CS5011 - Practical 4

[190026921]

May 10, 2022

Word Count: 2163

Contents

1 Introduction and Overview

The objective of this practical was to implement different artificial neural networks for a classification task on the TREC dataset.

1.1 Implementation status

All requirements were implemented, including an extension.

Part	Attempted	Status	Details
Part 1	yes	fully working	-
Part 2	yes	fully working	-
Part 3	yes	fully working	-
Part 4	yes	fully working	word2vec embedding

1.2 Compiling and running instructions

1.2.1 Basic usage

The code can be compiled and run as per the shell scripts `build.sh` and `run.sh`. The running instructions for parts 1 through 3 were left unchanged, and I added an example for part 4 to `run.sh`.

```
java A3Main <part1/part2/part3/part4> <seed> <trainFile> <devFile>
                <testFile> <vocabFile> <classesFile>
                [<trackingFile (optional)>]
```

Part 4 requires different input files. The command to run part 4 with the provided word2vec files is the following (as given in `run.sh`):

```
java -cp lib/*:minet/:src:. src/A4Main part4 123
    data/part4/output/train.txt
    data/part4/output/dev.txt
    data/part4/output/test.txt
    data/part4/output/subset_model.txt
    data/part4/input/classes.txt
```

1.2.2 Instructions for extension, testing, and evaluation

To run the evaluation scripts and JUnit tests, please use the commands provided in the Makefile. This ensures that the dependencies are included in the class path.

All Make commands must be run from the project root and `make compile` must be run first to compile the project.

The following commands are available:

```
compile      Compile all java files
clean        Remove java class files
test         Run JUnit tests
evaluation   Run the evaluation script
tuning       Run the hyperparameter tuning script
timing       Run the timing script
help         Print available commands
```

1.2.3 Python scripts

In Part 4, I use a python script to transform the word2vec file and create the appropriate input files for the models. This script requires the Python library `gensim` [Řehůřek and Sojka, 2010]. The corresponding dependency is given in `word2vec-requirements.txt` and must be installed before running the script.

The plots for the evaluation are also created with a python script. Since some of the dependencies of gensim are incompatible with those required for the plots, I included them in a separate file `evaluation-requirements.txt`. They must be installed before running the script.

2 Design and Implementation

2.1 Hyperparameters

In all parts of the practical, I make use of the `HyperParameter` class, which encodes a specific hyperparameter setting. The provided base setting from the specification is created as an enum `HyperParamsConfig.PART1`.

2.2 Part 1

In Part 1, a classifier class for the TREC data set is implemented together with a corresponding class to represent the data.

The data set is implemented in the `TrecDataset` class, which extends the `Dataset` class - in particular with the input parameter set to `BagOfWords`. The data set class implements the required `fromFile` method.

The `BagOfWords` class is a wrapper around an integer array (with values 0 and 1), which represents a bag of words. The length of the array corresponds to the size of the vocabulary and a 1 at index i signifies that the i-th word of the vocabulary is in the phrase represented by the bag (0 if the word is not in the phrase).

The tree classifier is implemented very similarly to the provided MNIST example. However, for design reasons, the steps of the classification were separated into individual methods:

1. `tc = TrecClassifier`: Instantiate the classifier
2. `tc.load()`: Load the data set
3. `net = tc.getNetwork()`: Get the network preset for the current part
4. `tc.createNetwork(net)`: Create the network
5. `tc.train()`: Train the model
6. `tc.evaluate()`: Evaluate the model

In the `TrecClassifier` class, note the `getNetworkP1` method, which returns the network structure required by Part 1. In particular, the network includes a linear input layer, three subsequent ReLu layers, and a final Softmax output layer (since this is a classification task).

Since none of the logic of the classifier was fundamentally changed compared to the MNIST example, I will not go into further detail regarding the implementation.

2.3 Part 2

Part 2 was implemented in the `EmbeddingBag` class, which replaces the linear input layer from part 1.

The implementation of the `forward` and `backward` methods is key here.

During the `forward` procedure, the matrix passed to the method is converted into explicit form, which can be thought of as an adjacency list. Instead of the sparse bag of word representation, each row (sample) maps to a list of integers that denote the indices of the words of the phrase in the vocabulary. The explicit representation is implemented as `List<int[]>`.

Figure 1 depicts an example of the bag of words representation compared to the explicit representation.

