

1. Introduction

Assignment Number: A4

Date: 10/05/2022

Student ID: 210017984

Word Count: XXXX

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 2. word2vec word embeddings 3. 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> <seed> <trainFile> <devFile> <testFile> <vocabFile>  
<classesFile> <silent> <tune>
```

The first argument specifies the part of the exercise that you wish to run. The *seed* argument can be used to generate reproducible results (pseudo random). The *trainFile*, *devFile*, and *testFile* arguments specify the path for the training, validation, and testing datasets respectively. *VocabFile* and *classesFile* specify the location of the vocabulary and output classes files respectively.

The *silent* and *tune* flags are extensions, where *silent* does the training of the network without printing any intermediate steps, and *tune* runs the hyperparameter randomized search algorithm developed (extension).

Running Example:

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
```

2. Design & Implementation

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

PEAS? – look at lecture notes

Table 2 briefly describes 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 <i>minet.data.Dataset</i> 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 <i>VocabClassifier</i> class contains all the functions that are used to train and evaluate a Neural Network.
EmbeddingBag	Implements the Minet's <i>Layer</i> 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.

2.1 VocabDataset

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

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 <code>trainingWeights</code> flag is set to true (upon initialisation of the class) and the weights associated with each word are stored in an <code>ArrayList</code> as a double array (<i>allWeights</i>).
oneHotEncode	One hot encode a given dataset using the Bag-of-Word strategy.
createDoubleMatrixForWeights	Creates a <code>DoubleMatrix</code> that stores all the pre-trained weights passed in. Uses the <i>allWeights</i> list. The <code>DoubleMatrix</code> is used in the embedding layer when choosing to use the pre-trained weights.

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 x shows a visual example of how a sample is converted to a one-hot encoded vector.

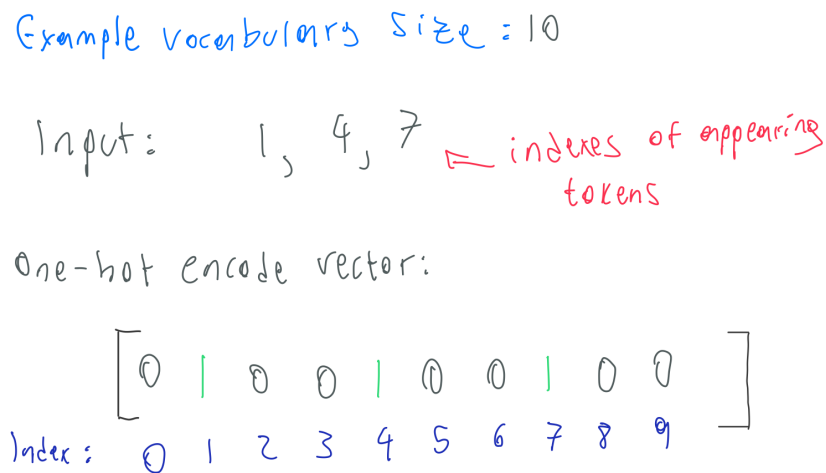


Figure 1: One-hot encoded conversion.

2.2 VocabClassifier

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

VocabClassifier Class – Most Important Methods	
Method	Description
<i>trainAndEval</i>	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.
<i>train</i>	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 <i>train</i> method uses the Early Stopping technique, which stops the training when performance on validation set stops improving – prevent overfitting.
<i>eval</i>	Evaluate the performance of the ANN during validation and testing. Uses the <i>forward</i> function to compute the predictions of the network and then the DoubleMatrix's

	<i>rowArgmaxs</i> 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.
<i>convertToDoubleMatrixPair</i>	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).

2.3 EmbeddingBag

In our Bag-of-Word approach each sample contains a binary number for all the words/tokens in the vocabulary (3249). 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. This is achieved by calculating the **output** of each node by performing 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 x depicts a simplified example of how Forward step computes the output of the first hidden layer node in a sparse vs dense matrix.

