Comparing simple LSTM and classical NN, Report

Homework 1: Classification Task

Contact info

GitHub link: https://github.com/kirilldaniakin/AML-DS-2021

Kirill Daniakin: k.daniakin@innopolis.university

Motivation

Motivation behind this project is to compare relatively simple neural network architectures (baseline LSTM model, custom LSTM model, and classical NN model) on the example of gender classification by name task.

Task definition and data description

Dataset containing names of people and corresponding gender (Male, Female) is given. The dataset is presented as two .csv files (one for train, one for test) with two columns each: “Name” and “Gender”. The task is to create an artificial neural network (ANN) able to get names as an input, and give corresponding to these names genders as an output. The task is binary classification.

Data Preprocessing

The dataset is a collection of strings of variable length. The labels for the training samples are also strings. This format is not very friendly for learning algorithms. The simplest way to convert the string representation into the machine-readable format is to substitute the characters with a unique integer identifier. This can be achieved by creating the character vocabulary.

In order to create vocabulary (and ANNs themselves) tensorflow API was used. Namely, the layer `tensorflow.keras.layers.experimental.preprocessing.TextVectorization`. This layer has method “adapt”, which fits it to data and creates vocabulary.

The desire at this step is to prepare data for feeding to an ANN. That is why integer representation is used. Assuming that the vocabulary consists only of alphabetical letters (upper and lower case), every character in a name should be replaced with its number in vocabulary. For

example, for the name Elizabeth, integer representation with described dictionary, where first comes upper case, will look like this:

```
[ 5 38 35 52 27 28 31 46 34].
```

The goal here is to encode every character, so data should be preprocessed first. The preprocessing step is to separate characters in names by adding whitespace in between them, so for example, the name “Terrone” will become “T e r r o n e”. This way, TextVectorization layer will adapt to data and produce vocabulary containing only 52 alphabet letters (lowercase 26 + capital 26) and two special characters for empty and unknown words. TextVectorization separates characters by whitespace, so no whitespace will be in vocabulary.

By exploring the dataset, it can be seen that the longest name in train data contains 15 characters. The example of name of this length from train data is “Christiandaniel”. It is better to feed an ANN fixed length sequence even if this is LSTM (for computational purposes). That is why the padding to fixed length is used: if a name is shorter than 15 characters, integer representation will be padded with zeros to fill in missing characters.

After applying described preprocessing to the name Elizabeth, the following integer representation is achieved:

```
[ 5 38 35 52 27 28 31 46 34  0  0  0  0  0  0].
```

Gender labels ‘F’ and ‘M’ are transformed to 0 and 1 correspondingly.

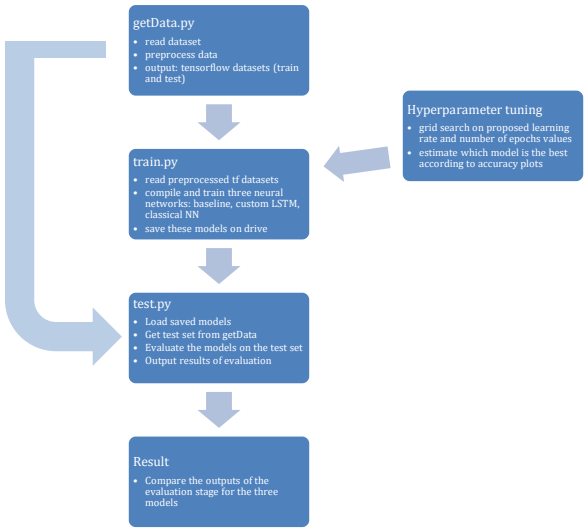
Justification for design decisions

Data preprocessing mechanism described above was chosen because tensorflow keras expects to get tensor or tensorflow dataset as an input, but initial data comes as .csv file and is read by pandas, resulting in a DataFrame. Therefore, to avoid any complicated manipulations with data types, TextVectorization layer was used with the preprocessing step described above.

For the sake of simplicity and time efficiency, complicated models and preprocessing steps (such as maxpool, weighted average for flattening the output of the embedding layer) were left out of the scope of this project.