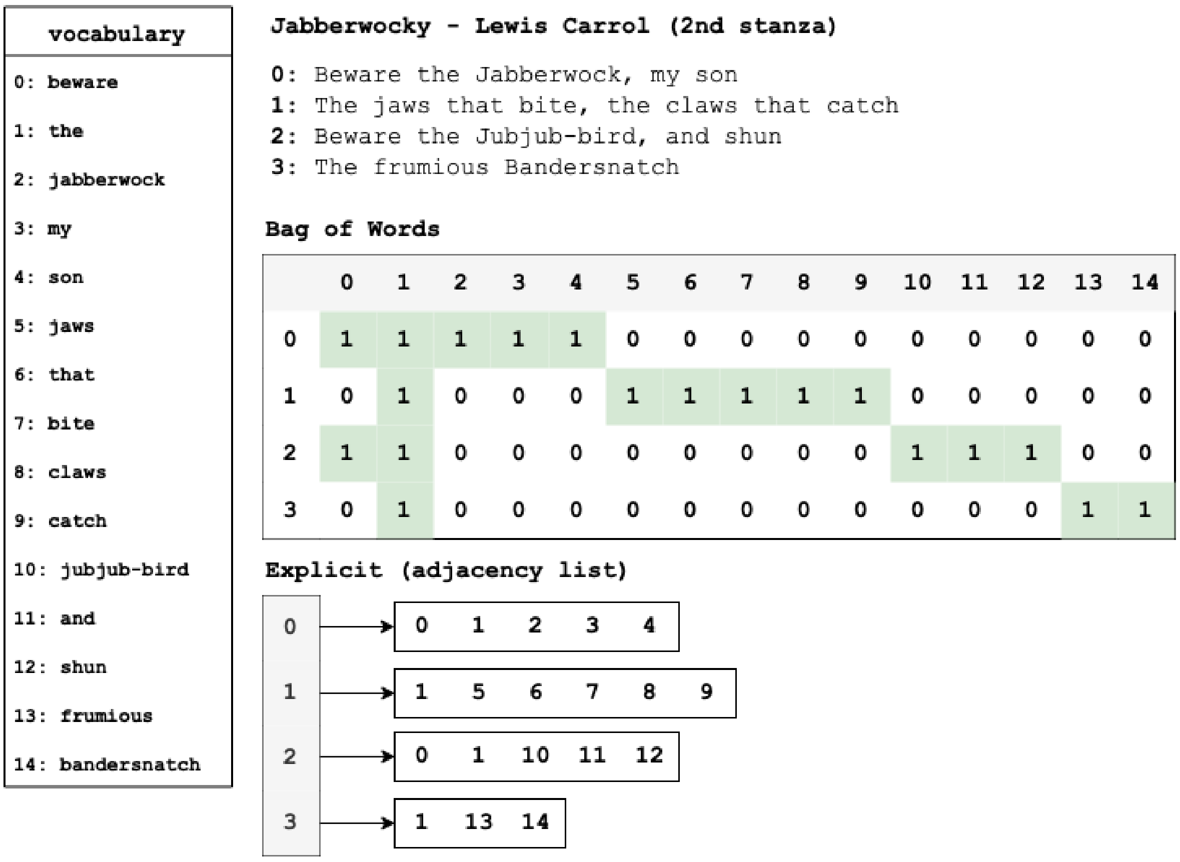
This provides a crucial advantage when performing the computation of the forward propagation. Instead of performing a full matrix multiplication of $X \times W$, we only consider those elements of W, that appear in the corresponding list in the explicit representation. More precisely, each element i, j in the resulting matrix is calculated by taking column j of W ($W_{\cdot j}$) and summing up the elements of the indices that appear in the list at index i in the explicit representation.

Algorithm 1 details this calculation step.

The `backward` propagation step uses a similar computation.

First, the explicit representation (computed in the forward step) is transposed. This is equivalent to transposing the matrix X to X^T and then converting X^T to explicit form.

