1. Introduction

Assignment Number: A4
Date: 10/05/2022
Student ID: 210017984
Word Count: 2438 (includes all extensions descriptions and evaluations)

1.1 Parts Implemented

Part	Status
Part 1	Attempted, fully working.
Part 2	Attempted, fully working.
Part 3	Attempted, fully working.
Part 4	Attempted, fully working. 1. Freezing Embedding Layer
Part 5	Attempted, fully working - word2vec word embeddings (different part because requires different number of layers).
General Extension (applies to all parts)	Randomized search – hyperparameter tuning.

Table 1: Parts Implemented.

1.2 Compiling & Running Instruction

1. Navigate to the **src** directory included in the AI-Neural-Network folder:

```
cd AI-Neural-Network/src/
```

2. From within src, **compile and run** the program:

```
./build.sh
```

```
java A4Main <part1/part2/part3/part4/part5> <seed> <trainFile> <devFile> <testFile> <vocabFile> <classesFile> <silent> <tune>
```

Arguments Explanation	
Argument	**Explanation**
`<part1/part2/part3/part4/part5>`	Exercise you wish to run
`<seed>`	Can be used to generate reproducible results (pseudo random
`<trainFile> <devFile> <testFile>`	Path for the training, validation, and testing datasets respectively
`<vocabFile> <classesFile>`	Path of the vocabulary and output classes files respectively.
`<silent> <tune>`	*silent* does the training of the network without printing any intermediate steps, and tune runs the hyperparameter randomized search algorithm developed (extensions).

Table 2: Passed in Arguments

Running Examples:

Run the second part using the datasets and files provided.

```
java –cp lib/*:minet/:src:. src/A4Main part2 123 data/part1/train.txt
data/part1/dev.txt data/part1/test.txt data/part1/vocab.txt
data/part1/classes.txt
```

How to run – Randomized Search Hyperparameter Tuning		
Mode	**Command**	**Instructions**
Silent tune	java -cp lib/*:minet/:src:. src/A4Main part2 123 data/part1/train.txt data/part1/dev.txt data/part1/test.txt data/part1/vocab.txt data/part1/classes.txt silent tune	Enter the command and then follow the instructions given on the terminal, giving a range of values to try for each hyperparameter. **Separate each value with a space**.
Verbose tune	java -cp lib/*:minet/:src:. src/A4Main part2 123 data/part1/train.txt data/part1/dev.txt data/part1/test.txt data/part1/vocab.txt data/part1/classes.txt verbose tune	

Table 3: Instructions to run hyperparameter tuning.

2. Design & Implementation

The task of this practical was to build a basic Artificial Neural Network (ANN) question classifier, that classifies the topic of a question.

Table 24describes the purpose of each of the classes that were created or expanded.

Code Structure	
Class	**Purpose**
A4Main	The class is used to select which part of the practical to run and construct an appropriate network.
VocabDataset	The VocabDataset class extends *minet.data.Dataset* and it is used to load an appropriate vocabulary dataset from a given data file. This class is used to load the training, validation, and testing datasets.
VocabClassifier	The *VocabClassifier* class contains all the functions that are used to train and evaluate a Neural Network.
EmbeddingBag	Implements the Minet's *Layer* interface. Provides a more efficient way of computing the forward and backward pass – addressing the issue of slow matrix computation that occurs when given sparse input vectors.
HyperparameterTuning (Extension)	An extension class that is used to implement the RandomizedSearch algorithm.

Table 4: Code structure and purpose of each class.

2.1 VocabDataset

The most important methods of the *VocabDataset* class are briefly described in Table 5.

VocabDataset Class – Most Important Methods	
Method	**Description**
fromFile	Load data from file and vocabulary.
countLinesInFile	Count the lines in a given file. Used to determine the instances in the set provided and the dimensions of the whole vocabulary. If we are using pre-trained weights then the trainingWeights flag is set to true (upon initialisation of the class) and the weights associated with each word are stored in an ArrayList as a double array (*allWeights*).
oneHotEncode	One hot encode a given dataset using the Bag-of-Word strategy.
createDoubleMatrixForWeights	Creates a DoubleMatrix that stores all the pre-trained weights passed in. Uses the *allWeights* list.

	The DoubleMatrix is used in the embedding layer when choosing to use the pre-trained weights.

Table 5: Most important methods - VocabDataset.

The *oneHotEncode* method transforms each input data into a binary fixed-length vector (Bag-of-Word approach) with length equal to the vocabulary size. Every token (index) that appears in a sample is assigned a value of one; otherwise, it is assigned a value of zero. Figure 1 shows a visual example of how a sample is converted to a one-hot encoded vector.

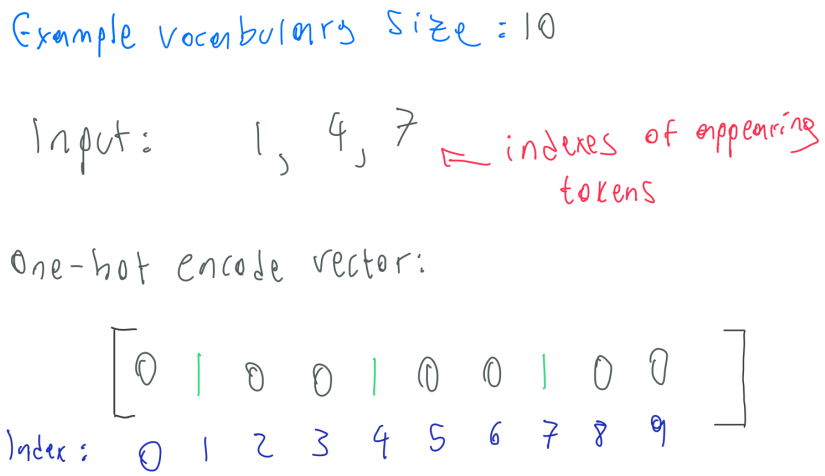


Figure 1: One-hot encoded conversion.

2.2 VocabClassifier

The provided Minet example were the starting foundations and skeleton for this class.
The class most important are described in Table 6.

