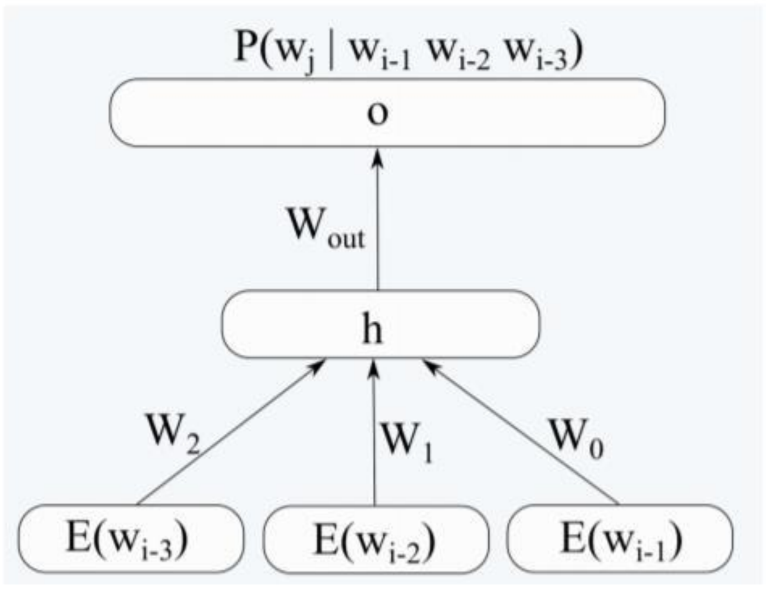
Creating a Language Model based on Feed Forward Neural Network

This project aims to create a language model using the Feed-Forward neural network and then investigate the effect of different parameters on the quality of this language model. Two files train.txt and test.txt are available for model training and evaluation, respectively.



Question 1 – A, B

Preprocessing

In the preprocessing step, the below operations were done:

- Replacing all "\n" with blank.
- Removing HTML tags
- Changing to lowercase text
- Running tokenization by sent_tokenize, word_tokenize of nltk library.
- Removing not English characters

These processing were done by the prepare_text() method in this project source code.

	Train.txt	Test.txt
Number of characters	5،034،054	**126،477**
Number of tokens	839,510	**20,031**
Number of sentences	42,801	**992**

Word Embedding

The file *glove.6B.50d.txt* had been provided for embedding. This file has vectors including 50 numbers for each of 400,000 words. I extracted the word2vec object using *glove2word2vec()* and *load_word2vec_format()* methods of the genism library. These were done by the *build_glove_word2vec()* method in this project's source code.

Creating one-hot vectors

For Identifying unique tokens and creating one-hot vectors I used LabelEncoder and OneHotEncoder classes of sklearn library. (*build_encoder()* in source code)

Based on the extracted tokens of the train.txt file, 57,500 unique tokens were obtained.

Creating Input and Output vectors for FF network

I used the keras library for creating a model and training and evaluating it. I used different n-grams in different evaluations, however, the number of vectors for train and test, were different, so I've reported them in the table below.

n-gram	number of Train vectors	number of Test vectors	Input vector type	Input vector size	Output vector size
5-grams	641293	22,143	embedding	200	57,500
5-grams	641293	22,143	One-hot	230,000	57,500
4-grams	687418	15,751	embedding	150	57,500
3-grams	735535	16,802	embedding	100	57,500

As can be seen, we can't have all the input and output vectors in memory simultaneously. For example, in 5-gram and embedding type, we need around 34GB.

(200 + 57500) * 641292 = 37,002,548,400 ~= 34GB

Therefore instead of using *fit()*, I used *fit_generator()* method of keras, to create input and output vectors at the same type of running.

The below methods were coded:

- ✓ *generate_file_ngram_X_Y*
- ✓ *generate_file_trigram_X_Y*
- ✓ *generator_batch*

Table below shows the filename for different n-grams with their information.

Filename	File size (MB)	Description
train_5gram_nopad_vectors.csv	1200	201 columns-200 columns for embedding and 1 column for indexin vocab as output
train_4gram_nopad_vectors.csv	967	151 columns-150 columns for embedding and 1 column for indexin vocab as output
train_3gram_nopad_vectors.csv	693	101 columns-100 columns for embedding and 1 column for indexin vocab as output
test_5gram_nopad_vectors.csv	41	201 columns-200 columns for embedding and 1 column for indexin vocab as output
test_4gram_nopad_vectors.csv	22	151 columns-150 columns for embedding and 1 column for indexin vocab as output
test_3gram_nopad_vectors.csv	15	101 columns-100 columns for embedding and 1 column for indexin vocab as output

Project source code

Each of the practices for this project was done in a separated notebook their source code are similar, and only the parameters or a few methods were different.