This project workflow

First, the dataset is loaded and preprocessed by getData.py script. After that, the resulting two tensorflow datasets are fed to declared and compiled neural networks inside train.py script, which train on the train data. Also, after each epoch, performance is measured on the test set. Then, the three tuned models are saved, after which the test.py scripts run, loading the models and evaluating them on the test data. The workflow is depicted lower (hyperparameter tuning was done outside of the repository):



Comparison of LSTM model with different hyper-parameters

Baseline LSTM architecture

1. This model is built as a `tf.keras.Sequential`.

2. The first layer is the `encoder`, which converts the text to a sequence of token indices (integer representation).

3. After the encoder is an embedding layer. An embedding layer stores one vector per character. When called, it converts the sequences of character indices to sequences of vectors. These vectors are trainable.

4. A long short-term memory neural network (LSTM) processes sequence input by iterating through the elements. LSTMs pass the outputs from one timestep to their input on the next timestep.

5. After the LSTM has converted the sequence to a single vector the `layers.Dense` does some final processing, and converts from this vector representation to a single logit as the classification output.

Table 1 Baseline model summary

Layer (type)	Output Shape	Param #
text_vectorization (TextVect	(None, 15)	0
embedding (Embedding)	(None, 15, 5)	270
lstm (LSTM)	(None, 5)	220
dense (Dense)	(None, 1)	6

Total params: 496
Trainable params: 496
Non-trainable params: 0

For the sake of simplicity the following hyperparameters were tuned: learning rate (1e-3, 1e-4, 1e-5) and number of epochs (5, 10, 20).

Baseline LSTM was taken with learning rate=1e-3, number of epochs=100.

According to the plot (Fig. 1) of baseline model accuracy on train and test data, baseline overfits train data, therefore the performance on the test set is getting lower with increasing number of epochs.

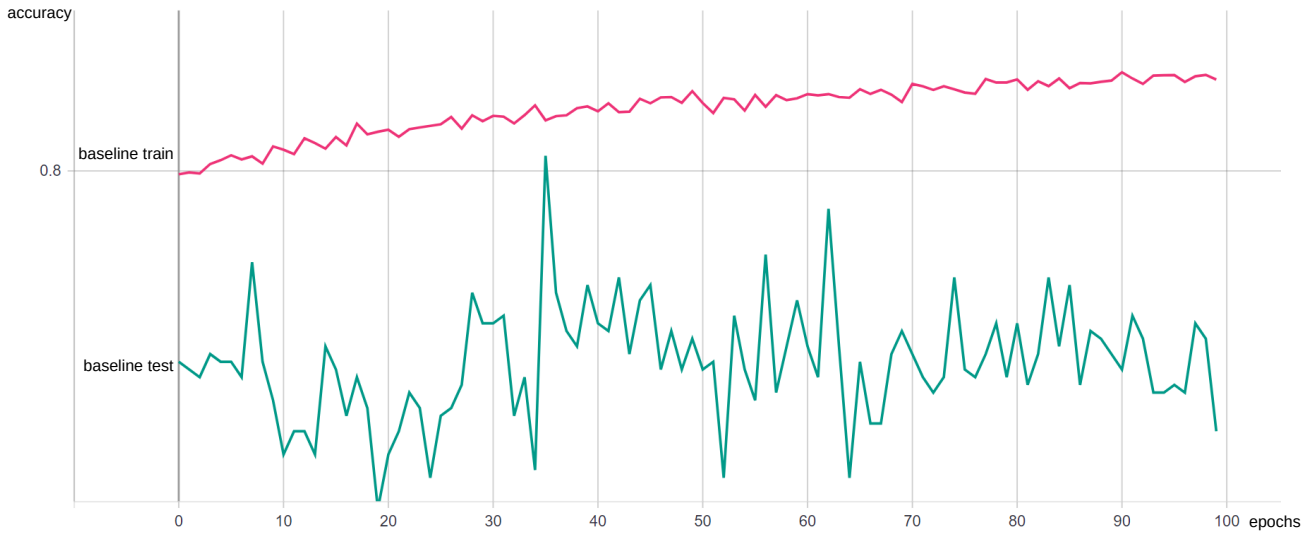


Fig. 1 Baseline model accuracy on train and test sets

In terms of test accuracy the best model on the test set is the one with learning rate=1e-3, and number of epochs=20 (green line in Fig. 2). According to the plot of this model's accuracy on

train and test sets (Fig. 3), it already has started to slightly overfit the training data, as test accuracy went slightly down (from epoch 8, for example).