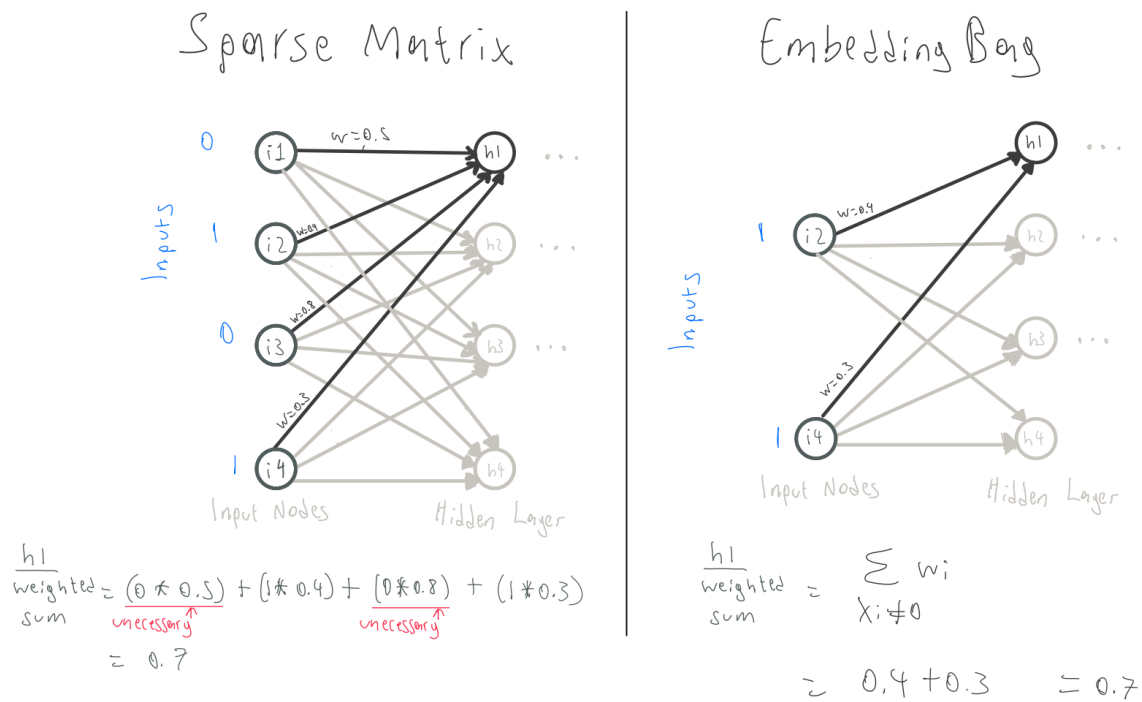
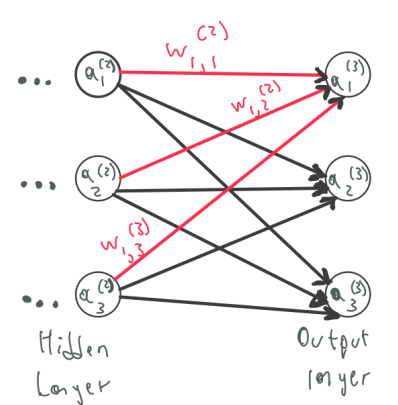


Figure 2: ForwardPass weighted sum calculation - Sparse Matrix vs EmbeddingBag

As evident, the sparse matrix method does a lot of **unnecessary** (2 out of 4 in example) calculations when computing the weighted sum. Remembering that this is a very simplified example, this problem becomes even worse, as the number of non-zero elements (value of one) are much fewer (<1%) than the vector's length in the real scenario. In contrast, the Embedding Bag method just calculates the sum of the weights where the one-hot value is one (words present in the sample), leading to a much faster and 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 is the essence of training of an ANN as it 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 last (output) layer going back to the first (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 x.