Notebook Name	Description
CA3_FFNN_LM_5gram_35h_lr_0_02_glove.ipynb	Input: 4 embedded words-vector size=200 Output: one-hot vector with a size of 57500 Number of neurons in hidden layer: 35 Learning rate: 0.02
CA3_FFNN_LM_4gram_35h_lr_0_02_glove.ipynb	Input: 3 embedded words-vector size=150 Output: one-hot vector with a size of 57500 Number of neurons in hidden layer: 35 Learning rate: 0.02
CA3_FFNN_LM_3gram_35h_lr_0_02_glove.ipynb	Input: 2 embedded words-vector size=100 Output: one-hot vector with a size of 57500 Number of neurons in hidden layer: 35 Learning rate: 0.02
CA3_FFNN_LM_5gram_35h_lr_0_01_glove.ipynb	Input: 4 embedded words-vector size=200 Output: one-hot vector with a size of 57500 Number of neurons in hidden layer: 35 Learning rate: 0.01
CA3_FFNN_LM_5gram_35h_lr_0_03_glove.ipynb	Input: 4 embedded words-vector size=200 Output: one-hot vector with a size of 57500 Number of neurons in hidden layer: 35 Learning rate: 0.03
CA3_FFNN_LM_5gram_35h_lr_0_1_glove.ipynb	Input: 4 embedded words-vector size=200 Output: one-hot vector with a size of 57500 Number of neurons in hidden layer: 35 Learning rate: 0.1
CA3_FFNN_LM_5gram_50h_lr_0_02_glove.ipynb	Input: 4 embedded words-vector size=200

	Output: one-hot vector with a size of 57500 Number of neurons in hidden layer: 50 Learning rate: 0.02
CA3_FFNN_LM_5gram_100h_lr_0_02_glove.ipynb	Input: 4 embedded words-vector size=200 Output: one-hot vector with a size of 57500 Number of neurons in hidden layer: 100 Learning rate: 0.02
CA3_FFNN_LM_5gram_150h_lr_0_02_glove.ipynb	Input: 4 embedded words-vector size=200 Output: one-hot vector with a size of 57500 Number of neurons in hidden layer: 150 Learning rate: 0.02
CA3_FFNN_LM_5gram_35h_lr_0_2_onehot.ipynb	Input: 4 one-hot words-vector size=230000 Embedding layer: size of 200 Output: one-hot vector with a size of 57500 Number of neurons in hidden layer: 35 Learning rate: 0. 2

Parameters Initialization

```
from tensorflow.keras.optimizers import SGD
from tensorflow.keras.layers import Dense
from tensorflow.keras.models import Sequential

def build_model(input_dim, vocab_size, hidden_units, learning_rate):
   model = Sequential()
   model.add(Dense(hidden_units, input_dim=input_dim, activation='sigmoid'))
   model.add(Dense(vocab_size, activation='softmax'))
   sgd = SGD(lr=learning_rate, momentum=0., decay=0., nesterov=False)
   model.compile(loss='categorical_crossentropy', optimizer=sgd, metrics=[perplexity])
   return model
```

Updating Parameters

To train the network and update the parameters in each iteration, the *cross_entropy* cost function and the *Stochastic_gradient_descent* algorithm have been used. As in this issue, we have a multi-class mode, so the cost function is set as *loss='categorical_crossentropy'*, whose formula is as follows:

$$L_{CE}(\hat{y}, y) = -\sum_{i=1}^{C} y_i \log \hat{y}_i$$

To train the parameters, after calculating the cost function, the weights are modified in the reverse direction of the gradient in order to reduce the cost value based on *gradient_descent* and *back_propagation* algorithms .

$$\nabla_\theta L(f(x;\theta),y)) = \begin{bmatrix} \frac{\partial}{\partial\theta_1} L(f(x;\theta),y) \\ \frac{\partial}{\partial\theta_2} L(f(x;\theta),y) \\ \vdots \\ \frac{\partial}{\partial\theta_m} L(f(x;\theta),y) \end{bmatrix}$$

$$\theta_{t+1} = \theta_t - \eta \nabla L(f(x;\theta),y)$$

The learning rate in the above formula has been determined manually in different experiments. (0.01 or 0.02 or 0.03 or 0.1 or 0.2) in the above SGD code, was adjusted as follows:

