CS5011 A4

200010781

MAY 10, 2022

Table of Contents

1 Introduction

This report describes the fourth assignment for CS5011 module. General information can be seen in the Table 1 below.

Table 1 General information of the assignment

Assignment Number	Assignment 4
Date of Submission	10 May 2022
Student ID	200010781
Word Count	1998

There are three parts in this assignment, including building an ANN question classifier for the TREC dataset in part 1, implementing an Embedding layer to deal with the issue of slow matrix computation in part 2, using pre-trained word embeddings for setting ANN weights in part 3. The completion of each agent is described as Table 2 below.

Table 2 Completion of each part

Part 1	Build ANN	attempted, fully working
Part 2	Embedding layer	attempted, fully working
Part 3	Initial weights	attempted, fully working

About running the source code, several steps are needed.

1. Enter the terminal.
2. Change to the CS5011_A4 directory.
3. Compile all source code
4. Run the source code

```
java A4Main <part1/part2/part3> <seed> <trainFile> <devFile> <testFile>
<vocabFile> <classesFile>
```

2 Design and Implementation

2.1 Problem Overview

This assignment focuses on constructing and use Artificial Neural Networks (ANNs) to build a question classifier. In this assignment, the PEAS model is described as Table 3 below.

Table 3 PEAS model for assignment 2

Agent	Performance Measure	Environment	Actuator	Sensor
Question classifier agent	Training time, Classify accuracy	Artificial Neural Networks	Classify questions	Using ANN

2.2 System Architecture

All source code can be found in the src directory. The main function is addressed in A4Main.java file which locates in src. In order to keep the structure of this project, there are no packages in the src directory. There are 7 classes related to the functionalities in this assignment.

- A4Main: This class includes the main function of the assignment.
- A4Dataset: This class inherit Dataset class. It mainly aims at reading different datasets. fromFile() method is used to load training, validation and testing datasets and transfer them into one-hot-encoding format. readVocab() method is used to read vocabulary file in part 3 and store the provided words and pre-trained vector values for each word.
- EmbeddingBag: A class for Embedding bag layers. There are forward() and backward() methods for forward and backward propagations. getX() method is used to get the word indices from input object, which will be used in the backward process. initW() method aims at initializing the weight as required in part 3.
- Agent: This class includes the train(), eval() and fromBatch() methods to support training and evaluation processes in agents.
- P1: This class inherit Dataset class. It mainly aims at part 1 of the assignment.
- P2: This class inherit Dataset class. It mainly aims at part 2 of the assignment.
- P3: This class inherit Dataset class. It mainly aims at part 3 of the assignment.

2.3 Implementation

2.3.1 Part 1

This part focuses on building an ANN question classifier for the TREC dataset.

At first, I create three objects of A4Dataset named trainset, devset and testset to represent the training, validation, and testing datasets with batch size set. Then, it calls fromFile() method in A4Dataset to load datasets and change data into one-hot encoding format.

A network is constructed as required in the constructor of P1 class. There are three hidden layers. The size of the first hidden layer is 100, and the sizes for other two are both 200. The outdims equals to 50 since there are 50 classes. In this assignment, I use cross entropy to calculate loss and stochastic gradient descent for optimization.

After building the network successfully, I call train() method for training. Lastly, I call eval() to perform the ANN on test set.

2.3.2 Part 2

This part focuses on using an Embedding layer to deal with the issue of slow matrix computation. P2 class focuses on this part.

General process is similar as part 1. The main difference is that part 2 uses EmbeddingBag to represent the embedding layer in your ANN model.

In the forward() method in EmbeddingBag class, I get the words indices of each sample from input object through getX() method at first. For each sample, I get the indices of each word and use it to find the corresponding weight. Then, sum up all weights and put the sum into Y matrix.

In the backward() method in EmbeddingBag class, I get the words indices of each sample and store it in variable num. Then, for each output node, I sum up gy and gw, where gy refers to the gradient of ouput Y and gw refers to the gradirent of weight W, and this sum result will be put into gW matrix.

Similar as part 1, after building the network successfully, I call train() method for training. Lastly, I call eval() to perform the ANN on test set.