$$\begin{aligned}
 & w_{1,j}^{(2)} \\
 & \nearrow \frac{dz}{dw} \\
 & z = \left(w_{1,j}^{(2)} \right)^T a_j^{(2)} \\
 & \uparrow \frac{da}{dz} \\
 & \hat{y} = a_1^{(3)} = g(z^{(3)}) \\
 & \uparrow \frac{dL}{da} \\
 & 4 \\
 & L(w) = \sum_{i=1} y_i \log \hat{y}_i
 \end{aligned}$$


$$\frac{dL}{dw^{(2)}} = \frac{dL}{da^{(3)}} \frac{da^{(3)}}{dz^{(3)}} \frac{dz^{(3)}}{dw^{(2)}}$$

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

For part 3 we were asked to use the pre-trained word embeddings for initialising the weights. 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 Freezing Layers and Words2Vec (Extension)

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.

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. The network's layers changed to 300 hidden dimensions for the Embedding Layer and then 400 for the following hidden layers.

Hyperparameter Tuning Randomized Search TODO

3. Testing

3.1 One-hot encoding

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). In the screenshot below I demonstrate that the 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]
```

3.2 Forward Pass

The same example as the one in 3.1 is used. Here I print the weights that were randomly initialised on initialisation of the EmbeddingBag and then manually check the operation 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 3: 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 4:

Dimensions		
	0	1
Vocab Size	0	0.315616, 0.577177
	1	0.694236, 0.496510
	2	<u>-0.348896</u> , <u>0.672062</u>
	3	<u>0.153867</u> , <u>-0.61434</u>
	4	0.432565, -0.606879
	5	<u>0.530914</u> , <u>-0.578764</u>
	6	0.305539, -0.106820
	7	-0.605401, 0.529469
	8	0.418976, 0.106213
	9	0.111323, -0.146376

Figure 5:

Dimension 0

$$\begin{aligned}
 \sum_{x_i \neq 0} w_i &= (-0.348896) + 0.153867 + 0.530914 \\
 &= 0.335885 \quad (6 \text{ d.p.}) \quad \checkmark \text{ correct}
 \end{aligned}$$

Dimension 1

$$\begin{aligned}
 \sum_{x_i \neq 0} w_i &= 0.672062 + (-0.61434) + (-0.578764) \\
 &= -0.521042 \quad (6 \text{ d.p.}) \quad \checkmark \text{ correct}
 \end{aligned}$$

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 testClassificationForEmbedding() {
    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);
}
}
```

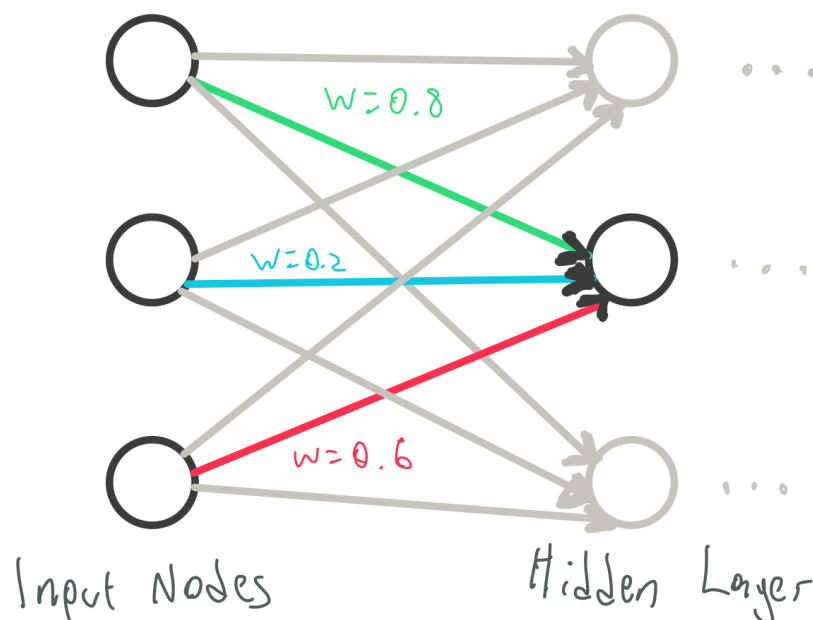
correct backward for weights

4. Evaluation

4.1 Part 1

4.1.1 Training Process and Performance on Test Data

An ANN is made up of multiple neurons that are connected by links. Each of these links has a weight value associated with it, which can be thought of as the connection's strength, affecting 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). Please note for simplicity this example excludes the bias.