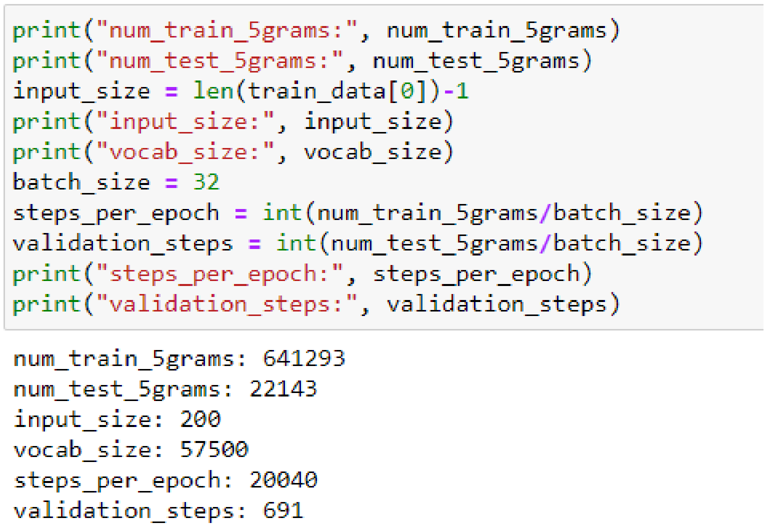
```
SGD(lr=learning_rate, momentum=0., decay=0., nesterov=False)
```

Momentum and Nesterov parameters can be used to determine the influence of previous gradients in the calculation of new weights, where zero and False means that only the current gradient is effective.

Stop training condition

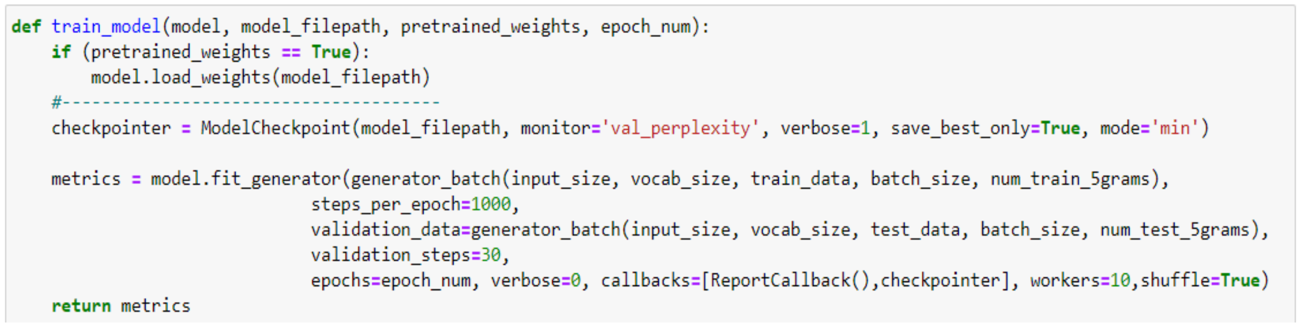
Due to the fact that the training of the network in various experiments was very time-consuming, therefore, only the training of the models was continued up to a limited number of 25 epochs, and therefore, good results were not achieved.

In this model, according to the obtained parameters shown below; In fact, for all training n-grams to be used for model training and all test data n-grams to evaluate the model, it is necessary that the parameters step_per_epoch=20040 and validation_steps=691 were used in the fit_generator call, but then the training would be very slow. Therefore, due to lack of time, these parameters were changed to 1000 and 30, respectively, and the same values were used in all experiments.



```python
print("num_train_5grams:", num_train_5grams)
print("num_test_5grams:", num_test_5grams)
input_size = len(train_data[0])-1
print("input_size:", input_size)
print("vocab_size:", vocab_size)
batch_size = 32
steps_per_epoch = int(num_train_5grams/batch_size)
validation_steps = int(num_test_5grams/batch_size)
print("steps_per_epoch:", steps_per_epoch)
print("validation_steps:", validation_steps)
```

```
num_train_5grams: 641293
num_test_5grams: 22143
input_size: 200
vocab_size: 57500
steps_per_epoch: 20040
validation_steps: 691
```