The calculation step required is $gW = X^T * gY$. Thus, we perform the same calculation as in the forward step, only this time with the explicit representation of X^T and the matrix gY.



vocabulary

0:	beware
1:	the
2:	jabberwock
3:	my
4:	son
5:	jaws
6:	that
7:	bite
8:	claws
9:	catch
10:	jubjub-bird
11:	and
12:	shun
13:	frumious
14:	bandersnatch

Jabberwocky - Lewis Carrol (2nd stanza)

```
0: Beware the Jabberwock, my son
1: The jaws that bite, the claws that catch
2: Beware the Jubjub-bird, and shun
3: The frumious Bandersnatch
```

Bag of Words

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
0	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0
1	0	1	0	0	0	1	1	1	1	1	0	0	0	0	0
2	1	1	0	0	0	0	0	0	0	0	1	1	1	0	0
3	0	1	0	0	0	0	0	0	0	0	0	0	0	1	1

Explicit (adjacency list)

$$\begin{tikzcd}
0 \arrow[r] & \boxed{0\ 1\ 2\ 3\ 4} \\
1 \arrow[r] & \boxed{1\ 5\ 6\ 7\ 8\ 9} \\
2 \arrow[r] & \boxed{0\ 1\ 10\ 11\ 12} \\
3 \arrow[r] & \boxed{1\ 13\ 14}
\end{tikzcd}$$

Figure 1: Example of bag of words representation and explicit representation.

Algorithm 1 Calculate element `Y[i,j]` from `Y = X * W`

```
procedure EmbeddingBag.calcElem(i,j)
    sum = 0
    for all index in explicit[i] do
        sum += W[index,j]
    end for
    return sum
end procedure
```

2.4 Part 3

Part 3 expands on Part 2 by using a pre-computed word embedding, specifically the GloVe model.

To use the GloVe model, the provided matrix is parsed as a `DoubleMatrix` in the `fromFile` method of the `PreComputedWordEmbedding` class.

By adding an additional constructor to `EmbeddingBag`, the parsed matrix is passed as the initial matrix `W`. In case the provided `outdims` are smaller than the number of columns in `W`, all excess columns are truncated.

2.5 Part 4

Part 4 is similar in principle to Part 3. However, instead of using the GloVe model, the word2vec model is used [Mikolov, 2013]. The file contains a vocabulary of 3 million words with a weight vector of 300 elements associated. The goal is to process the raw word2vec data set into a format that can be used as initial weight matrix in the same way as GloVe in Part 3. This process involved the following steps (see `data/part4/word2vec.py`).

1. Load the original vocabulary from part 1 with the words of the training set.
2. Load the raw word2vec file as a keyed vector model using the gensim library.
3. Extract a subset from the word2vec model as the intersection of the part 1 vocabulary and the word2vec vocabulary.
4. Convert the subset model into a text file of the same format as the GloVe model.
5. Extract a new vocabulary file (used for the word indices) from the subset model.
6. Re-tokenize the raw input data sets (`dev.txt`, `train.txt`, `test.txt`) based on the index of the updated vocabulary file. If a word does not occur in the vocabulary, it is ignored. Save the new files as text files in the same format as the tokenized files from Part 1.

Running instructions for the word2vec conversion are provided in `data/part4/README.md`.

We can now use the files created by the script as input for the model. Use the following paths as input:

```
Train data set:     data/part4/output/train.txt
Dev data set:       data/part4/output/dev.txt
Test data set:      data/part4/output/test.txt
word2vec embedding: data/part4/output/subset_model.txt
Classes:            data/part4/input/classes.txt
```

The word2vec embedding is used as input during the creation of the `EmbeddingBag` layer as the initial weight matrix.

3 Test Summary

Key parts of the implementation were tested using JUnit tests. This was particularly useful for the helper methods of the `EmbeddingBag` class, that are responsible for the conversion of the bag of words to explicit representation as well as the multiplication methods. The unit tests can be run as described in Section 1.

The `EmbeddingBagTest` class includes JUnit tests to check the gradient of the embedding layer using the provided utility class of the minet library. This is to verify the gradients of the `EmbeddingBag` layer are computed correctly. The first tests a simple scenario, while the second tests runs 100 tests on random matrices, verifying the correctness for each test.

In addition, extensive manual testing with dummy input data was conducted, especially during the implementation of the `EmbeddingBag` class as well as the word2vec transformation script.

Lastly, I set up a GitHub actions workflow, which runs the unit tests whenever updates are pushed into the private repository, hosted on GitHub. The workflow can be found in the `.github/workflows/main.yml` file. Using this continuous integration workflow had the benefit of detecting breaking changes and bugs early on.

4 Evaluation and Conclusions

In this section, the performance and accuracy of the ANN models is evaluated from different perspectives before concluding with some general remarks.

4.1 Evaluation

4.1.1 Part 1

Using the provided parameters for Part 1, the model achieves a test accuracy of 0.7700 after 64 epochs of training (random seed: 123).