VocabClassifier Class – Most Important Methods	
Method	**Description**
trainAndEval	Trains and evaluates a neural network. Takes in the network to be trained, and the training, validation, and testing datasets. Initialises CrossEntropy as the loss function and SGD (Stochastic Gradient Descent) as the method to update the ANN weights.
train	Trains the ANN for our NLP problem. The begins by shuffling the dataset before training. It then uses the forward, backward passes and update weights methods to find the network's optimal weights. The *train* method uses the Early Stopping technique, which stops the training when performance on **validation** set stops improving – prevent overfitting.

eval	Evaluate the performance of the ANN during validation and testing. Uses the *forward* function to compute the predictions of the network and then the DoubleMatrix's *rowArgmaxs* method to assign as prediction the output node (NLP classification topic) with the highest probability. Returns the **classification accuracy** - calculated by finding the percentage of correct predictions out of all predictions.
convertToDoubleMatrixPair	Converts a mini-batch of the vocabulary dataset to a data structure that can be used by the network – a pair of f double matrices, where the first element of the pair is the input (X) and the second is the label (Y).

Table 6: Most important methods in VocabClassifier class.

2.3 EmbeddingBag

In our Bag-of-Word approach each sample contains a binary number for all the words/tokens in the vocabulary. When training the model, this leads to a **huge number of unnecessary computations**. To deal with that, the created EmbeddingBag class creates a **dense** layer by calculating the Forward and Backwards pass values, doing computations on elements where the word is present (one-hot value of one). To achieve that, it needs to use the word indexes of each sample (not one-hot). This is achieved using the *getX* method which takes in the one-hot encoded samples and outputs the indexes of where the value is one. I mindfully chose to reconstruct the indexes within the EmbeddingBag class from the one-hot matrix rather than passing the originals. This allowed me to use the same *train* function as the other layers - which accepts a DoubleMatrix, leading to less code repetition and less unnecessary complexity.

2.3.1 Forward

The *forward* method is used to calculate the output of each layer from the data it receives by calculating the **output** of each node using matrix multiplication (weighted sum) between the input data matrix (from previous layer) and the weights (matrix) directed towards that node. The output is then passed to a **non-linear differentiable** activation function which determines by how much the neuron should be activated. Once the last layer is propagated, the loss value is calculated.

Figure 2 depicts a simplified example of how Forward step computes the output of the first hidden layer node in a sparse vs dense matrix.

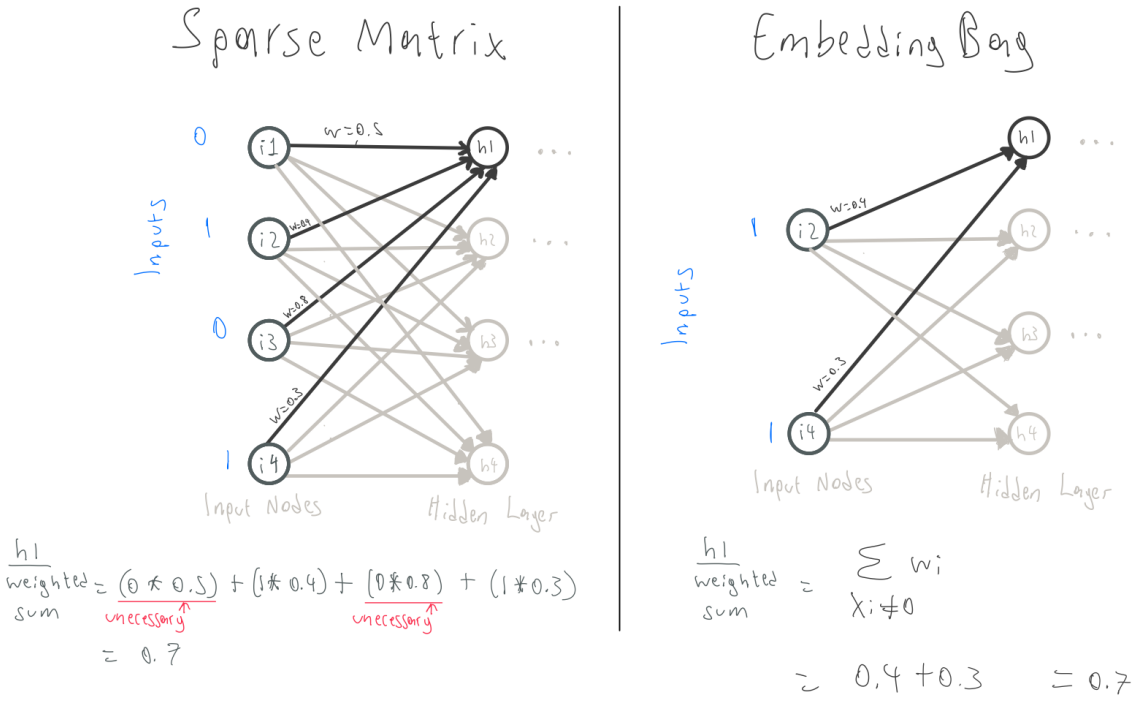


Figure 2: ForwardPass weighted sum calculation - Sparse Matrix vs EmbeddingBag.

The sparse matrix method does a lot of **unnecessary** (2 out of 4 in example) calculations when computing the weighted sum. This problem becomes even worse in the real scenario, as the number of non-zero elements (value of one) are <1% than the vector's length. In contrast, the Embedding Bag just calculates the sum of the weights where the one-hot value is one, leading to an efficient operation. This is done using the *getSumOfWeights* method which gets the sum of all the weights for each sample in the batch, for every dimension and populates the DoubleMatrix Y.

2.3.2 Backward

The backpropagation algorithm allows the network to fine-tune its weights based on the error rate between the predictions made in the forward pass and the actual outputs. It starts at the output layer going back to the input, calculating the gradient of the loss function for a single weight by the chain rule. The way the gradient of the loss regarding a weight is depicted in Figure 3.