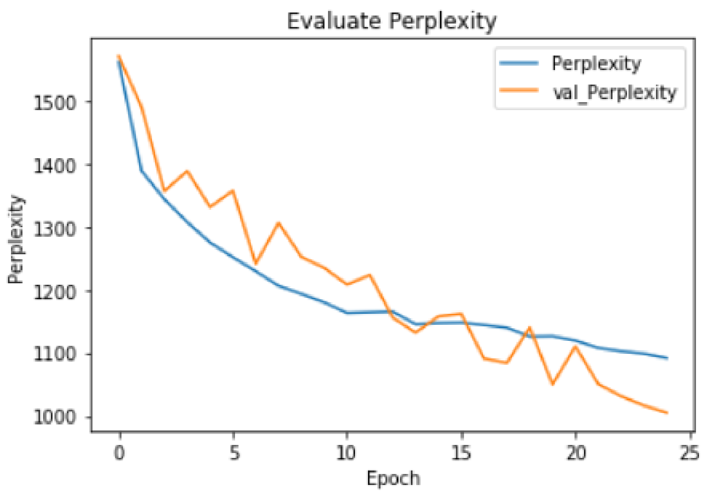
```python
def train_model(model, model_filepath, pretrained_weights, epoch_num):
    if (pretrained_weights == True):
        model.load_weights(model_filepath)
    #-------------------------------------
    checkpointer = ModelCheckpoint(model_filepath, monitor='val_perplexity', verbose=1, save_best_only=True, mode='min')

    metrics = model.fit_generator(generator_batch(input_size, vocab_size, train_data, batch_size, num_train_5grams),
                                  steps_per_epoch=1000,
                                  validation_data=generator_batch(input_size, vocab_size, test_data, batch_size, num_test_5grams),
                                  validation_steps=30,
                                  epochs=epoch_num, verbose=0, callbacks=[ReportCallback(),checkpointer], workers=10,shuffle=True)
    return metrics
```

Question1 - C

The following method was used to calculate perplexity, which was added to the model as a metric and in the form of a callback, and its results were recorded.

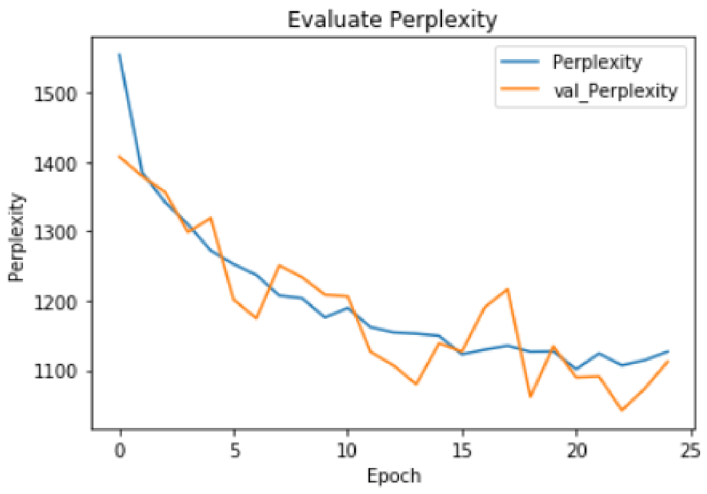
```
def perplexity(y_true, y_pred):
  cross_entropy = losses.categorical_crossentropy(y_true, y_pred)
  perplexity = backend.pow(2.0, cross_entropy)
  return perplexity
```

Based on what was said in the previous step, in fact, a part of training n-grams and test n-grams will be used randomly in each epoch, which has a negative effect on perplexity and is also clear in the figure below. The number of test n-grams that are not seen in the training is large and therefore fluctuates a lot.

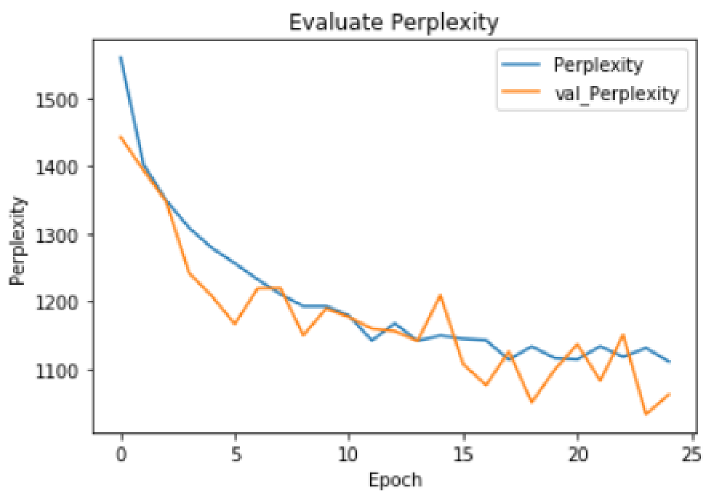


Question1 - D

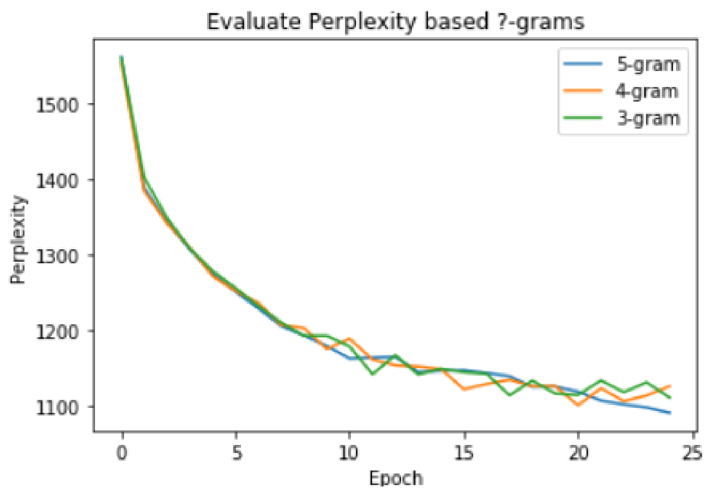
In the case that the 150 vector input consists of embedding three words, the learning rate is 0.02 and the number of neurons in the hidden layer is equal to 35. The output is as follows :