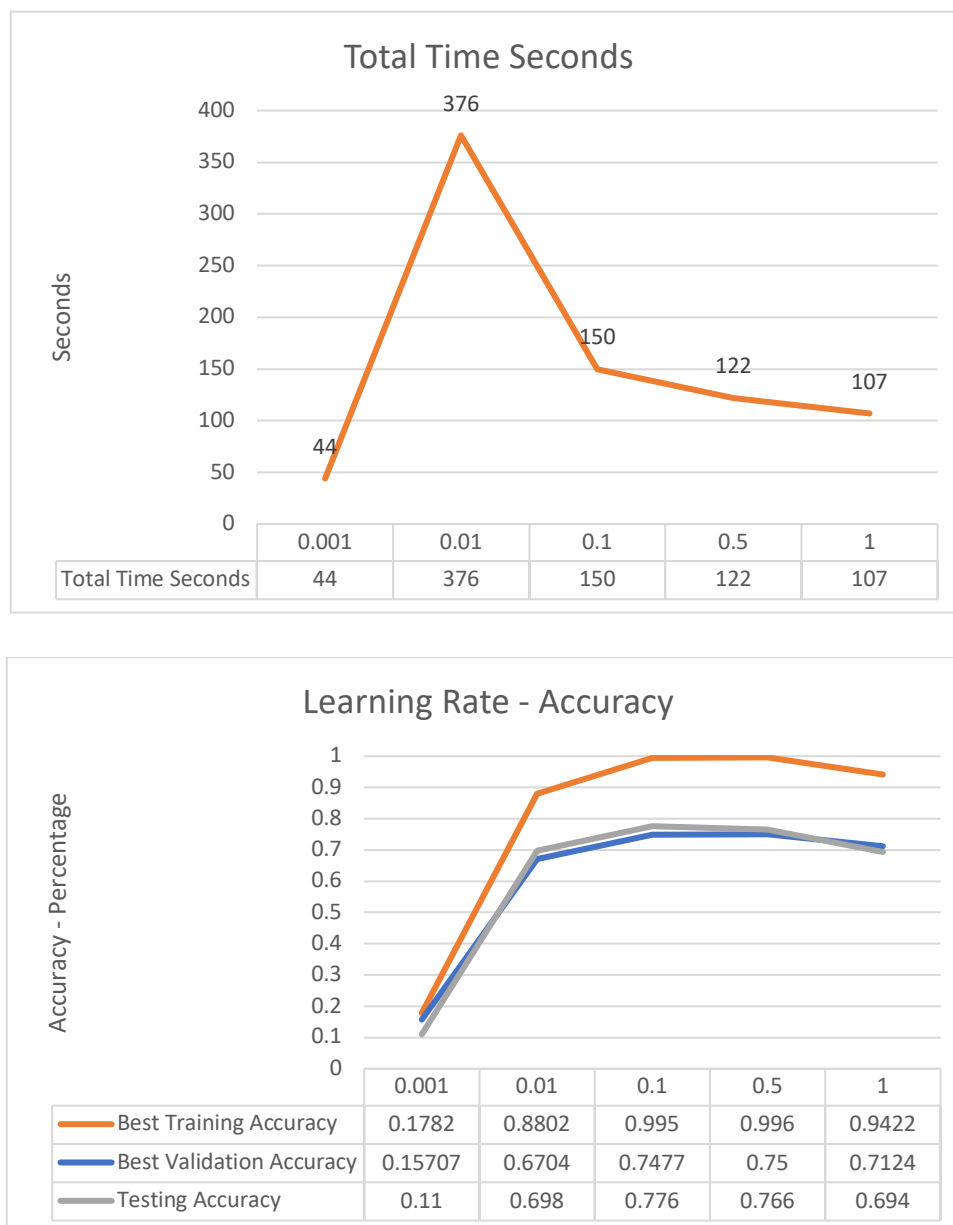
When training an ANN, our goal is to find the most **optimal weights** for all the network's layers so that the loss function (Cross Entropy for our task) is minimized. The training happens in batches. Each batch is passed through the network where on each layer the forward and backward pass are computed - 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 criteria where the algorithm stops training if the validation accuracy does not improve for more than *patience* number of epochs. Once the ANN is completely trained 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 hyperparameter tuning properly you need to tune them all together at the same time (not one at a time). However here for evaluation purposes, I experimented with the hyperparameters **systematically** to examine the effect of each whilst keeping the rest of the hyperparameters fixed to the ones that were suggested in the specification.