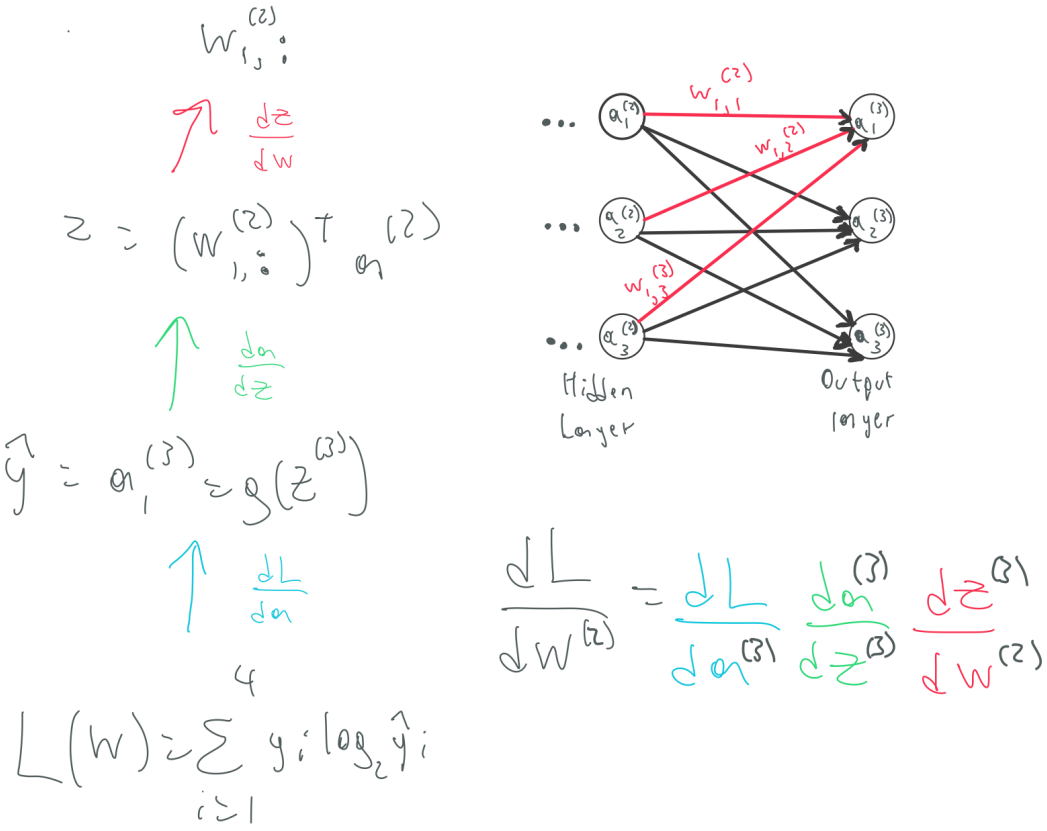


Figure 3: Backward propagation gradient of loss.

As shown above, the gradient of the loss is calculated by using the chain rule on the following partial derivatives:

1. Gradient of loss with regards to activation function (blue)
2. Gradient of activation function with regards to the sum-weight value (z) (green)
3. Gradient of sum-weight value (z) with regards to the weight (red)

Inside the *backward* method of the EmbeddingBag class, we calculate the third partial derivative - gradient of sum-weight value (z) with regards to the weight. To make it efficient, we only calculate the gradient of the weights for the items that have a value of one in the one-hot encoding. This is done by iterating through the nodes of the current layer and then through the samples (in current batch). For each sample we get its non-zero indexes, and then we update its gW value by getting the gY value of the current sample and dimension/node. We then get the prior (before update) gradient value of the gW and we then update gW for the current index in current dimension by adding the prior with the gY value.

2.4 Pre-trained embeddings

To read the weights for each word a new method was created in VocabDataset called *placeInWeightsList* which takes in the weights of each word as one large string and then converts it to a double array containing all the weight values. This double array is then added to the *allWeights* ArrayList<double[]> which holds the weights for all the words in the vocabulary. After each word's weight is added to the ArrayList, *allWeights* is converted into a DoubleMatrix using the *createDoubleMatrixForWeights* method. A new getter function is then created which is used in the Main to get the pretrained weights for the training set.

In the EmbeddingBag class a new constructor was created which takes in the *pretrainedWeights* and assigns them to W (instead of random initialisation).

2.5 Extensions

2.5.1 Freezing EmbeddingBag Layers

As an extension I experimented with implementing the functionality of freezing the EmbeddingBag's layer weights. This was done by simply creating a *freeze* flag which when set to true (Part 4) the gradients of the weights are not updated during backpropagation.

2.5.2 Words2vec – Python (Jupyter Notebook)

I also experimented with generating and feeding the network with words2vec embeddings. My goal was to make them the same style as the ones provided using GloVe so that I can reuse the code created for Part 3. To create these weight embeddings, I used Python and specifically the *spacy* library. All the word weights were first loaded. Then I initialised all the words in the practical's vocabulary and created an empty (zero) array of size vocab_size x 300 dimensions. I then created a simple for loop to populate the matrix, getting the weights for each word by calling *nlp(words_in_vocab[i]).vector*. The *ipynb* file is included with my submission (words2vec_preprocessing folder). The network's layers changed to 300 hidden dimensions for the Embedding Layer and then 400 for the following hidden layers.

2.5.3 Hyperparameter Tuning – Randomized Search

The HyperparametTuning class created implements the popular random search optimization algorithm that randomly tries different combinations to tune the hyperparameters. To achieve this, the A4Main class was extended by creating the *getScannerInputValues* that reads input values from the user (using a Scanner), and *performHyperparameterTuning* that initialises the HyperparamerTuning class and calls the *randomizedSearch* method.

The *randomizedSearch* method works by running as times as the number of iterations. On each iteration it creates a temporary network and then selects a random hyperparameter value from the three ArrayLists that store the hyperparameter values for each hyperparamer being optimized (learningRate, epochs, patience). It then calls a new function in vocabClassifier that trains the new network on the given training and validation sets and input values. Finally, there is a check to check if the validation accuracy of the trained model is better than the previously best. If that is the case

then the learning rate, max epochs, and patience values used get initialised as the best ones found so far. After all the iterations are finished, the best hyperparameters found are returned.

3. Testing

To test the one-hot encoding and the forward pass of the EmbeddingBag, I created a small example containing only 10 training data and 10 vocabulary words (available at "testing" folder).

