CS5011 Assignment 4: Learning

Student ID: 210034146

(Dated: May 11, 2022)

1: INTRODUCTION

1.1: Sections implemented

The following list describes the sections implemented in this assignment:

- Part 1 (sparse ANN): Attempted and fully working.
- Part 2 (ANN with randomised embedding layer): Attempted and fully working.
- Part 3 (ANN with GloVe embedding layer): Attempted and fully working.
- Part 4 (ANN with frozen GloVe embedding layer): Attempted and fully working.
- Part 5 (ANN with word2vec embedding layer): Attempted and fully working.
- Part 6 (ANN with frozen word2vec embedding layer): Attempted and fully working.

2: DESIGN AND IMPLEMENTATION

2.1: Brief introduction to Artificial neural networks

Supervised machine learning (ML) problems typically involve searching for some unknown function $f_\phi : \mathbf{X} \to \mathbf{Y}$, where $\mathbf{X}$ and $\mathbf{Y}$ are the inputs (observed data), and the target outputs respectively, here ϕ represents the parameters of the function f_ϕ. The parameters are learned by minimising the loss function $L(\mathbf{y}, f_\phi(\mathbf{x}))$ which describes how far the predictions are from the true values. Two commonly used loss functions are:

$$\text{Mean Square Error (MSE)} = \frac{1}{N}\sum_{i=1}^{N}(y_i - f_\phi(\mathbf{x}_i))^2 \quad (1)$$

$$\text{Cross entropy loss (CE)} = -\frac{1}{N}\sum_{i=1}^{N}(y_i log(f_\phi(\mathbf{x}_i))) \quad (2)$$

Artificial neural networks can be described as a series of transformations from the input state ($\mathbf{x}$) to the output state ($\mathbf{y}$) with a few intermediate states in-between ($\mathbf{h}_i$). The transformation from one layer to the next is described by equation 3.

$$\mathbf{h}_{i+1} = g_i(W_i\mathbf{h}_i + b_i) \quad (3)$$

Here, the i denotes the layer number, g_i is the activation function, and b_i is the bias vector . Each transformation is guided by a weight matrix W_i. The weights are randomised at first, and the network is then trained (the weights are tuned) in such a way that reduces the loss function. The learning process in done in two stages.

- 1) Forward pass: Series of transformations till the output layer (equation 3).
- 2) Backward pass: This stage involves calculating the gradient of the loss function with respect to the weights $\nabla_\phi L(\mathbf{y}, f_\phi(\mathbf{x}))$. The negative of the gradient calculated here points in the direction (within the parameter space) that reduces the loss. Therefore, the weights are updated in the direction proportional to the negative of their gradient w.r.t the loss function.

A cycle of training through all samples is known as an epoch. This training process is repeated until the network reaches a particular loss value (tolerance), or the process stops after a set number of epochs.

2.2: Main Design & Implementation

The problem at hand involves creating a neural network using the MINET package, which can classify questions (input) into 50 distinct question types (categorical output). The MNIST example provided by MINET is used as a template to create such a neural network since both networks are intended for classification tasks, albeit both networks are of different sizes. The input data consists of sentences described as a list of indices of words from a given vocabulary, and the output data are labels, which describe which class a particular question belongs to. This practical is concerned with the bag of word approach where the questions are fed as an input to the network, but the order of the words is not taken into account. The following sections describe the implementation of the different parts of this practical.

2.2.1: Part 1

The input data (questions) contains sentences with different numbers of words. Neural networks cannot process inputs with variable sizes. For this reason, part 1 of this practical works by converting each sentence (sample) into a binary fixed-sized vector where the length of the vector equals the vocabulary size of the training set (3249 in this case). For each vector, the values are set to 1 for the corresponding word index and 0 for the rest. For example, if the list *indices* contains the indices of words for a sample sentence, then the transformed input *xs* can be obtained in the following way (see listing 1).

```
// filling the vector with zeros
for (j = 0; j < vocab.size; j++){xs[j] = 0}
// indices which correspond to the words in the
    sentence are set to 1.
for (index in indices) {
    xs[index] = 1
    }
```

Listing 1: Bag of words / one hot encoding process.