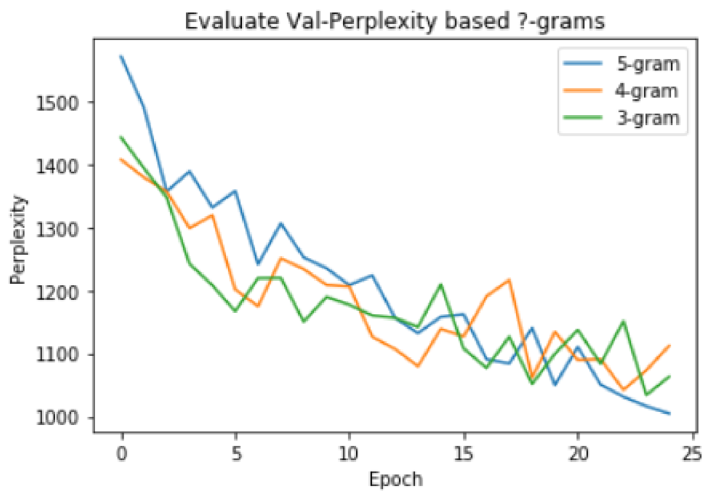


In the case that the input is a 100 vector consisting of embedding two words, the learning rate is 0.02 and the number of neurons in the hidden layer is 35. The output is as follows:



In the following, for all three modes 5-gram, 4-gram, and 3-gram perplexity diagrams for train and test are drawn separately in one diagram so that comparison is possible.

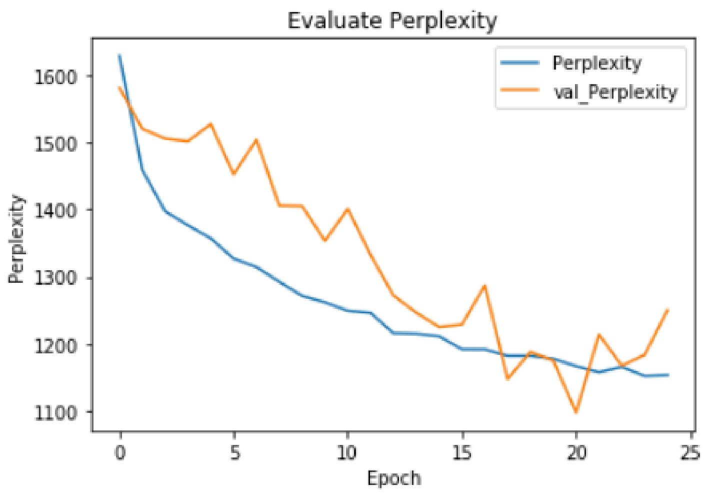




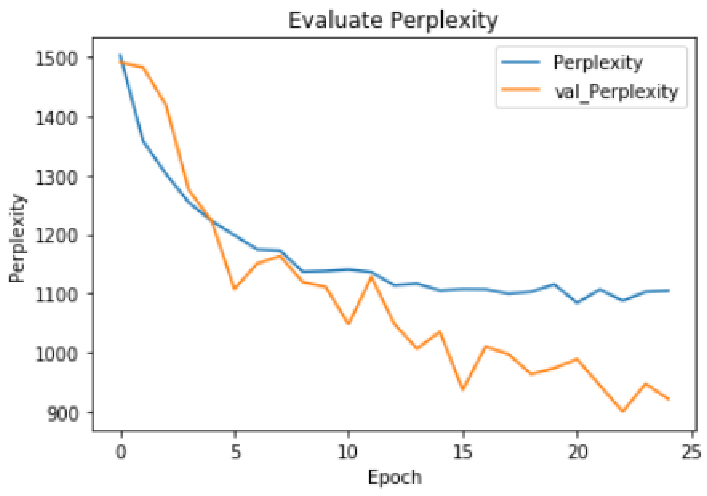
From the comparison of these graphs, it can be said that the effect of the n-gram type on the quality of the network in this example is not very obvious. Although the number of training data is more in 3-gram mode, but for this project, not all training and test data have been used and in terms of the number of data used, all modes are the same, so more data could not help.