3.1 One-hot encoding – using small example

One-hot encoding is performed correctly for all the samples, placing a value of 1 on the passed indexes.

```
path: testing/train.txt
SAMPLE 0
1 2 3 4
ONE HOT ENCODED MATRIX: [0.0, 1.0, 1.0, 1.0, 1.0, 0.0, 0.0, 0.0, 0.0, 0.0]

SAMPLE 1
2 4 3
ONE HOT ENCODED MATRIX: [0.0, 0.0, 1.0, 1.0, 1.0, 0.0, 0.0, 0.0, 0.0, 0.0]

SAMPLE 2
4 5 3
ONE HOT ENCODED MATRIX: [0.0, 0.0, 0.0, 1.0, 1.0, 1.0, 0.0, 0.0, 0.0, 0.0]

SAMPLE 3
1 2 5
ONE HOT ENCODED MATRIX: [0.0, 1.0, 1.0, 0.0, 0.0, 1.0, 0.0, 0.0, 0.0, 0.0]

SAMPLE 4
2 5 3
ONE HOT ENCODED MATRIX: [0.0, 0.0, 1.0, 1.0, 0.0, 1.0, 0.0, 0.0, 0.0, 0.0]

SAMPLE 5
1 4 2
ONE HOT ENCODED MATRIX: [0.0, 1.0, 1.0, 0.0, 1.0, 0.0, 0.0, 0.0, 0.0, 0.0]

SAMPLE 6
5 2 1
ONE HOT ENCODED MATRIX: [0.0, 1.0, 1.0, 0.0, 0.0, 1.0, 0.0, 0.0, 0.0, 0.0]

SAMPLE 7
0 4 0 2
ONE HOT ENCODED MATRIX: [1.0, 0.0, 1.0, 0.0, 1.0, 0.0, 0.0, 0.0, 0.0, 0.0]

SAMPLE 8
2 2 4
ONE HOT ENCODED MATRIX: [0.0, 0.0, 1.0, 0.0, 1.0, 0.0, 0.0, 0.0, 0.0, 0.0]

SAMPLE 9
5 2 3
ONE HOT ENCODED MATRIX: [0.0, 0.0, 1.0, 1.0, 0.0, 1.0, 0.0, 0.0, 0.0, 0.0]
```

Figure 4: one-hot encoding test.

3.2 Forward Pass – using small example

The randomly initialised weights are manually checked the on the **first** sample.
Specifically, I tested:

1) The correct weights (indexes) are selected
2) Summation is performed correctly

```
INITIAL WEIGHTS
[0.315616, 0.577177; 0.694236, -0.496510; -0.348896, 0.672062; 0.153867, -0.614340; 0.432
565, -0.608794; 0.530914, -0.578764; 0.305539, -0.106820; -0.605401, 0.529469; 0.418976,
0.106213; 0.111323, -0.146376]
(
    Embedding: 10 rows, 2 dims
    ReLU
    Linear: 2 in, 3 out
    ReLU
    Linear: 3 in, 3 out
    ReLU
    Linear: 3 in, 50 out
    Softmax
)
```

Figure 5: The initial weights assigned and the structure of the very simplified network created for testing.

```
Sample number 0 dimension: 0 | indexes: [2, 3, 5]
All weights of sample:  -0.3488962360276431 0.15386695938051453 0.5309138523908128
Sum weight: 0.33588457574368424

Sample number 0 dimension: 1 | indexes: [2, 3, 5]
All weights of sample:  0.6720624238752921 -0.6143399852311678 -0.5787636565666334
Sum weight: -0.5210412179225091
```

Figure 6: Weights where index is 1.

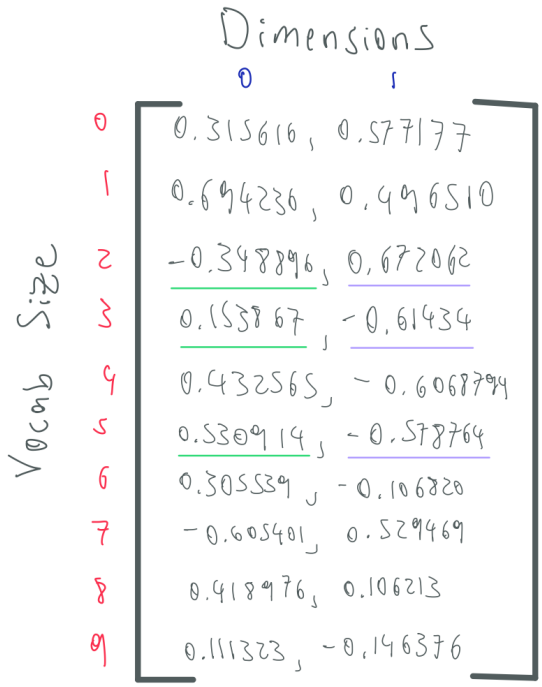


Figure 7:The weights of the sample in a matrix.

Dimension 0

$\sum_{x_i \neq 0} w_i = (-0.348896) + 0.153867 + 0.530914$

$= 0.335885$ (6 d.p.) ✓ correct

Dimension 1

$\sum_{x_i \neq 0} w_i = 0.672062 + (-0.61434) + (-0.578764)$

$= -0.521042$ (6 d.p.) ✓ correct

Figure 8: Forward pass manual calculation to test correctness.

As evident, the forward pass selects the correct weights (non-zero in one-hot) and the summation is correct.

3.3 Backward Pass

To test the *backward* function, I used the Minet's GradientChecker using the following example:

```
public static void testClasificationForEmbedding() {
    System.out.println("Testing Backward Function - Embedding Layer");
    // Initialise X matrix
    DoubleMatrix X = new DoubleMatrix(
            new double[][] {
                    {1.00, 0.00, 0.00, 0.00, 1.00},
                    {1.00, 0.00, 0.00, 1.00, 1.00},
                    {1.00, 1.00, 0.00, 0.00, 0.00},
                    {1.00, 1.00, 0.00, 1.00, 1.00},
                    {0.00, 1.00, 1.00, 1.00, 1.00},
                    {0.00, 0.00, 0.00, 1.00, 0.00}});

    // Initialise ys.
    double[] ys = new double[] {0,1,2,3,4,5};
    DoubleMatrix Y = new DoubleMatrix(ys);

    CrossEntropy loss = new CrossEntropy();
    Sequential net = new Sequential(new Layer[] {
        new EmbeddingBag( vocabSize: 5, outdims: 6, new WeightInitXavier()),
            new Softmax()});

    checkGradient(net, loss, X,Y);
    }
}
```

Figure 9: GradientChecker example.

```
correct backward for weights
```

Figure 10: Output after running gradient checker.