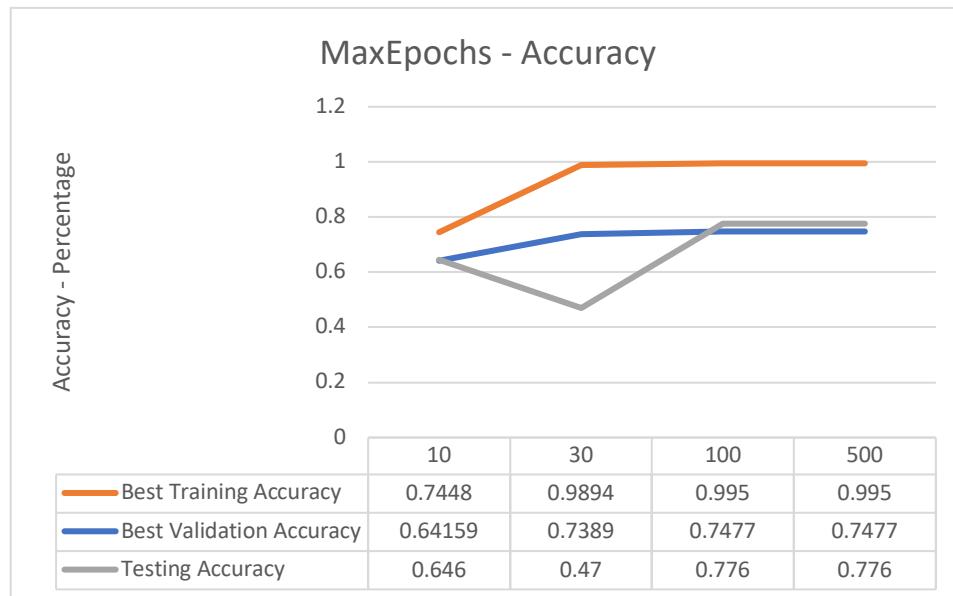
Learning Rate



The 0.001 learning rate (lr) converged – stopped training much faster than the other lrs. This might seem unorthodox at first glance; however, this is the case due to the patience value being relatively low (10). When the lr is 0.001 the algorithm fails to converge (or 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 accuracies figure this is at the expense of jumping over some local minimum(s) that were found by the 0.1 lr.

There is therefore a trade-off between training time and accuracy. The best performing learning rate on all three sets was 0.1 and the worst 0.001.

MaxEpochs



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

The MaxEpochs hyperparameter specifies the maximum number of epochs to occur during training. As evident from the Figure and Table above, 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

The patience value influences the Early Stopping strategy which stops training if the validation accuracy for Patience number of times does not improve. As evident, having less patience causes the ANN to stop training sooner, 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 early signs of **overfitting**, where the ANN performs better in training and testing data but fails to generalise better on unseen data.