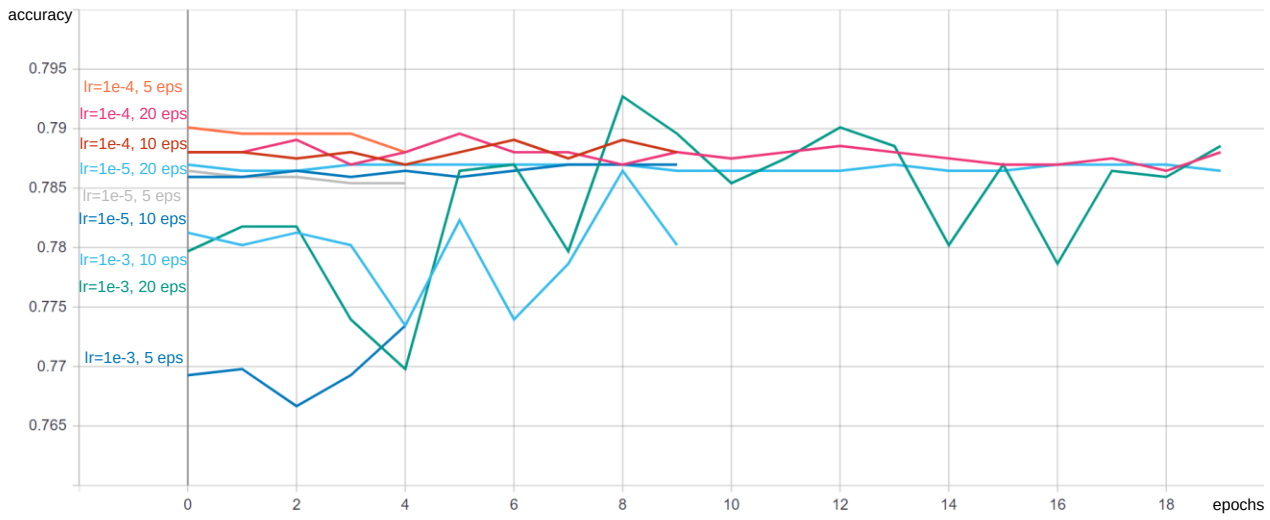


Fig. 2 Hyperparameter tuning results on test, lr=[1e-3,1e-4,1e-5], number of epochs=[5,10,20] (smoothing applied)

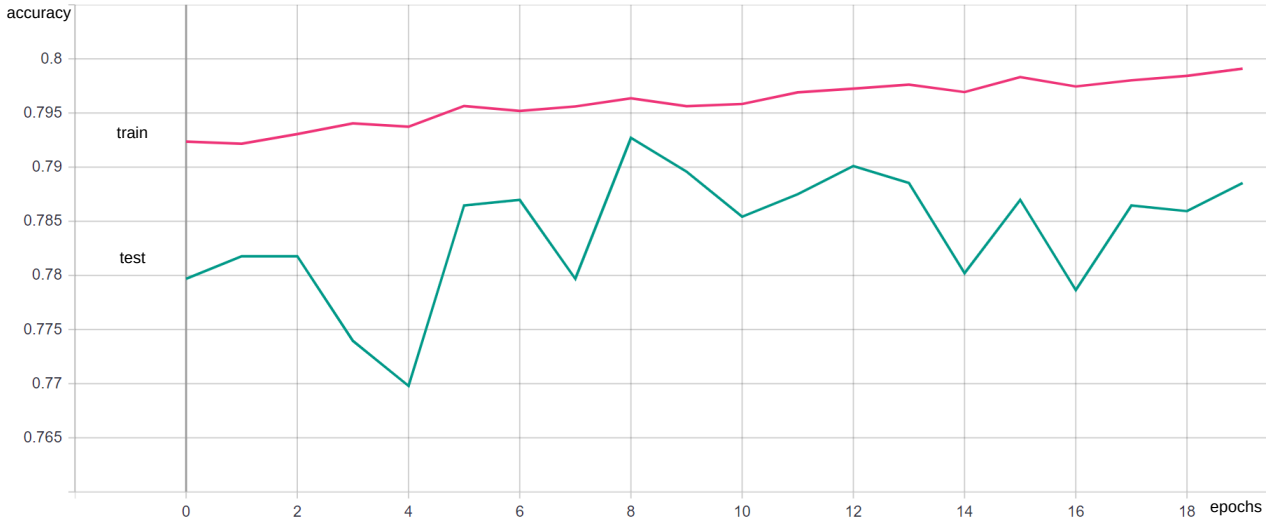


Fig. 3 Tuned custom LSTM train and test accuracy

As can be seen from the plot comparing baseline and the tuned LSTM model (Fig. 4), tuned model at the end of the training process shows slightly higher accuracy on test data then baseline at the end, and lower accuracy on train than baseline, which suggest that baseline overfits.