3.4 Pre-trained weight initialisation

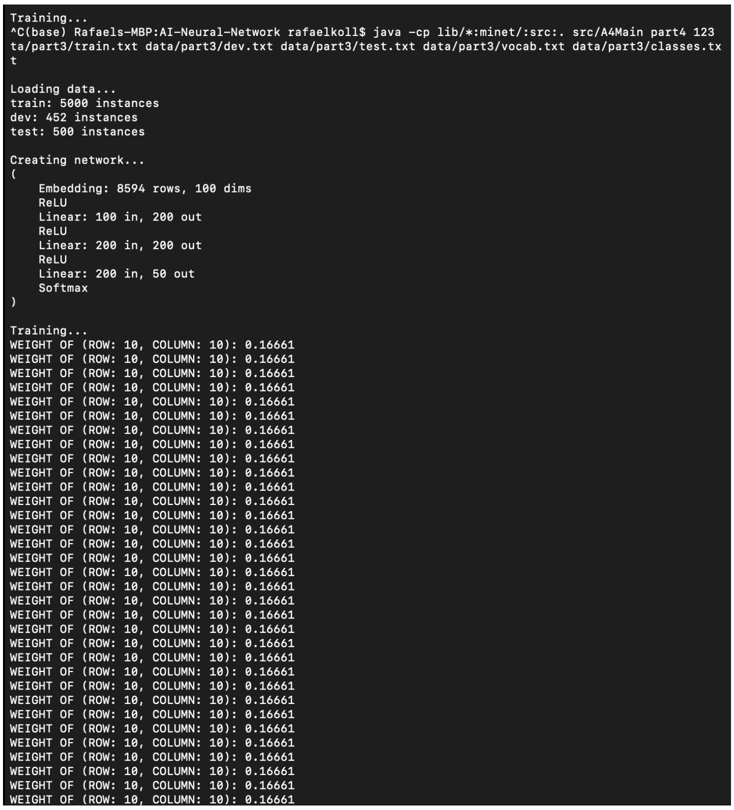
Figure 11: Pre-trained weights of first two words **from vocab file**.

Figure 12: weights loaded from file in the algorithm.

As evident, the pre-trained weights are loaded correctly/match.

3.5 Frozen layers – Embedding Bag (Extension)

Weight indexed 10,10 was randomly chosen. As evident, it is not updated over the iterations.



```
Training...
^C(base) Rafaels-MBP:AI-Neural-Network rafaelkoll$ java -cp lib/*:minet/:src:. src/A4Main part4 123
ta/part3/train.txt data/part3/dev.txt data/part3/test.txt data/part3/vocab.txt data/part3/classes.tx
t

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

Creating network...
(
    Embedding: 8594 rows, 100 dims
    ReLU
    Linear: 100 in, 200 out
    ReLU
    Linear: 200 in, 200 out
    ReLU
    Linear: 200 in, 50 out
    Softmax
)

Training...
WEIGHT OF (ROW: 10, COLUMN: 10): 0.16661
WEIGHT OF (ROW: 10, COLUMN: 10): 0.16661
WEIGHT OF (ROW: 10, COLUMN: 10): 0.16661
WEIGHT OF (ROW: 10, COLUMN: 10): 0.16661
WEIGHT OF (ROW: 10, COLUMN: 10): 0.16661
WEIGHT OF (ROW: 10, COLUMN: 10): 0.16661
WEIGHT OF (ROW: 10, COLUMN: 10): 0.16661
WEIGHT OF (ROW: 10, COLUMN: 10): 0.16661
WEIGHT OF (ROW: 10, COLUMN: 10): 0.16661
WEIGHT OF (ROW: 10, COLUMN: 10): 0.16661
WEIGHT OF (ROW: 10, COLUMN: 10): 0.16661
WEIGHT OF (ROW: 10, COLUMN: 10): 0.16661
WEIGHT OF (ROW: 10, COLUMN: 10): 0.16661
WEIGHT OF (ROW: 10, COLUMN: 10): 0.16661
WEIGHT OF (ROW: 10, COLUMN: 10): 0.16661
WEIGHT OF (ROW: 10, COLUMN: 10): 0.16661
WEIGHT OF (ROW: 10, COLUMN: 10): 0.16661
WEIGHT OF (ROW: 10, COLUMN: 10): 0.16661
WEIGHT OF (ROW: 10, COLUMN: 10): 0.16661
WEIGHT OF (ROW: 10, COLUMN: 10): 0.16661
WEIGHT OF (ROW: 10, COLUMN: 10): 0.16661
WEIGHT OF (ROW: 10, COLUMN: 10): 0.16661
WEIGHT OF (ROW: 10, COLUMN: 10): 0.16661
WEIGHT OF (ROW: 10, COLUMN: 10): 0.16661
WEIGHT OF (ROW: 10, COLUMN: 10): 0.16661
WEIGHT OF (ROW: 10, COLUMN: 10): 0.16661
WEIGHT OF (ROW: 10, COLUMN: 10): 0.16661
WEIGHT OF (ROW: 10, COLUMN: 10): 0.16661
WEIGHT OF (ROW: 10, COLUMN: 10): 0.16661
WEIGHT OF (ROW: 10, COLUMN: 10): 0.16661
WEIGHT OF (ROW: 10, COLUMN: 10): 0.16661
WEIGHT OF (ROW: 10, COLUMN: 10): 0.16661
WEIGHT OF (ROW: 10, COLUMN: 10): 0.16661
```

Figure 13: Frozen weights do not get updated after iterations.

4. Evaluation

4.1 Part 1

4.1.1 Training Process and Performance on Test Data

Each of the ANN links has a weight value associated with it which affects how much influence a change in the input has on the output. A low weight value results in almost no change in output, whereas a higher weight value results in larger changes. For example, in Figure x, the first input node has the strongest link with the second node in the first hidden layer, influencing its output four times more than the second input node (weight).