2.3.3 Part 3

This part focuses on using pre-trained word vector from GloVe to initialize weights in embedding layer. P3 class focuses on this part.

General process is similar as part 2. The main difference is that P2 uses random initialization but P3 uses pre-trained word vector to set weights.

At the beginning, I create a new A4Dataset named vocabset, and load it from input path through readVocab() method in A4Dataset. This method put all words into an ArrayList<String> named word and corresponding vector into an ArrayList<double[]> named vectorValues.

A new constructor is created in EmbeddingBag class to pass the vocabset. initW() method is used to initialize the weight using pre-trained vectors. In this method, I use put() method of DoubleMatrix to set the value of each weight.

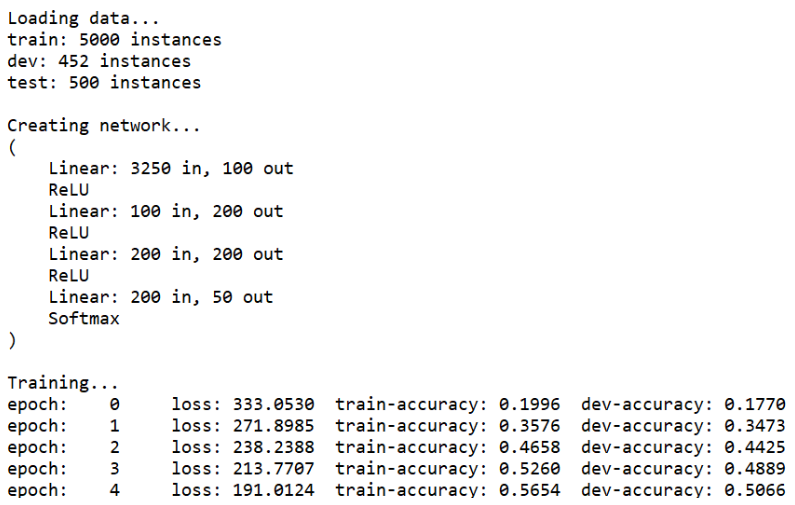
Similar as part 2, after building the network successfully, I call train() method for training. Lastly, I call eval() to perform the ANN on test set.

3 Test Summary

This section mainly discusses the correctness of the system.