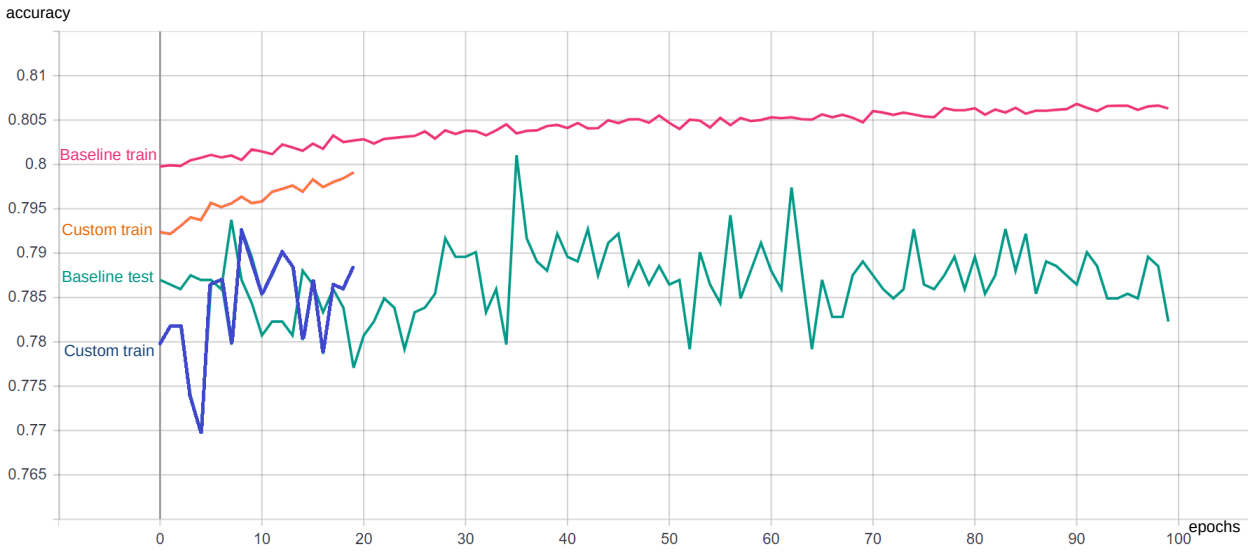


Fig. 4 Comparing custom LSTM and baseline

Comparison of fully-connected models with different hyper-parameters

Classical (fully-connected) ANN architecture

1. This model is built as a `tf.keras.Sequential`.

2. The first layer is the `encoder`, which converts the text to a sequence of token indices.

3. After the encoder is an embedding layer. An embedding layer stores one vector per character. When called, it converts the sequences of character indices to sequences of vectors. These vectors are trainable.

4. Embedding layer's output is flattened and then connected to a chain of Dense layers. The last one in this chain outputs one single value, activation function is sigmoid. If this value is greater than 0.5 model considers input name as male's name, otherwise - female's.

Table 2 Classical NN model summary

Layer (type)	Output Shape	Param #
text_vectorization (TextVect	(None, 15)	0
embedding_1 (Embedding)	(None, 15, 5)	270
flatten (Flatten)	(None, 75)	0
dense_1 (Dense)	(None, 64)	4864
dense_2 (Dense)	(None, 32)	2080
dense_3 (Dense)	(None, 8)	264
dense_4 (Dense)	(None, 1)	9

Total params: 7,487
Trainable params: 7,487
Non-trainable params: 0

Again, for the sake of simplicity the following hyperparameters were tuned: learning rate (1e-3, 1e-4, 1e-5) and number of epochs (5, 10, 20).

The best on the train set is the one with learning rate=1e-3 and number of epochs=20 (magenta line on Fig. 5). Model does not seem to overfit much (Fig. 6).