Question1 - E

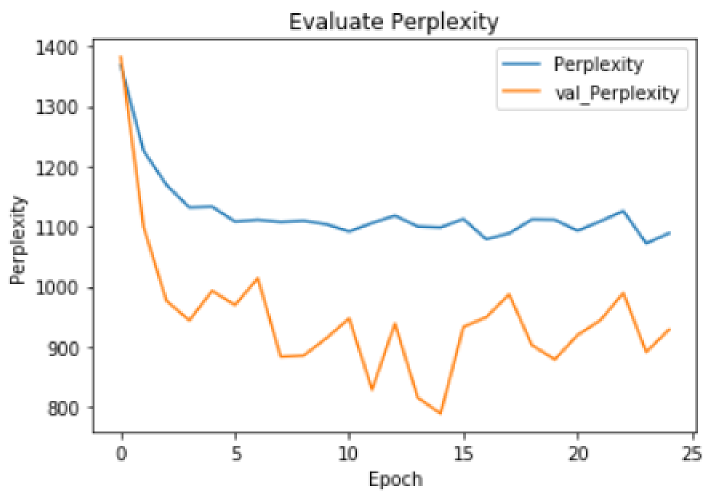
If the input is a 200 vector consisting of embedding four words, the learning rate is 0.01 and the number of neurons in the hidden layer is 35. The output is as follows:



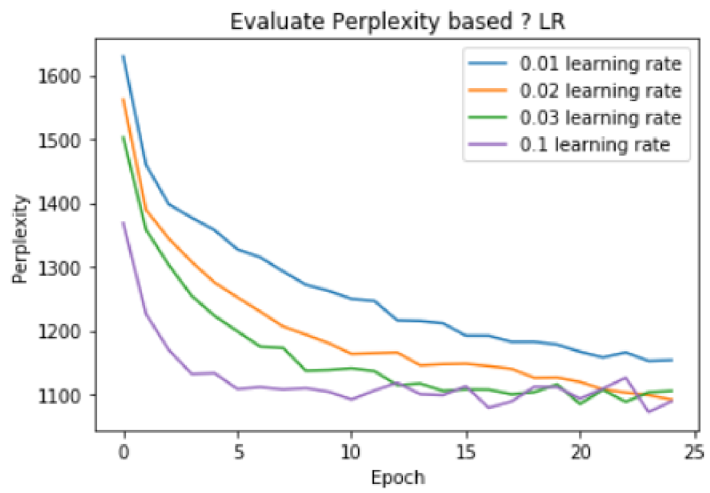
If the input is a 200 vector consisting of embedding four words, the learning rate is 0.03 and the number of neurons in the hidden layer is 35. The output is as follows:

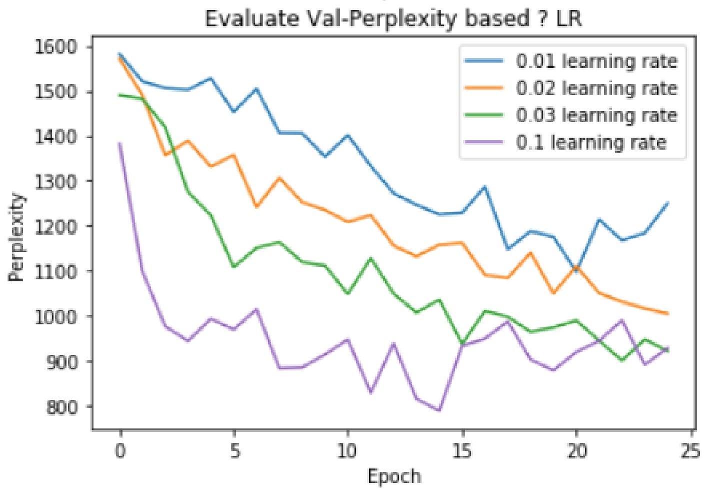


In the case that the input is a 200 vector consisting of embedding four words, the learning rate is 0.1 and the number of neurons in the hidden layer is 35. The output is as follows:



In the following, for all four modes used to adjust the learning rate, the perplexity graphs for train and test are drawn separately in one graph, so that comparison is possible.