Layer Dimensions

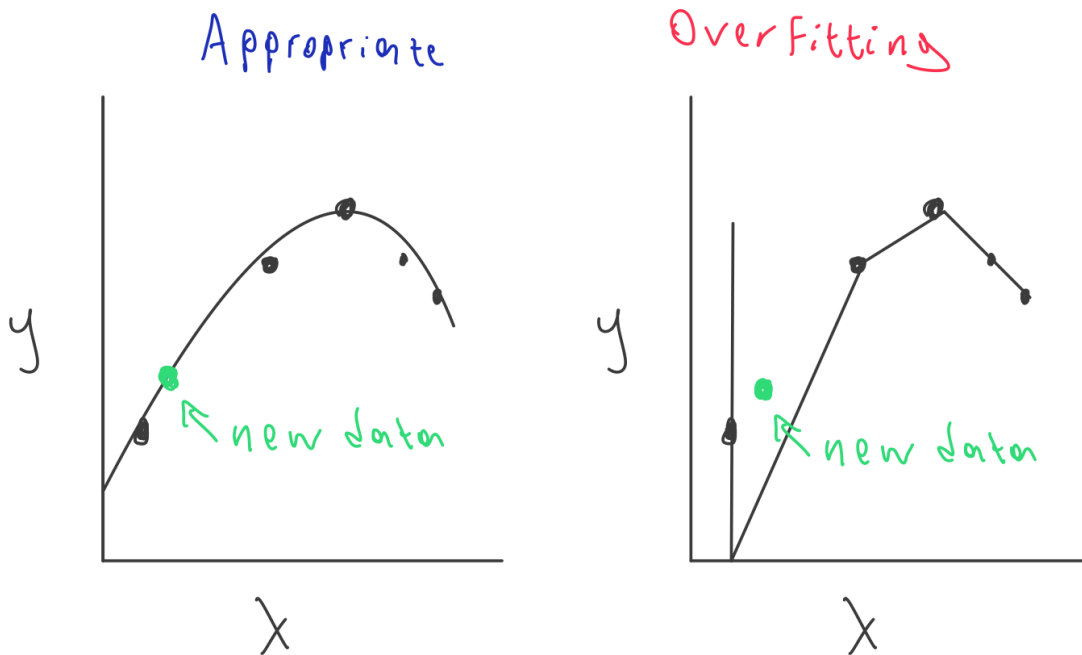
Different Dimension Accuracy Performance – Hidden Layers						
Case	Embedding Dimensions	Other Dimensions	Training Accuracy	Validation Accuracy	Testing Accuracy	Training time (seconds)
Decrease	50	100	0.9948	0.7367	0.762	86
Normal (specification)	100	200	0.995	0.7477	0.776	150
Increase	400	800	0.9946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

As evident 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 (generalises better) in the “Increase” case where the first hidden layer (Embedding) has 400 dimensions, and the other hidden layers have 800 dimensions. Additional nodes provide excess capacity – additional weights that the network can utilise in its optimization task and as such higher accuracy can be expected. Of course, a more complex network requires more training time as evident from the training time (Table x). 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

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 due to overfitting. Figure x shows this – 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.



The ANN performs the best on the testing set with 3 hidden layers, followed by 1 and then by 6. There seems to be also a weak correlation with training time \rightarrow more layer, more training time.

BATCH SIZE

4.2 Part 2

4.2.1 Correctness

I ensured my EmbeddingBag implementation is correct using manual and automatic testing as described in Section 3.