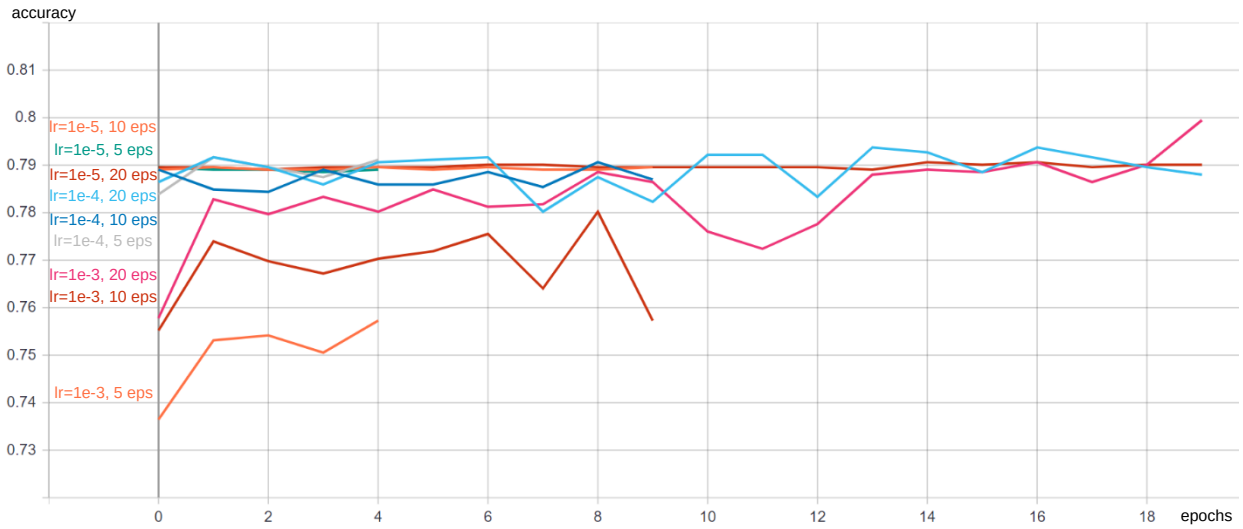


Fig. 5 Classical NN model hyperparameter tuning lr=[1e-3,1e-4,1e-5], number of epochs=[5,10,20]

accuracy

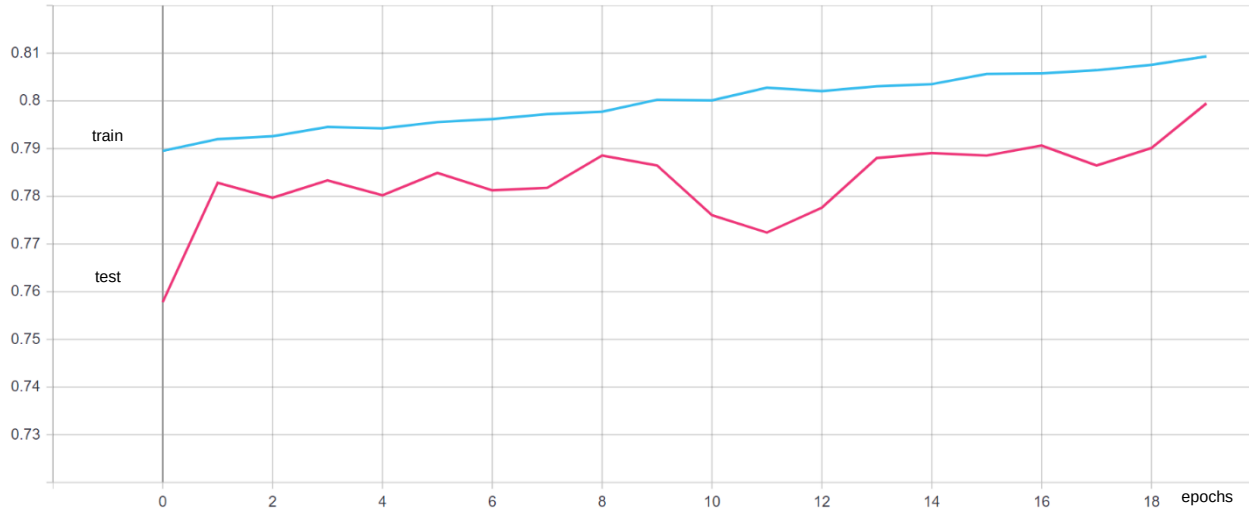


Fig. 6 Tuned classical NN train and test accuracy

This model performs better than baseline, its accuracy is better both on train and test set (Fig. 7).