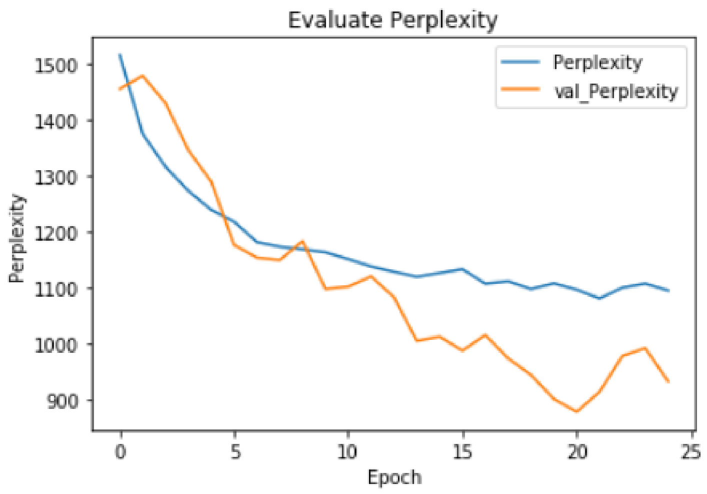




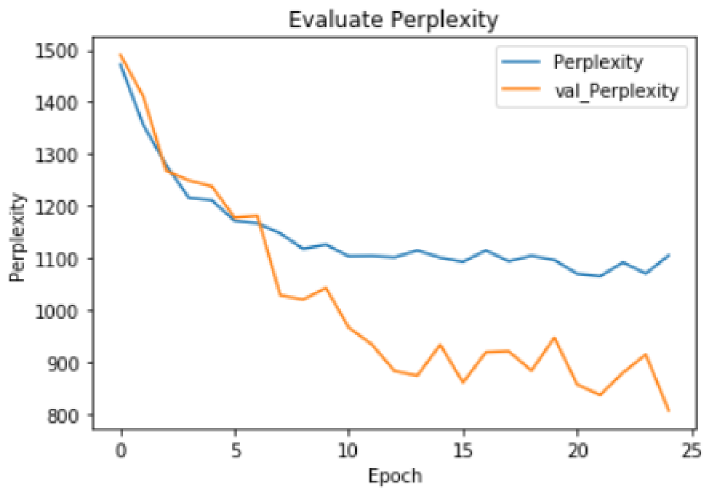
It seems that the learning rate of 0.1 has reached a stable state very soon and has better accuracy than the other modes, and in total at each stage; Increasing the learning rate has improved performance.

Question1 - F

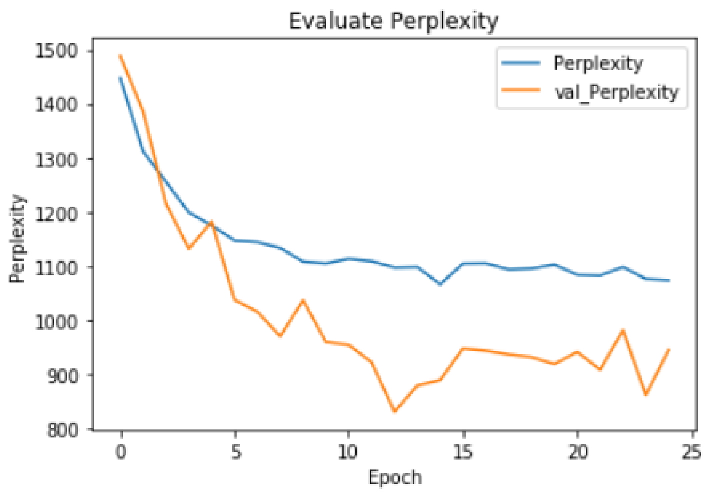
In the case that the input is a 200 vector consisting of embedding four words, the learning rate is 0.02 and the number of neurons in the hidden layer is 50. The output is as follows:



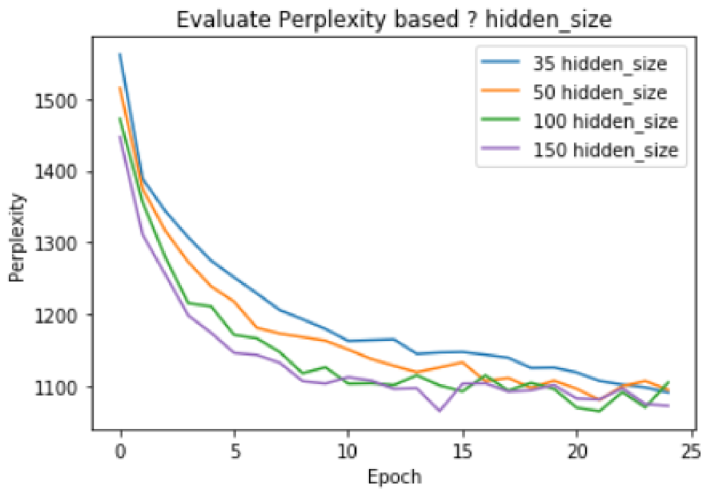
In the case that the input is a 200 vector consisting of four word embeddings, the learning rate is 0.02 and the number of neurons in the hidden layer is 100. The output is as follows:

