('found '+word+' in indexes. its index is '+
str(indexes[word]))
            encoded.append(indexes[word])
    print('----------------------')
    print(np.array(encoded))

    reverse_index = dict([(value, key) for (key, value) in
indexes.items()])

    decoded = " ".join([reverse_index.get(i , "#") for i in
np.array(encoded)]) # не пон почему в ориге i-3
    print(decoded)
    test_x, test_y, train_x, train_y = load_data()

    print(decoded)
    #print(len(text.split()))
    model = build_model()
    model.fit(train_x, train_y, epochs=2, batch_size=200,
validation_data=(test_x, test_y))
     # vectorize just like we did with data
    #print(model.predict(vectorize([np.array(encoded)])))
    return model.predict(vectorize([np.array(encoded)]))

activation = 'relu'
epochs = 2

if __name__ == '__main__':
    #test_dimensions([ 100, 1000, 5000, 10000, 20000, 30000, 40000])
    #test_dimensions() # for default launch on vector 10000
    #test_my_text('test.txt')
    print(str(test_my_text('test.txt'))+'
'+str(test_my_text('test2.txt')))
   # print()
```