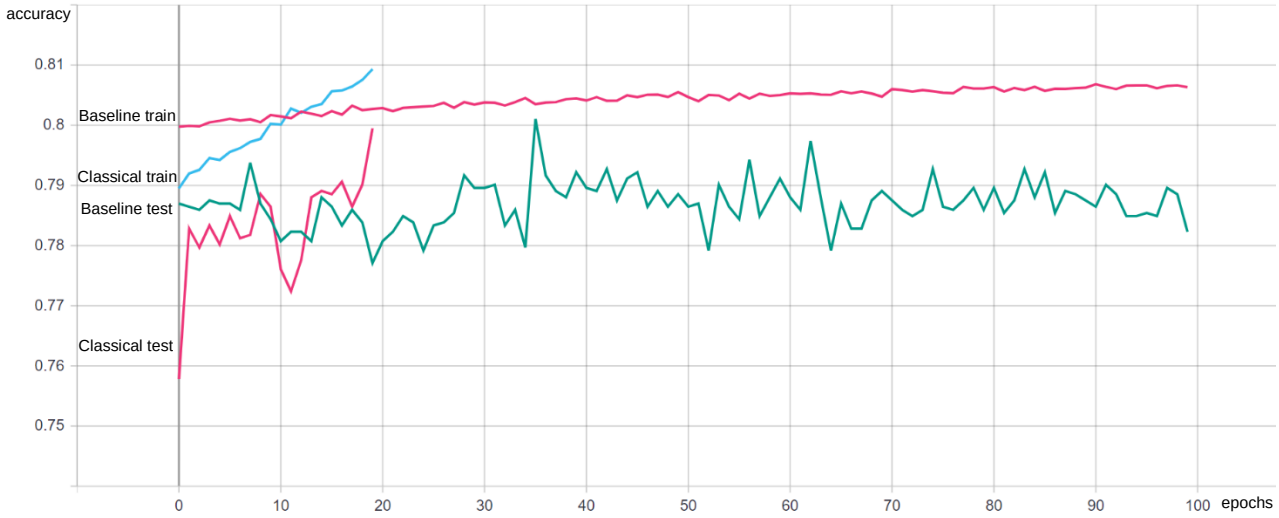


Fig. 7 Comparison of baseline LSTM and classical NN on train and test sets

Resulting model comparison

After hyperparameter tuning and model training, resulting tuned models and baseline were saved with help of the tensorflow API. This is where the training process stops, and testing starts. First, saved models are read from drive by test script, then they are evaluated on the test dataset. Metric scores gathered during evaluation stage are reflected in Table 3.

Table 3 Resulting model comparison

	Baseline LSTM	Custom LSTM	Classical NN
Accuracy	0.7823	0.7885	0.7995
Loss	0.4204	0.4202	0.4338
F1-score	0.4924	0.4926	0.5030

Conclusion and future work

In this project several neural network models were compared on the task of gender classification by name. Hyperparameter tuning for custom and baseline LSTM didn't provide much improvement in model's accuracy, which is to be expected as they share the same architecture. From the other hand, classical (fully connected) neural network with flattened embedding layer output showed better results taking less epochs in training than baseline and the same learning rate. But are this results significantly better? It is possible that this "better" result of classical NN comes from random weight initialization, where classical NN just got "lucky". Besides, classical NN has more parameters than baseline, so some would say that this comparison can't be counted as completely "fair". For future work it is worth it to consider more complicated models with more hyperparameters being tuned as well as alternative ways of flattening embedding layer output (for example, maxpool).

About repository

There are separate files for training and testing the models in the GitHub repository. But the test set is used in both, as in train.py it is desirable to have tensorboard (which was used for getting Fig. 1-7) logs for the test data after each epoch as well.

Script getData.py, which is located in scripts folder, is used for data loading and preprocessing. It returns two tensorflow datasets (train and test) as well as maximum name length found in the train set.

Training script train.py saved the three tuned models: baseline, custom LSTM and classical NN.

Testing script test.py loads these models and evaluates them on the test data.

In order to run train and test procedures in repository, one would need to go to "actions" tab of GitHub from the repository page and run "NN Model Test" workflow, which will install all dependencies, run train.py script, which internally uses getData.py script, then, after training, run test.py script, outputting the accuracy, loss and F1-score on the test set of the 3 models: baseline LSTM, custom LSTM and classical NN.

There is also a Jupyter notebook, which contains preprocessing, train and test all together. In case Git actions won't work, one can easily open this notebook (hw1.ipynb.txt in notebooks folder contains the link to Google Colab), upload the dataset (4 .csv files) there under the folder

named ’data’, and easily check that everything works well. Also, this way one can make use of tensorboard right inside the notebook. Tensorboard logs accuracy, loss, F-score and model weights (via property `write_images=True`).

It is also worth mentioning that no hyperparameter tuning takes place in the GitHub repository scripts. If it is desirable to go deeper in the data preprocessing and encoding examples as well as hyperparameter tuning, one should use abovementioned Jupyter notebook.