This transformation of the input sentence to a fixed size vector allows the neural network to work with the input data. Listing 2 describes the architecture of this neural network.

```
new Linear(3249, 100),
new ReLU(),
new Linear(100, 200),
new ReLU(),
new Linear(200, 200),
new ReLU(),
new Linear(200, 50),
new Softmax()};
```

Listing 2: Neural network architecture part 1.

2.2.2: Part 2

As described before by equation 3, the transformation from one layer to the next involves a matrix multiplication of the weights with the input of the previous layer ($W_i\mathbf{h}_i$). However, in this case, the bag of words approach of converting each sentence into a vector of length a fixed length is computationally inefficient due to the sparse nature of the fixed-size vector ($\#0 \gg \#1$). To circumvent this, the first hidden layer of the neural network in part 1 is replaced by an embedding layer. The embedding layer skips the costly sparse matrix multiplication in the forward and backward pass by making use of the following equations, where $z = W_i\mathbf{x}$, and $z = [z_1, z_2 ....z_n]$

Forward pass:

$$z_j = \sum_{x_i \neq 0} w_{ij} \tag{4}$$

Backward pass:

$$\frac{\partial z_j}{\partial w_{ij}} = 1 \, \forall x_i \neq 0 \tag{5}$$

$$\frac{\partial z_j}{\partial w_{ij}} = 0 \, \forall x_i = 0 \tag{6}$$

For a weight matrix W, listing 3 describes the forward pass in the embedding layer.

```
DoubleMatrix Y = DoubleMatrix.zeros(X.length,
    outdims);
for (int i = 0; i < X.length; i++){
    DoubleMatrix node = DoubleMatrix.zeros(outdims)
    ;
    // list of indices for each word
    double[] sentence = X[i];
    for (int j = 0; j < sentence.length; j++) {
        // adding only the relevant weights
        node.addi(W.getRow((int) (sentence[j])));
    }
    Y.putRow(i,node);
}
```

Listing 3: Forward pass.

For a gradient matrix gW, listing 4 describes the backward pass in the embedding layer.

```
for (int i = 0; i <X.length; i++){
    double[] indices = X[i];
    for (double index:indices){
        int indexx = (int) index;
        gW.putRow(indexx,gW.getRow(indexx).add(gY.
    getRow(i)));
    }
}
```

Listing 4: Backward pass.

The network architecture in part 2 is identical to part 1, except that the input word indices are directly read into the embedding layer; this can be seen in listing 5.

```
new EmbeddingBag(3249, 100),
new ReLU(),
new Linear(100, 200),
new ReLU(),
new Linear(200, 200),
new ReLU(),
new Linear(200, 50),
new Softmax()};
```

Listing 5: Neural network architecture part 2.

Although the input size is not 3249 anymore due to skipping the creation of the one-hot encoded vector, the value 3249 is needed to generate the weight matrix in the embedding layer, which has the dimensions of $vocabulary_{(size)} \times 100$ i.e. (3249×100).

2.2.3: Part 3

For parts 1 & 2, the weights in every layer were initialised randomly. In part 3, the weights of the embedding layer are imported from the provided vocab.txt file (pre-trained word embeddings from GloVe), and the weights for the rest of the layers are still initialised randomly.

The use of pre-trained GloVe word embeddings is a form of transfer learning. The use of pretrained weights allows for an informed initialization of the problem. Pre-trained Word embeddings capture the semantics of each word in the data set, which is represented by a vector. The dot product between two such vectors represents how close these words are in a higher dimensional semantic space. For example, consider equations 7 and 8, where the embedding vector for word i is represented as $\vec{v_i}$.

$$\vec{v_{king}} - \vec{v_{queen}} + \vec{v_{actress}} \approx \vec{v_{actor}} \tag{7}$$

$$v_{\vec{Paris}} - v_{\vec{France}} + v_{\vec{UK}} \approx v_{\vec{London}} \quad (8)$$