It is likely that tuning the hyperparameters of the model could further increase that performance. In particular, we tune the following parameters:

```
Learning rate:                  {0.05, 0.1, 0.15, 0.2}
Size of first hidden layer:     {25, 50, 100}
Size of other hidden layers:    {50, 100, 200, 400}
```

After fitting a model for each of these configurations (48 in total), the best test accuracy was achieved for the model with the following parameters.

```
Learning rate:                  0.2
Size of first hidden layer:     100
Size of other hidden layers:    100
```

Figure 3 plots the accuracy scores of the model configurations depending on each of the parameters. The blue line is a simple linear regression and the blue shadow shows the confidence intervals.

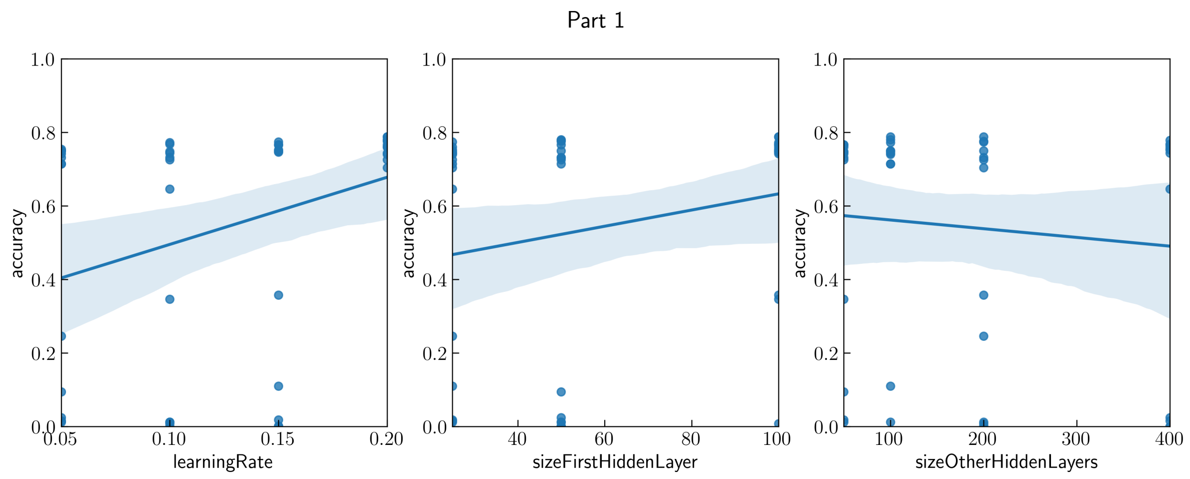


Figure 2: Tendencies of tuning hyperparameters for Part 1.

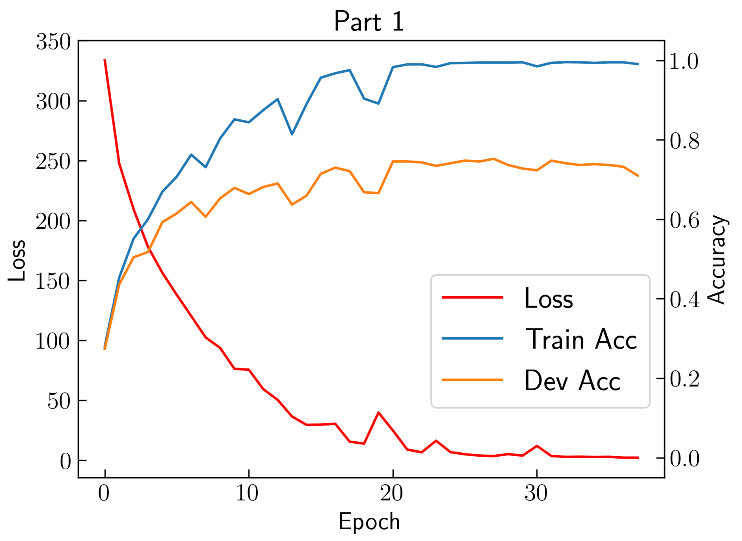


Figure 3: Loss, training accuracy and validation accuracy during the training of the part 1 model.

From Figure3 it appears that, a higher learning rate provides better accuracy. The same is the case for the size of the first hidden layer, while the accuracy score appears to be better for smaller sizes of the other layers. Note that these findings are only valid for the tested range of the parameters and should not be extrapolated.