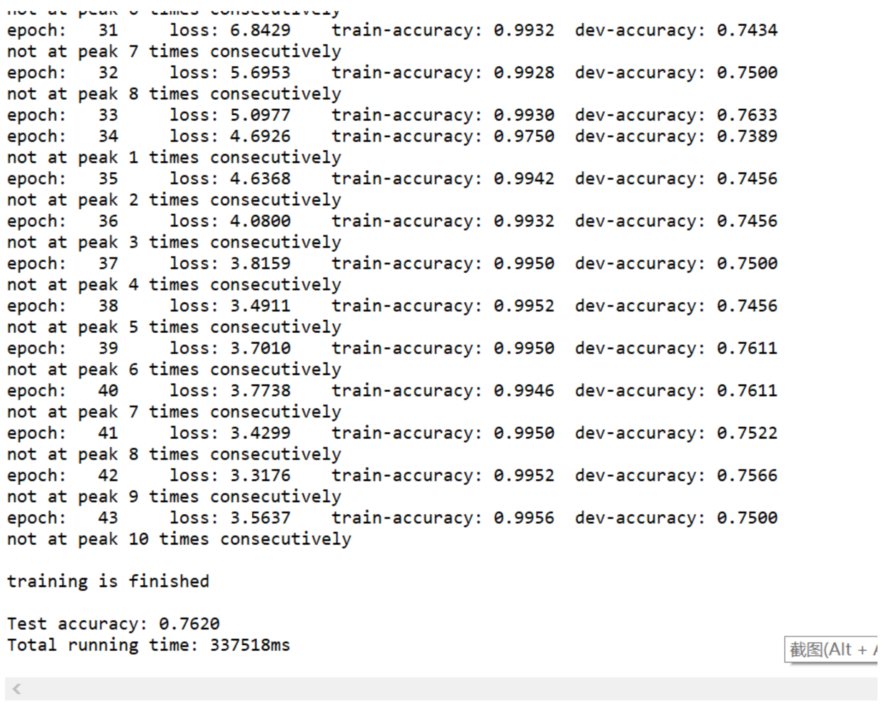
- Part 1

  I show screen shots of the starting and ending epochs of part 1 using starting hyper-parameters with 123 seed. At the beginning, it prints the number of samples in datasets and the network we used in ANN. The training process will stop when there is no improvement for 10 epochs. At the end, I print the accuracies of three datasets and total running time.



```
Loading data...
train: 5000 instances
dev: 452 instances
test: 500 instances

Creating network...
(
    Linear: 3250 in, 100 out
    ReLU
    Linear: 100 in, 200 out
    ReLU
    Linear: 200 in, 200 out
    ReLU
    Linear: 200 in, 50 out
    Softmax
)

Training...
epoch:    0     loss: 333.0530  train-accuracy: 0.1996  dev-accuracy: 0.1770
epoch:    1     loss: 271.8985  train-accuracy: 0.3576  dev-accuracy: 0.3473
epoch:    2     loss: 238.2388  train-accuracy: 0.4658  dev-accuracy: 0.4425
epoch:    3     loss: 213.7707  train-accuracy: 0.5260  dev-accuracy: 0.4889
epoch:    4     loss: 191.0124  train-accuracy: 0.5654  dev-accuracy: 0.5066
```

Figure 1 The start of part 1 using starting hyper-parameters with 123 seed

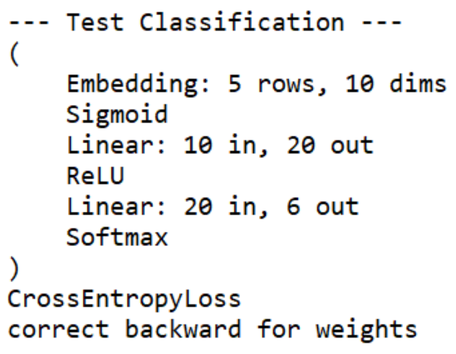


```
epoch:   31     loss: 6.8429    train-accuracy: 0.9932  dev-accuracy: 0.7434
not at peak 7 times consecutively
epoch:   32     loss: 5.6953    train-accuracy: 0.9928  dev-accuracy: 0.7500
not at peak 8 times consecutively
epoch:   33     loss: 5.0977    train-accuracy: 0.9930  dev-accuracy: 0.7633
epoch:   34     loss: 4.6926    train-accuracy: 0.9750  dev-accuracy: 0.7389
not at peak 1 times consecutively
epoch:   35     loss: 4.6368    train-accuracy: 0.9942  dev-accuracy: 0.7456
not at peak 2 times consecutively
epoch:   36     loss: 4.0800    train-accuracy: 0.9932  dev-accuracy: 0.7456
not at peak 3 times consecutively
epoch:   37     loss: 3.8159    train-accuracy: 0.9950  dev-accuracy: 0.7500
not at peak 4 times consecutively
epoch:   38     loss: 3.4911    train-accuracy: 0.9952  dev-accuracy: 0.7456
not at peak 5 times consecutively
epoch:   39     loss: 3.7010    train-accuracy: 0.9950  dev-accuracy: 0.7611
not at peak 6 times consecutively
epoch:   40     loss: 3.7738    train-accuracy: 0.9946  dev-accuracy: 0.7611
not at peak 7 times consecutively
epoch:   41     loss: 3.4299    train-accuracy: 0.9950  dev-accuracy: 0.7522
not at peak 8 times consecutively
epoch:   42     loss: 3.3176    train-accuracy: 0.9952  dev-accuracy: 0.7566
not at peak 9 times consecutively
epoch:   43     loss: 3.5637    train-accuracy: 0.9956  dev-accuracy: 0.7500
not at peak 10 times consecutively

training is finished

Test accuracy: 0.7620
Total running time: 337518ms
```