Listing 6 describes how the pre-trained word vectors were loaded to the embeddeding layer.

```java
double[][] embedding_vectors = new double[][];
BufferedReader br = new BufferedReader(new
    FileReader(path));
String[] ss;
String line;
int count = 0;
while ( (line = br.readLine()) != null ) {
    // reading through each line
    ss = line.split(" ");
    // empty vector
    ArrayList<Double> Xs = new ArrayList<Double>();
    for (int j = 1; j < ss.length; j++) {
        // adding elements to the vector
        Xs.add(Double.parseDouble(ss[j]));
    }

    double[] xs = new double[Xs.size()];
    xs = Xs.stream().mapToDouble(d -> d).toArray();
    // setting vector for ith word
    embedding_vectors[count] = xs;
    count += 1;
}
W = new DoubleMatrix(embedding_vectors);
```

Listing 6: Reading weights from the vocab.txt file.

2.2.4: Part 4

The fourth part of this practical involves training the networks using the same word embeddings as before (vocab.txt); however, in this part, the embedding layer is frozen (**i.e.** none of the weights in the embedding layer are updated). The weights are usually updated by adding the gradient weight matrix (gW) to the current weight matrix in the embedding layer. The weights are frozen in the embedding layer by filling gW with zeros (i.e. $W + gW = W$). Listing 7 describes how the weights in the embedding layer were frozen.

```java
public DoubleMatrix backward(DoubleMatrix gY) {
    if (part.equals("part4"){gW.fill(0);}
    return null;
}
```

Listing 7: Freezing weights in the embedding layer

2.2.5: Part 5

Part 5 involves using word2vec embedding vectors for training the network. A python script was used to obtain the embedding vector given the word (see listing 8). However, it is important to note that each word embedding vector in word2vec has a length of 300, unlike the GloVe vectors, which have a length of 100. This changes the architecture of the neural network since now the embedding layer has 300 neurons (an increase of 200 from parts 1,2,3 and 4). The new architecture is described by listing 9.

```python
import spacy
# Load the spacy model that you have installed
nlp = spacy.load('en_core_web_md')
# process a sentence using the model
word = "any_word"
vec = nlp(word).vector
```

Listing 8: Python script to obtain word2vec word. embedding vectors

```java
new EmbeddingBag(8594, 300),
new ReLU(),
new Linear(300, 200),
new ReLU(),
new Linear(200, 200),
new ReLU(),
new Linear(200, 50),
new Softmax()};
```

Listing 9: Neural network architecture part 5.

2.2.6: Part 6

Part 6 is identical to part 4, but here the word2vec weights are frozen in the embedding layer.

3: EVALUATION

3.1: Part 1

The effect of batch size and learning rate on the train/test accuracy was investigated to explore the space of hyper-parameters. Figures 1 and 2 depict the accuracy and the training time as a function of the learning rate, respectively.

A learning rate of $\alpha = 0.01$ results in a longer training time due to a slower descent at each backward pass and has relatively lower accuracy (figure 1) since it cannot recover (exit) from a local minimum. Slightly larger learning rates of $\alpha = 0.1$ and 0.5 train faster and perform better since the update step is large enough to pass through a local minimum or saddle. Learning of $\alpha = 1.0$ trains even faster but loses accuracy since this 'large' update of the weights using the gradient skips the optimal minima altogether (**i.e** update step is too large to converge). Figures 3 and 4 depict the accuracy and the training time as a function of the batch size, respectively. There's an inverse correlation between the batch size and the training time. smaller batch sizes then to be more accurate.