Consider Figure 3, which plots the value of the loss function, the training set accuracy and the validation set accuracy over the course of the epochs during training. The tuned hyperparameters are used.

Over the course of training, the loss decreases (mostly) steadily, while the accuracy scores on both sets increases (mostly) steadily - which is expected. The gradient of all three curves approaches zero as we reach the end of training, which suggests that we would likely not benefit from extending the training by increasing the patience parameter.

Note also, that the difference between the two accuracy curves are slightly above 20% apart from each other for the last epochs. Were they far apart, the model would likely be underfit. The difference seen in the model here appears reasonable - possibly with a slight underfit.

4.1.2 Part 2

In part 2, the fact that the bag of words matrix is very sparse was leveraged to simplify and speed up the matrix computations.

To evaluate this, two models were created: one with a single linear layer representing Part 1, and another one with a single `EmbeddingBag` layer representing Part 2. Secondly, 100 random matrices were created for each vocabulary size from {`10, 100, 1000, 10000, 100000`} and the forward propagation for the two models was performed given these matrices.

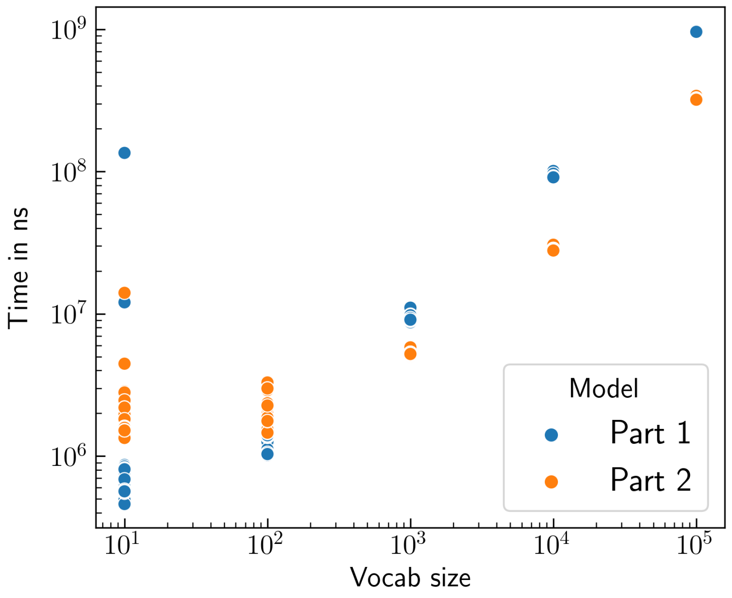


Figure 4: Comparison of the forward propagation time (nanoseconds) on 100 random matrices of different vocab sizes.

Figure 4 gives the results of the experiment. Part 1 is represented by the blue points, Part 2 by the orange ones. The axes show a log scale to better capture the exponential nature of the results. The x-axis shows the computation time in nanoseconds. All experiments were run on the host server of the school of computer science.

Notice that the matrix computation of Part 1 tends to outperform the computation strategy of Part 2 for the smallest vocab size (10). For all larger vocab sizes, the implementation of Part 2 is faster, and most notably for vocab sizes 1000 and 10000, it outperforms Part 1 by almost one order of magnitude. Given that the vocab sizes tend to be large in reality, this improvement for large vocab sizes is significant.

Since the model of Part 2 is the same as in Part 1 (except for the different matrix calculation), no additional hyperparameter tuning was performed for the second part.

4.1.3 Part 3

Provided the different initial weight matrix provided by the GloVe embedding, hyperparameter tuning is conducted for the third model. Similar to part 1, Figure 5 presents the tendencies in accuracy given changes in the three tuned parameters. The range of parameter values was the same as in part 1.