Figure 2 The end of part 1 using starting hyper-parameters with 123 seed

- Part 2

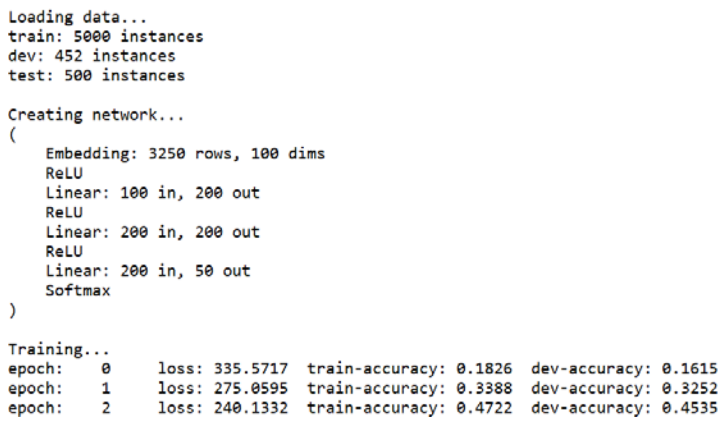
  In order to make sure my implementation is correct, I use gradient checker and compute manually to make sure the correctness of part 2.



```
--- Test Classification ---
(
    Embedding: 5 rows, 10 dims
    Sigmoid
    Linear: 10 in, 20 out
    ReLU
    Linear: 20 in, 6 out
    Softmax
)
CrossEntropyLoss
correct backward for weights
```

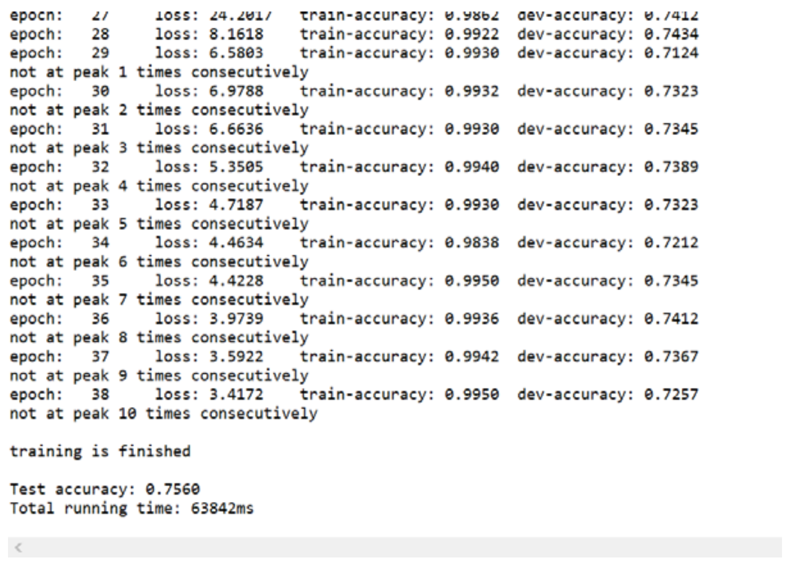
Figure 3 Use GradientChecker to ensure the correctness

In figure 4 and figure 5, I show screen shots of the starting and ending epochs of part 2 using starting hyper-parameters with 123 seed. Similar as part 1, it prints the number of samples in datasets and the network we used in ANN at the beginning. The training process will stop when there is no improvement for 10 epochs. At the end, I print the accuracies of three datasets and total running time as well. In part 2, I fix the issue of slow matrix computation. Therefore, the total training time is far less than part 1.



```
Loading data...
train: 5000 instances
dev: 452 instances
test: 500 instances

Creating network...
(
    Embedding: 3250 rows, 100 dims
    ReLU
    Linear: 100 in, 200 out
    ReLU
    Linear: 200 in, 200 out
    ReLU
    Linear: 200 in, 50 out
    Softmax
)

Training...
epoch:    0     loss: 335.5717  train-accuracy: 0.1826  dev-accuracy: 0.1615
epoch:    1     loss: 275.0595  train-accuracy: 0.3388  dev-accuracy: 0.3252
epoch:    2     loss: 240.1332  train-accuracy: 0.4722  dev-accuracy: 0.4535
```