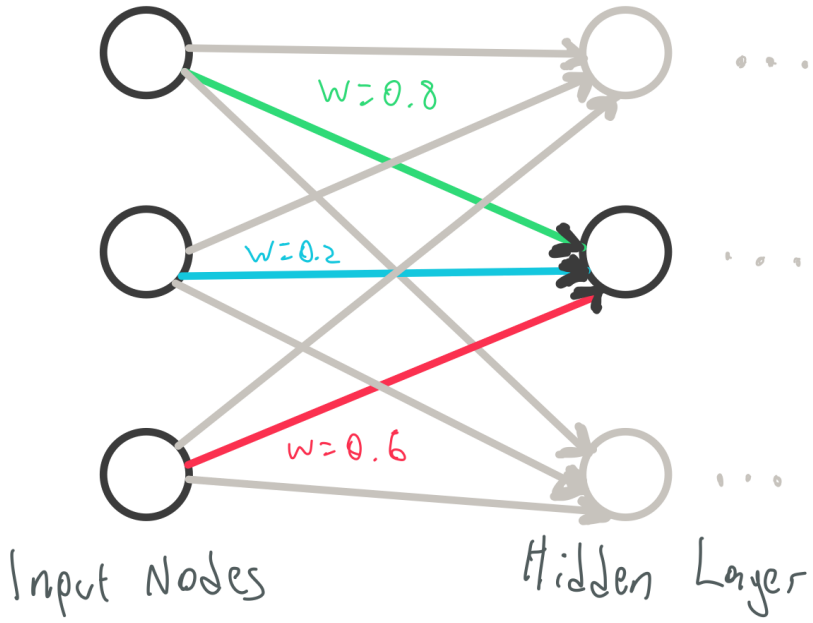


Figure 14: Weights influence illustration - bias excluded for simplicity.

When training an ANN, our goal is to find the **optimal weights** for all the network's layers so that the loss function is minimized. The training happens in batches, where each is passed through the network where on each layer the forward and backward pass are computed - updating the weights based on the loss value. When the ANN has worked through the entire training dataset (all batches) then an epoch is completed. On each epoch, the loss value, training accuracy, and validation accuracy are calculated. The ANN developed uses the Early-Stopping regularisation where the algorithm stops training if the validation accuracy does not improve for more than *patience* number of epochs. Once ANN completes the training, the classification accuracy is calculated for the testing dataset.

The ANN on the given hyperparameters (specification) has a testing accuracy of 77.6%.

4.1.2 Hyper-parameters influence

To do proper hyperparameter tuning you need to tune combinations, not one at a time. However here for evaluation purposes, I experimented with the hyperparameters **individually** to examine their effect, whilst keeping the rest of the hyperparameters fixed to the ones suggested in the specification.

Learning Rate

Figure 15: Learning rate - total running time.

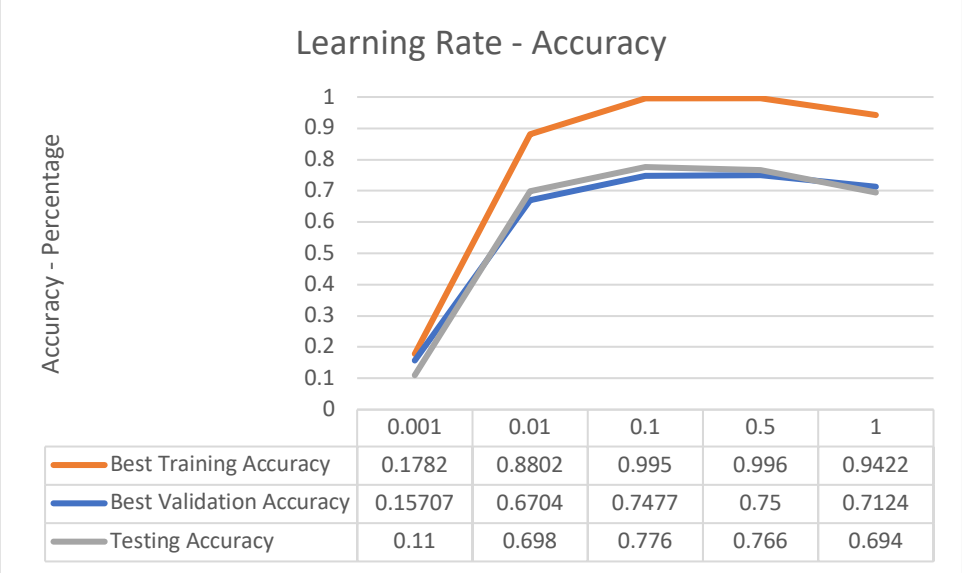


Figure 16: How Learning rate affect accuracy.

The 0.001 learning rate (lr) converged – stopped training much faster than the other lrs. This is the case due to the patience value being relatively low (10). When the lr is 0.001 the algorithm fails converges extremely slowly which leads to the ANN stopping training sooner as it did not improve

for over 10 epochs (patience). Higher learning rates seem to converge faster but as can be seen from the figure this is at the expense of jumping over some local minimum(s) that were found by the 0.1 lr. The best performing learning rate on all three sets was 0.1 and the worst 0.001.