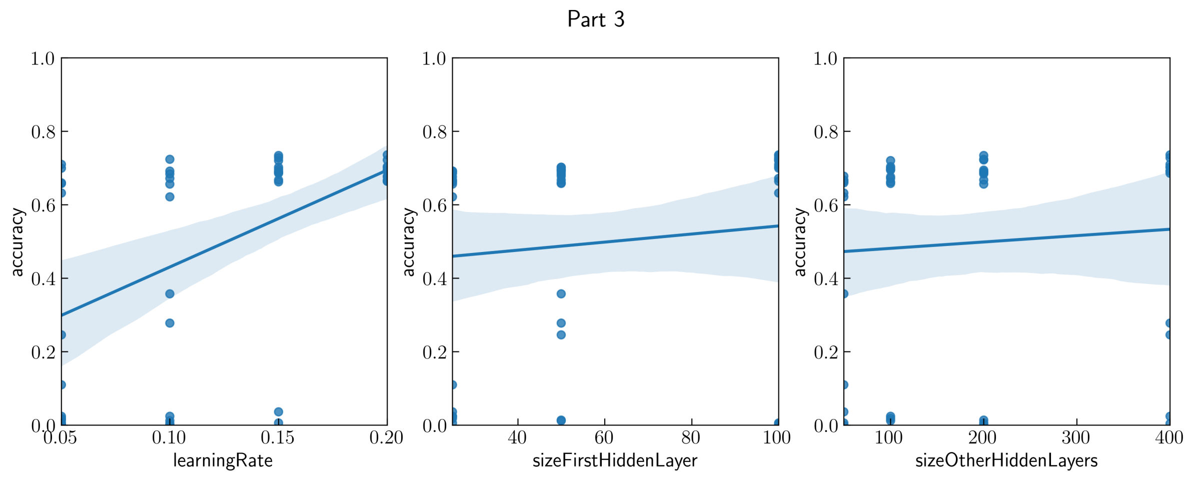


Figure 5: Tendencies of tuning hyperparameters for Part 3.

For all three parameters, `learningRate`, `sizeFirstHiddenLayer`, `sizeOtherHiddenLayers`, an increase appears to benefit the accuracy. This is very pronounced for the learning rate, but significantly less so for the other parameters.

After tuning, the optimal parameters were the following.

```
Learning rate:                0.2
Size of first hidden layer:   100
Size of other hidden layers:  400
```

Using these settings, Figure 6 shows the training process. The training and validation accuracy appear further apart than in Part 1. This could suggest an underfit of the model, which

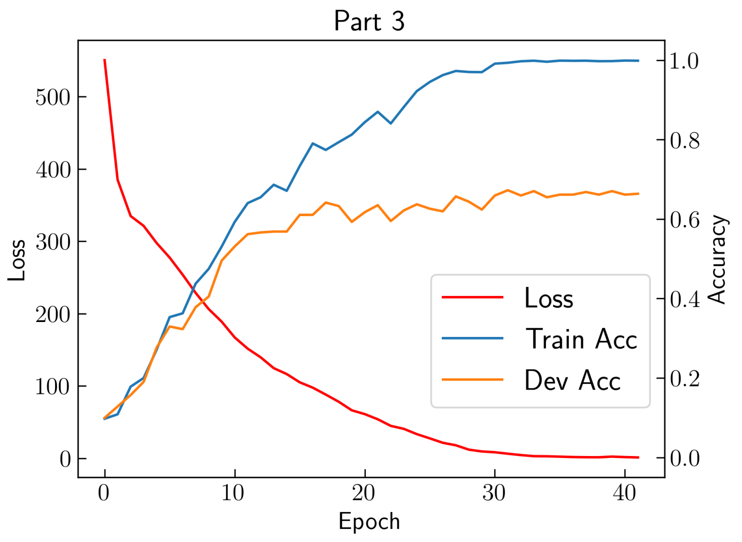


Figure 6: Loss, training accuracy and validation accuracy during the training of the part 3 model.

potentially could be resolved by adding additional layers or further increasing the size of the layers.

The test accuracy of the Part 3 model with the tuned parameters was 0.7460 (after 48 epochs with random seed 123), i.e. less than the first model, which means that the GloVe embedding did not improve the model - however, with more expansive parameter tuning this could change.

4.1.4 Part 4

For Part 4, hyperparameter tuning was conducted in the same way as before with the following results:

```
Learning rate:                  0.2
Size of first hidden layer:     50
Size of other hidden layers:    400
```

As in all cases so far, a higher learning rate appears to improve accuracy. The same holds for the size of the first layer, while the opposite is true for the size of the other layers (see Figure 7. However, the results are less pronounced for part 4.

Similar to Part 3, the gap between training and validation accuracy was slightly bigger than in part 1 (25%), which could suggest underfit.

With the tuned parameters, the part 4 model yielded a test accuracy of 0.7680 after 46 epochs (random seed 123), again not outperforming part 1.