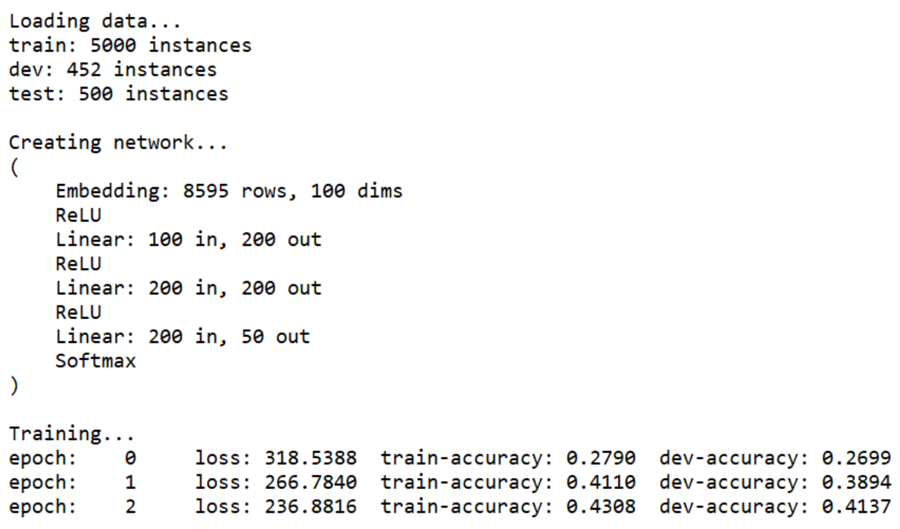
Figure 4 The start of part 2 using starting hyper-parameters with 123 seed



```
epoch:   27     loss: 24.2017   train-accuracy: 0.9862  dev-accuracy: 0.7412
epoch:   28     loss: 8.1618    train-accuracy: 0.9922  dev-accuracy: 0.7434
epoch:   29     loss: 6.5803    train-accuracy: 0.9930  dev-accuracy: 0.7124
not at peak 1 times consecutively
epoch:   30     loss: 6.9788    train-accuracy: 0.9932  dev-accuracy: 0.7323
not at peak 2 times consecutively
epoch:   31     loss: 6.6636    train-accuracy: 0.9930  dev-accuracy: 0.7345
not at peak 3 times consecutively
epoch:   32     loss: 5.3505    train-accuracy: 0.9940  dev-accuracy: 0.7389
not at peak 4 times consecutively
epoch:   33     loss: 4.7187    train-accuracy: 0.9930  dev-accuracy: 0.7323
not at peak 5 times consecutively
epoch:   34     loss: 4.4634    train-accuracy: 0.9838  dev-accuracy: 0.7212
not at peak 6 times consecutively
epoch:   35     loss: 4.4228    train-accuracy: 0.9950  dev-accuracy: 0.7345
not at peak 7 times consecutively
epoch:   36     loss: 3.9739    train-accuracy: 0.9936  dev-accuracy: 0.7412
not at peak 8 times consecutively
epoch:   37     loss: 3.5922    train-accuracy: 0.9942  dev-accuracy: 0.7367
not at peak 9 times consecutively
epoch:   38     loss: 3.4172    train-accuracy: 0.9950  dev-accuracy: 0.7257
not at peak 10 times consecutively

training is finished

Test accuracy: 0.7560
Total running time: 63842ms
```