MaxEpochs

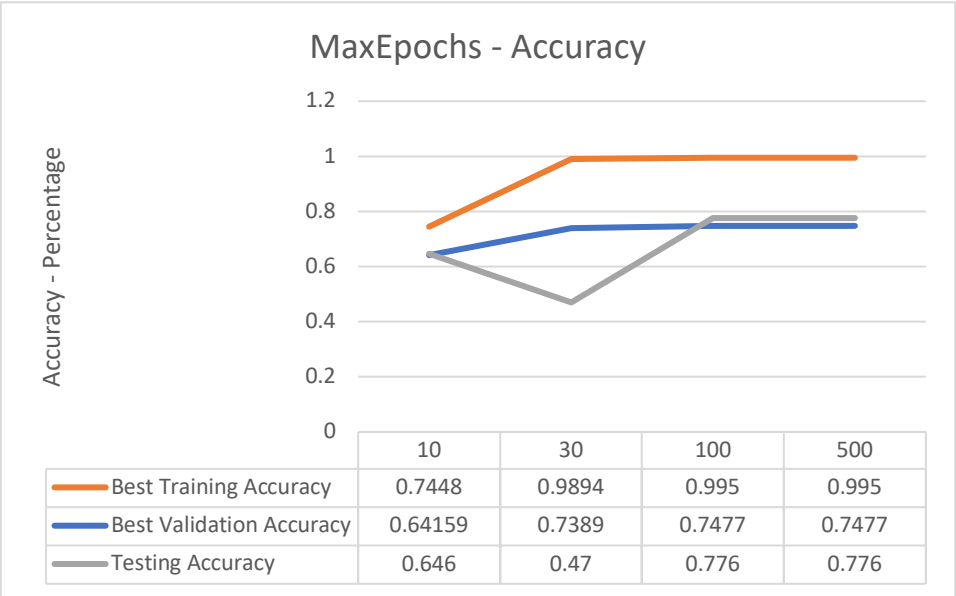


Figure 17: How maxEpochs affect accuracy.

MaxEpochs – Epoch Stopped Training	
Max Epochs	**Stopped At Epoch**
10	10
30	30
100	45
500	45

Table 7: Epoch stopped training.

Lower MaxEpochs value results in lower accuracies (all sets) as the network does not have enough time to find the optimal weights. Interestingly the results are the same for 100 and 500 MaxEpochs as the network stops training at 45 epochs due to the early stopping strategy.

Patience

Patience – Stopping Epoch & Accuracy				
Patience	**Stopped At Epoch**	**Best Training Accuracy**	**Best Validation Accuracy**	**Testing Accuracy**
2	19	0.9402	0.72124	0.722
5	22	0.9603	0.72124	0.734
10	45	0.995	0.7477	0.776
40	76	0.996	0.7478	0.766

Table 8: Patience stopping epoch and accuracy.

Having less patience causes the ANN to stop training sooner (early-stopping), whereas having more patience causes the ANN to train for a longer period. This is also reflected in small increases in accuracy for training and validation sets as the algorithm has more time to find the optimal weights. However, as evident the testing accuracy is better when patience is 10 rather than 40. This is an example of **overfitting**, where the ANN performs better in training and testing data but worse on unseen data.

Layer Dimensions

Different Dimension Accuracy Performance – Hidden Layers						
Case	**Embedding Dimensions**	**Other Dimensions**	**Training Accuracy**	**Validation Accuracy**	**Testing Accuracy**	**Training time (seconds)**
Decrease	50	100	0.9948	0.7367	0.762	86
Normal (specification)	100	200	0.995	0.7477	0.776	150
Increase	400	800	0.9946	0.7478	0.788	453

Table 9: How number of dimensions/nodes affecst performance.

There is a positive correlation between the number of dimensions/nodes in the hidden layers and the validation and testing accuracies. The ANN performs the best for testing in the "Increase" case where the first hidden layer (Embedding) has 400 dimensions, and the other hidden layers have 800 dimensions. Additional nodes provide additional weights that the network can utilise in its optimization task and as such higher accuracy can be expected. A more complex network requires more training time. There is also again the risk of overfitting if we give too many dimensions to our layers.

Number of Hidden Layers

Number of Hidden Layers	Training Accuracy	Validation Accuracy	Testing Accuracy	Training time (seconds)
1	0.9808	0.7566	0.768	86
3	0.995	0.7477	0.776	150
6	0.9918	0.7190	0.738	153

Table 10: How number of hidden layers affect performance.

The appropriate number of hidden layers depends on the complexity of the problem. Increasing the number of hidden layers more than the problem requires will cause accuracy in the test set to decrease (overfitting). When the model is too complicated it captures the relationships in them very precisely but fails to capture the trend and as such fails to capture new data.

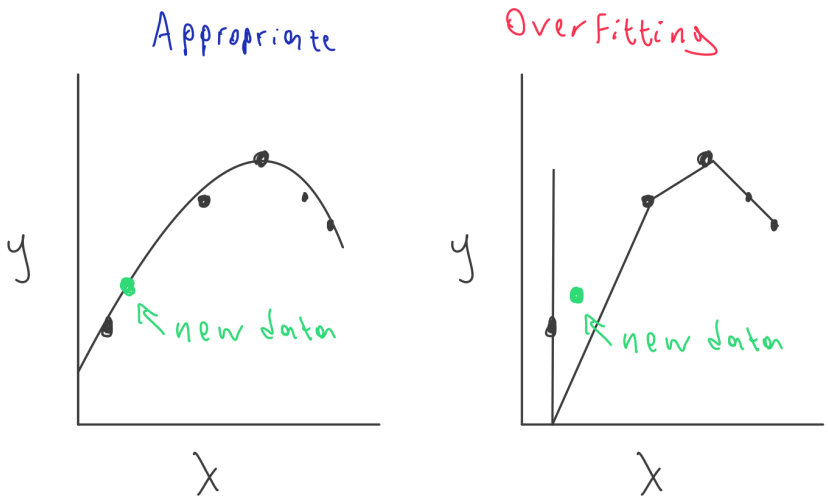


Figure 18: Overfitting visualisation.

The ANN performs the best on the testing set with 3 hidden layers. There seems to be also a positive correlation with training time.

4.2 Part 2

4.2.1 Correctness

EmbeddingBag implementation was tested manually and automatically in Section 3.