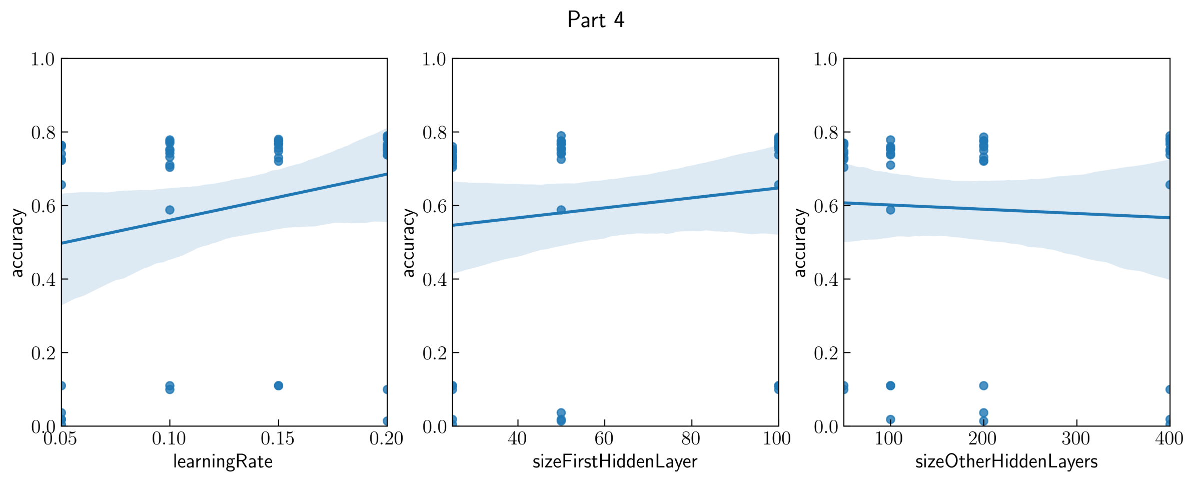


Figure 7: Tendencies of tuning hyperparameters for Part 4.

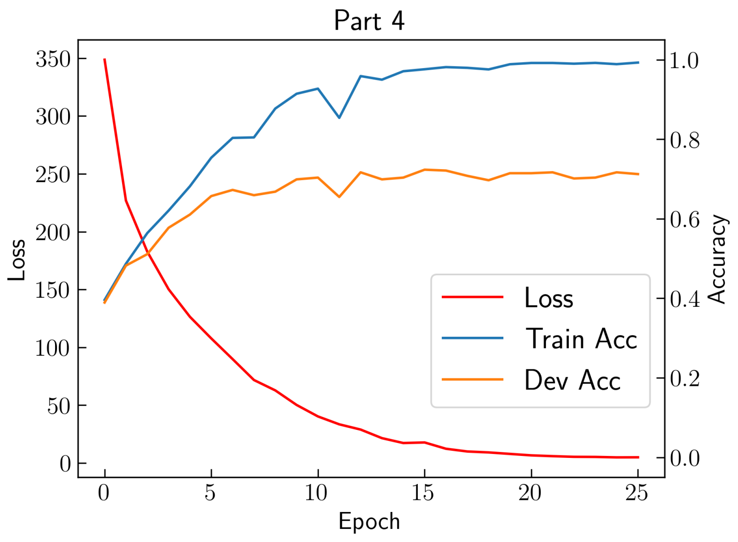


Figure 8: Loss, training accuracy and validation accuracy during the training of the part 4 model.

4.2 Concluding remarks

In this practical, an ANN for classification tasks on the TREC data set was implemented, as well as an `EmbeddingBag` layer for improved computation time. The GloVe and word2vec embeddings were used as initialized weight matrices.

hyperparameter tuning showed that all models can achieve models accuracy scores in the range of approximately 75% to 79%. Using pre-computed embeddings did not improve performance, however, the potential underfit (as seen in Figures 5 and 7) could suggest that adding more layers or increasing layer sizes could increase performance. In fact, to further improve performance of any of the models, hyperparameter tuning should be expanded to include more parameteres and wider parameter ranges.

References

T. Mikolov. word2vec, 2013. `https://code.google.com/archive/p/word2vec/`.

R. Řehůřek and P. Sojka. Software Framework for Topic Modelling with Large Corpora. In *Proceedings of the LREC 2010 Workshop on New Challenges for NLP Frameworks*, pages 45–50, Valletta, Malta, May 2010. ELRA. `http://is.muni.cz/publication/884893/en`.