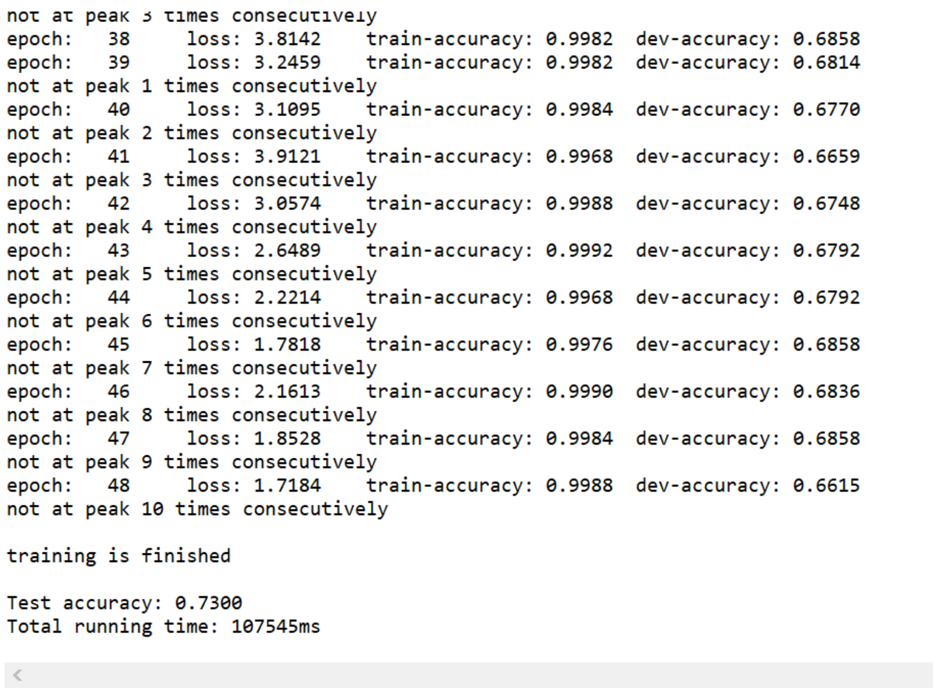
Figure 5 The end of part 2 using starting hyper-parameters with 123 seed

- Part 3
  In figure 6 and figure 7, I show screen shots of the starting and ending epochs of part 3 using starting hyper-parameters with 123 seed. Similar as previous parts, it prints the number of samples in datasets and the network we used in ANN at the beginning. Here, we use data from part3 directory, so the number of nodes in input layer is 8595. At the end, I print the accuracies of three datasets and total running time as well.



```
Loading data...
train: 5000 instances
dev: 452 instances
test: 500 instances

Creating network...
(
    Embedding: 8595 rows, 100 dims
    ReLU
    Linear: 100 in, 200 out
    ReLU
    Linear: 200 in, 200 out
    ReLU
    Linear: 200 in, 50 out
    Softmax
)

Training...
epoch:    0     loss: 318.5388  train-accuracy: 0.2790  dev-accuracy: 0.2699
epoch:    1     loss: 266.7840  train-accuracy: 0.4110  dev-accuracy: 0.3894
epoch:    2     loss: 236.8816  train-accuracy: 0.4308  dev-accuracy: 0.4137
```

Figure 6 The start of part 3 using starting hyper-parameters with 123 seed



```
not at peak 3 times consecutively
epoch:   38     loss: 3.8142    train-accuracy: 0.9982  dev-accuracy: 0.6858
epoch:   39     loss: 3.2459    train-accuracy: 0.9982  dev-accuracy: 0.6814
not at peak 1 times consecutively
epoch:   40     loss: 3.1095    train-accuracy: 0.9984  dev-accuracy: 0.6770
not at peak 2 times consecutively
epoch:   41     loss: 3.9121    train-accuracy: 0.9968  dev-accuracy: 0.6659
not at peak 3 times consecutively
epoch:   42     loss: 3.0574    train-accuracy: 0.9988  dev-accuracy: 0.6748
not at peak 4 times consecutively
epoch:   43     loss: 2.6489    train-accuracy: 0.9992  dev-accuracy: 0.6792
not at peak 5 times consecutively
epoch:   44     loss: 2.2214    train-accuracy: 0.9968  dev-accuracy: 0.6792
not at peak 6 times consecutively
epoch:   45     loss: 1.7818    train-accuracy: 0.9976  dev-accuracy: 0.6858
not at peak 7 times consecutively
epoch:   46     loss: 2.1613    train-accuracy: 0.9990  dev-accuracy: 0.6836
not at peak 8 times consecutively
epoch:   47     loss: 1.8528    train-accuracy: 0.9984  dev-accuracy: 0.6858
not at peak 9 times consecutively
epoch:   48     loss: 1.7184    train-accuracy: 0.9988  dev-accuracy: 0.6615
not at peak 10 times consecutively

training is finished

Test accuracy: 0.7300
Total running time: 107545ms
```

Figure 7 The end of part 2 using starting hyper-parameters with 123 seed

4 Evaluation

This section mainly discusses the performance of different parts. Firstly, I use different hyper-parameters to compare their effects to performance in part 1. Secondly, I use 6 different seeds to compare how seed effects the performance. Finally, I calculate the average epochs, loss, running time and accuracy rate of three parts to compare their performance.