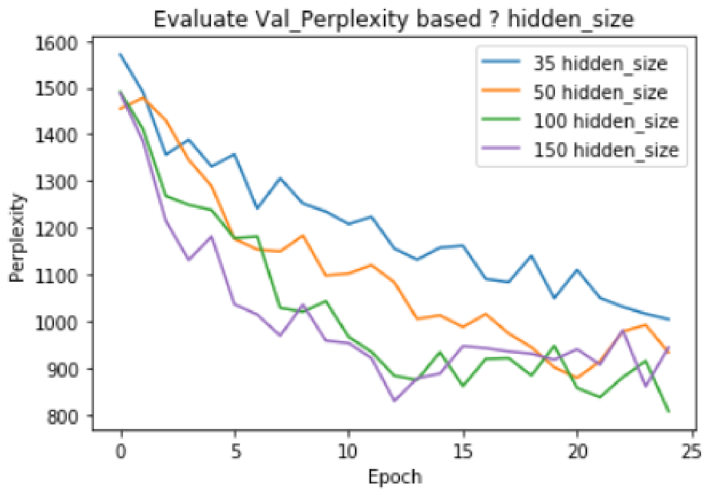


In the case that the input is a 200 vector consisting of four word embeddings, the learning rate is 0.02 and the number of neurons in the hidden layer is 150. The output is as follows:



In the following, for all four modes used to adjust the hidden size, the perplexity diagrams for train and test are drawn separately in one diagram, so that comparison is possible.





The graphs show that the best performance is related to the case where 150 neurons are used in the hidden layer, and in fact, as the number of neurons in the hidden layer increases, the efficiency of the model increases because the network has more parameters and could adjust them more.

Question 2 - A

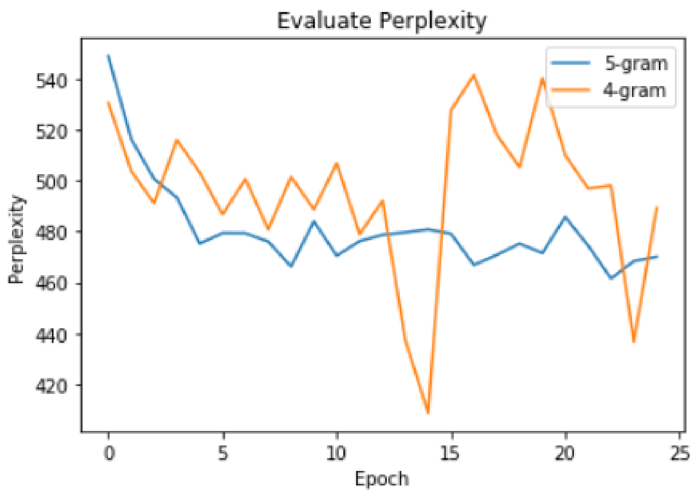
To do this part, instead of using glove-ready embedding vectors, a layer is added to the network so that the network itself learns the embedding vectors. Therefore, the input of the network is the one-hot vectors corresponding to 4 words, which will be 230000 long, and it is supposed that an embedding of length 50 is assumed for each word, that means, for the second layer, we have 200 neurons and then 35 neurons as the hidden layer and next dense layer. Its method is as follows :

```
from tensorflow.keras.optimizers import SGD

def build_model(input_dim, vocab_size, embedding_size, hidden_units, learning_rate):
    model = Sequential()
    model.add(Dense(embedding_size*4, input_dim=input_dim, activation='tanh'))
    model.add(Dense(hidden_units, input_dim=input_dim, activation='sigmoid'))
    model.add(Dense(vocab_size, activation='softmax'))
    sgd = SGD(lr=learning_rate, momentum=0., decay=0., nesterov=False)
    model.compile(loss='categorical_crossentropy', optimizer=sgd, metrics=[perplexity])
    return model
```

The tanh function is used as the activation function of the embedding layer to provide the possibility of embedding words in the range between 1 and -1. The generate_index_ngram_X_Y and generator_onehot methods are written in the corresponding notebook for the possibility of feeding one-hot vectors as the network input.

Question 2 - B

It was practically not possible to execute this section for the entire train data and despite spending a lot of time, no results were obtained. The code of this part is located in the notebook with the name CA3_FFNN_LM_5gram_35h_lr_0_2_onehot.ipynb, and therefore it was done for a small part (one-sixth of the train data) whose perplexity diagram is as follows. And of course, due to different conditions, it is not possible to compare it with part c of question 1.


Evaluate Perplexity
5-gram
4-gram
Perplexity
Epoch
540
520
500
480
460
440
420
0
5
10
15
20
25