4.2.2 Performance Comparison

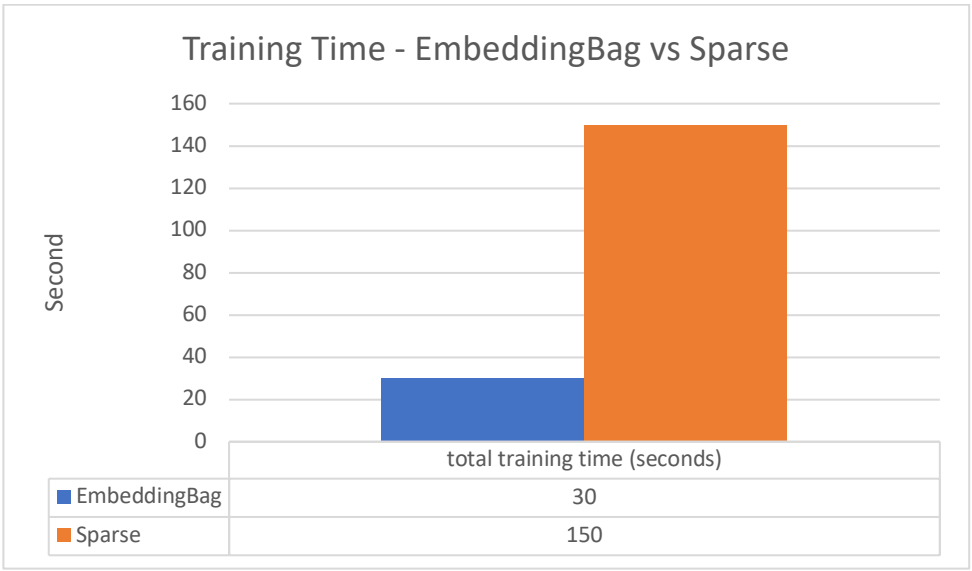


Figure 19: Training time comparison between EmbeddingBag and Sparse.

Figure 19 shows that the EmbeddingBag layer results in the network being trained much faster (5 times) than when using a Sparse Linear layer. As explained earlier, this is because the EmbeddingBag skips a lot of unnecessary calculations.

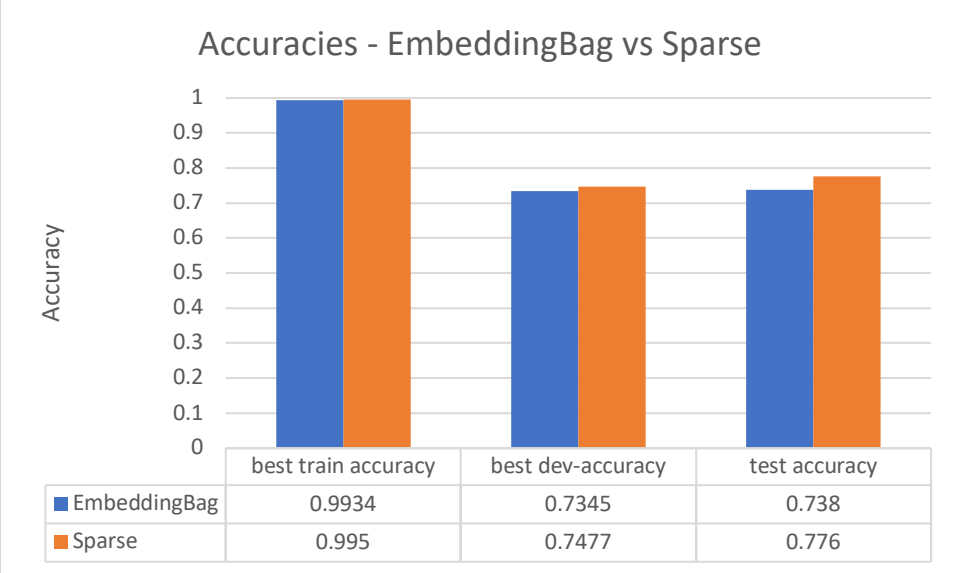


Figure 20: Accuracies comparison between EmbeddingBag and Sparse.

Interestingly there is a slight decrease in accuracy in all the tests. I believe that this is due to the random weights that are being initialised at the beginning but also because of the **bias term**. In our EmbeddingBag we omitted the inclusion of bias - leads to slight variations in the model's accuracy performance.

Part 3

Pre-trained weights vs Model in Task 2					
Model	**Training Accuracy**	**Validation Accuracy**	**Testing Accuracy**	**Training time (seconds)**	**Loss**
Pre-trained	0.9992	0.6858	0.73	66	1.7184
Part 2	0.9934	0.7345	0.738	30	6.6226

Table 11: Pre-trained weights vs task 2 model performance.

The model with pre-trained weights has a slightly higher training accuracy and lower loss value. However, the ANN developed for part 2 has higher accuracies for both the validation and testing sets. The lower loss value for the pre-trained model suggests that the distance between the true outputs and the predicted is more optimised compared to the model developed for Part 2. When combined with the lower accuracy, this might mean that the pre-trained model makes small errors in its predictions but these errors happen on more data. Perhaps other metrics such as precision and recall would give a better comparison.

Pre-trained weights are frequently used to provide an **informed initialisation** for the weights. They are also useful when not enough data is available. However, training a model from scratch often results in better accuracy as the pre-trained weights may have been trained on datasets that are too different from the one being used. We also must remember that these accuracies presented here are for untuned networks. When appropriate hyperparameter tuning takes place then the tuned pre-trained model might overperform the one for part 2.

Part 4

4.1 Freezing Embedding Layers weights

Freezing the weights leads to accelerated training; however, it often results in lower accuracy as we are only optimizing a subset of the problem, possibly missing on better optima. I chose to freeze the first hidden layer because there the features learned are more generic compared to the later layers which pick finer details of the task.

Model	Training Accuracy	Validation Accuracy	Testing Accuracy	Training time (seconds)	Loss
Pre-trained (Part 3)	0.9992	0.6858	0.73	66	1.7184
Frozen weights (Part 4)	0.8196	0.54867	0.582	48	67.9471

Table 12: Performance after freezing embedding layer's weights.

The model with the frozen weights' trains considerably faster – 48 seconds vs 66. However, it has a noticeable decrease in performance across all three sets, and worse loss value.

4.2 words2vec vs GloVe performance

Model	Training Accuracy	Validation Accuracy	Testing Accuracy	Training time (seconds)	Loss
GloVe (Part 3)	0.9992	0.6858	0.73	66	1.7184
Words2vec (part 5)	0.999	0.6969	0.74	214	1.2574

Table 13: Performance comparison using the two different pre-trained weights.

The words2vec embedding had very similar performance with GloVe. It did perform slightly better in terms of testing, validation accuracies, and loss but we must remember that the weights used by words2vec have different dimensions (300) – making it difficult to make fair comparisons (GloVe – 100 dimensions).