1. **Compare the effects of hyper-parameters to performance in part 1**
   Using the starting hyper-parameters: learning rate: 0.1; maximum number of epochs: 500; patience parameter: 10; batch size: 50, the performance of part 1 is shown in the table below.

Table 4 Performance of part 1 using starting hyper-parameters with 123 seed

Seed (123)	Num of epochs	Loss	Running time(ms)	Training Accuracy(%)	Validation Accuracy(%)	Testing Accuracy(%)
Starting Parameters	43	3.5637	323257	99.56	75.00	76.20

In order to find out the effects of each hyper-parameter to performance in part 1, I use Variable-controlling approach. I change each hyper-parameter one by one and record the performance data.

Table 5 Performance of part 1 using different parameters with 123 seed

Seed (123)	Num of epochs	Loss	Running time(ms)	Training Accuracy(%)	Validation Accuracy(%)	Testing Accuracy(%)
nEpoch=20	19	27.3746	145406	95.28	72.57	72.6
nEpoch=600	43	3.5637	379002	99.56	75.00	76.2
Patience=5	29	12.9134	224711	99.12	73.45	75.8
Patience=20	53	2.5983	420757	99.50	74.34	75.4
Batchsize=20	39	5.1523	308500	99.60	74.78	79.8
Batchsize=100	53	3.3467	426354	99.06	74.34	74.6
LearnRate=0.05	48	7.8978	362979	99.04	74.78	74.2
LearnRate=0.5	38	2.0997	276911	99.54	70.35	77.2
LearnRate=5	10	1485.4437	70589	7.84	6.42	24.6

Analysis

Comparing Table 4 and Table 5, different parameters have different effects.

- nEpoch
  Number of epochs controls the maximum iteration of the training process. As shown in Table 4, when setting nEpoch=500, we need 43 epochs to converge. When changing nEpoch to 600, there is no significant differences to performance. This is because the number of epochs we need is far more less than 500. However, if changing nEpoch to 20 as shown in the first line of Table 5, the number of epochs is exactly 20 (starting from 0). It is clear that the loss is higher than using large nEpoch and the accuracies of three datasets decrease. This is because we limit the maximum iteration to 20. The training process will stop when it reaches to 20 epochs where it has not reached the optimal solution yet.

- Patience
  This parameter is used to control the number of epochs on which we would like to see no improvement. When setting patience=10, that means if we see no improvements for 10 epochs, the training process will stop. As shown in Table 5, if I set patience = 5, the training process will stop when no improvements for 5 epochs. In this situation, the loss

is higher than patience = 10 and the accuracies of three datasets are slightly lower. When setting patience = 20, the training process will stop when no improvements for 20 epochs. In this situation, the loss is slightly lower than patience = 10 and the accuracies of the three datasets are quite similar. According to these different results, smaller patience value may lead to a situation that model has not reached optimum and stops.

- Batch size
  The batch size defines the number of samples that will be propagated through the network. When setting batchsize=20, networks train faster with less running time, but less accurate than using batchsize=50. When setting Batchsize = 100, time for training rises, but reaches higher accurate with smaller loss.

- Learning rate
  The learning rate is a hyperparameter that controls how much to change the model in response to the estimated error each time the model weights are updated. It helps to minimize the network's loss function. When learning rate is small, such as learnRate=0.05, the networks spend longer time to train. This is because we need more steps to reach the optimum solution. However, if setting learnRate too large, such as learnRate=5, the network may not find the best solution, which leads to low accuracies for all datasets as shown in Table 5.