4.2.2 Performance Comparison

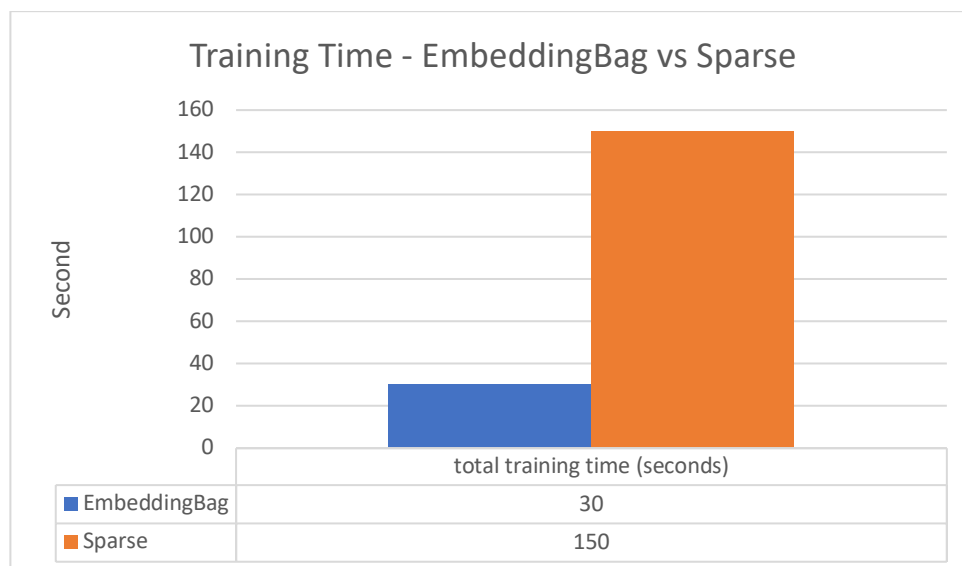


Figure x shows that the EmbeddingBag layer results in the network being trained much faster (5 times) than when using a Sparse Linear layer. As explained earlier, this is because the EmbeddingBag skips a lot of unnecessary calculations.

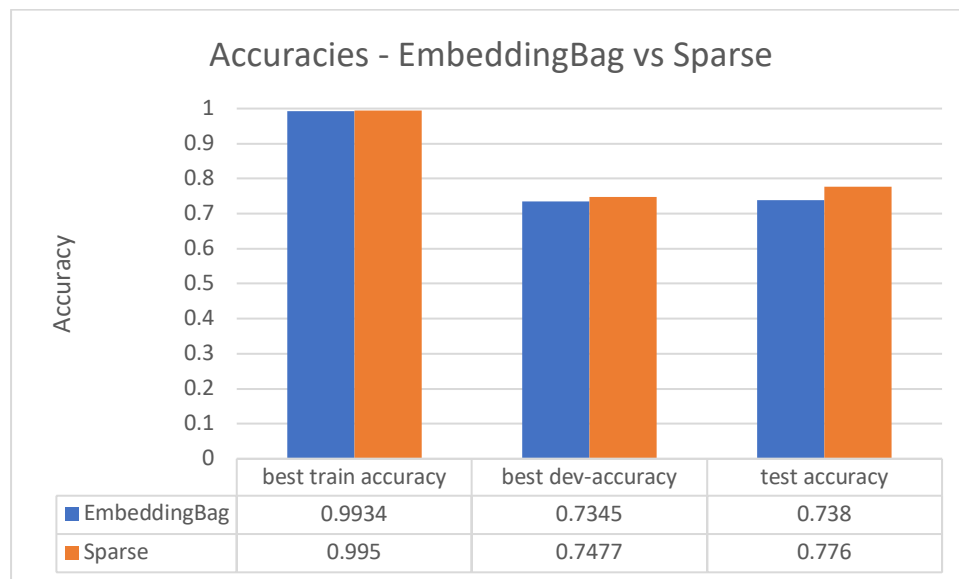


Figure 6:

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, this leads to slight variations in the model's accuracy performance.

Part 3

The pre-trained weights model is compared to the model developed for Task 2 as it has the same network architecture.

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

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 clearer comparison between the two models performance.

Pre-trained weights are frequently used to provide a **good/informed initialisation** for the weights. They are also extremely useful when not enough data is available. However, training a

model from scratch often results in better accuracy because 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 (using GridSearch) then the tuned pre-trained model might overperform the one for part 2.

Part 4

4.1 Freezing Embedding Layers weights

In part 4 the EmbeddingBag's weights were frozen – not updated at backpropagation. This technique is used to accelerate 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 are very tuned to the 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

When compared with the pre-trained model used in Part 3, the model with the frozen weights' trains considerably faster – 48 seconds compared to 66. However, it has a noticeable decrease in performance across all three sets, and worse loss value as evident in Table X.

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

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).