2. Compare the performance using different seeds

In order to find out the effects for all three parts using different seeds, I use 6 different seed (123, 10, 50, 500, 5000, 10000) and record the number of epochs, loss, running time, and training accuracies for three datasets. All results are shown in the Table 6 below.

From Table 6, we can't see the obvious rules between seed and the performance of ANN. But different seeds do lead to different results.

Table 6 Results of three parts using different random seed

Part	Seed	Num of epochs	Loss	Running time(ms)	Training Accuracy(%)	Validation Accuracy(%)	Testing Accuracy(%)
Part 1	**10**	28	15.0994	297766	99.58	74.34	79.6
Part 1	**50**	33	5.1418	258132	99.32	73.23	77.2
Part 1	**123**	43	3.5637	323257	99.56	75.00	76.2
Part 1	**500**	54	2.6087	414731	99.56	73.67	79.4
Part 1	**5000**	48	3.8767	373886	99.52	73.67	76.6
Part 1	**10000**	49	6.4439	514828	99.60	74.34	79.6
Part 2	**10**	20	13.5113	46733	99.30	73.23	77.2
Part 2	**50**	28	18.1244	63197	99.60	73.01	81.0
Part 2	**123**	22	14.4975	51190	99.58	73.45	77.8
Part 2	**500**	25	11.1560	57140	99.56	72.35	79.6
Part 2	**5000**	37	6.9639	82312	99.62	75.22	77.6
Part 2	**10000**	33	7.3886	74215	99.62	73.01	78.6
Part 3	**10**	31	34.1079	119422	96.36	66.81	71.8
Part 3	**50**	32	18.7679	119031	99.02	69.03	74.2
Part 3	**123**	40	1.6524	146240	99.92	68.36	76.0
Part 3	**500**	30	4.1907	109854	99.84	67.26	74.2
Part 3	**5000**	42	1.5148	152376	99.92	67.26	73.0
Part 3	**10000**	53	1.2122	191365	99.92	66.15	74.8

3. Compare the performance of three parts

In order to compare the performances of all three parts, I get the average results of using 6 different seeds. The results are shown in the Table 7 below.

Due to sparse input representation in part 1, it spends the longest time to train. In part 2, after fixing the issue of slow matrix computation, the running time decreases significantly. The testing accuracy of part 2 is slightly lower than part1, I assume it is because we did not use bias in embedding layer. In part 3, I use pre-trained embedding on your ANN models. However, there is no improvement in performance comparing to part 2. Besides, the time spent in part 3 is slightly longer than part 2. I assume this is because part 3 spend some time on assigning vector values to weights in embedding layer.

Average (on 6 seeds)	Num of epochs	Loss	Running time(ms)	Training Accuracy(%)	Validation Accuracy(%)	Testing Accuracy(%)
Part 1	42.5	6.1224	363767	99.52	74.04	78.1
Part 2	27.5	11.9403	62464	99.54	73.38	77.6
Part 3	38	10.2409	139714	99.16	67.47	74.0

5 Conclusion

There are three parts in this assignment, including building an ANN question classifier for the TREC dataset in part 1, implementing an Embedding layer to deal with the issue of slow matrix computation in part 2, using pre-trained word embeddings for setting ANN weights in part 3.

In part 1, I compare the effects of each parameter to the performance of ANN model. It can be seen that network trains faster with mini batches but lower accuracy. Too small nEpoch may cause a network stop early where it has not reached the optimum solution. Similar to parameter patience, smaller patience value may lead to a situation that model has not reached optimum and stops. It is tricky to choose the right learning rate in machine learning problem. It depends on different model with different loss function. Large learning rate may lead the network miss the minimum loss and small learning rate increases the training time.

There are no obvious rules between seed and the performance of ANN. After calculating the average results of three parts, part 2 saves more time than part 1 during training. Part 3 needs longer time than part 2 but less time than part 1. The reason might be assigning vector values to weights spends some time. In terms of accuracy, part 2 has slightly lower accuracy than part 1. I assume it is because we consider no bias in the embedding layer. There are no improvements in part 3 comparing to part 2.