3.2: Part 2

As discussed before, part 2 circumvents the inefficient sparse matrix multiplications of part 1. If implemented correctly, part 2 should perform similarly to part 1 (similar performance due to randomised weight initialization), and part

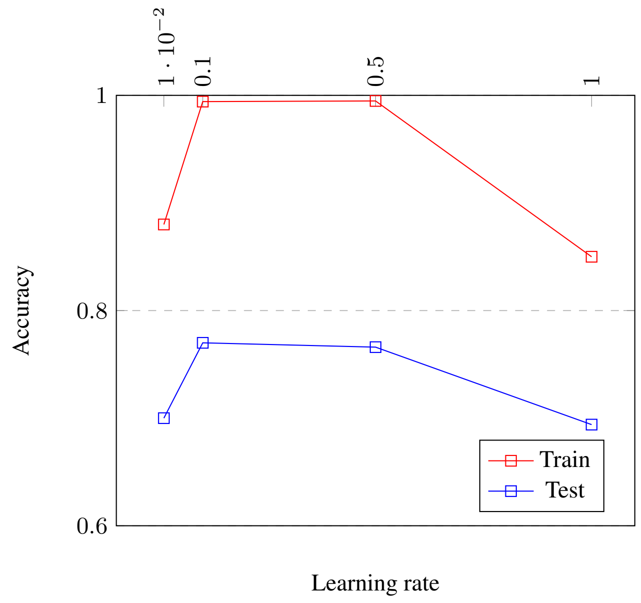

FIG. 1: Accuracy vs learning rate

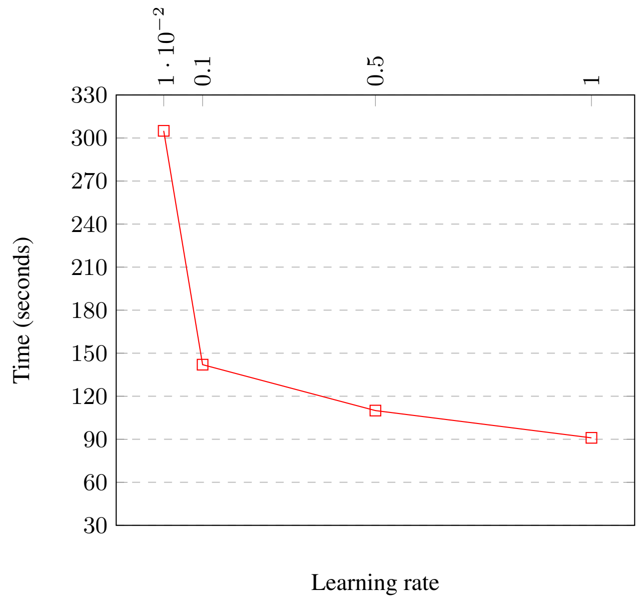

FIG. 2: Train time vs learning rate

2 should be significantly faster than part 1. Figures 5 and 6 show a side-by-side comparison of the accuracy and training time for both networks. As seen below, the performance of both neural networks is roughly identical, and part 2 trains 6.8 times faster than part 1. Furthermore, the gradient checker was used to check if the backward pass in part 2 was implemented correctly.

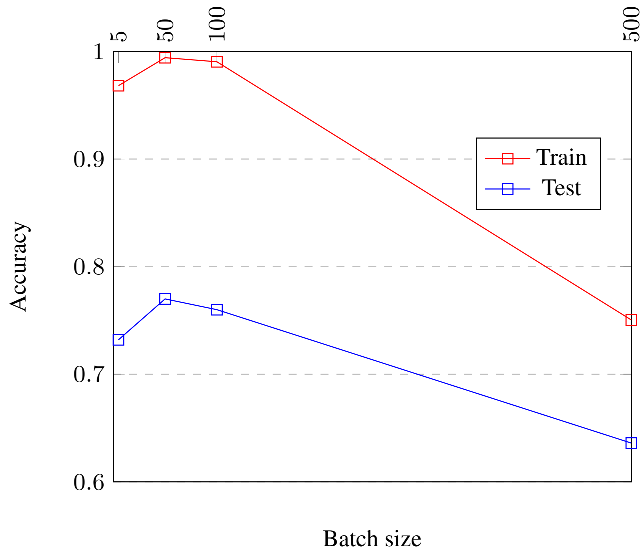

FIG. 3: Accuracy vs Batch size

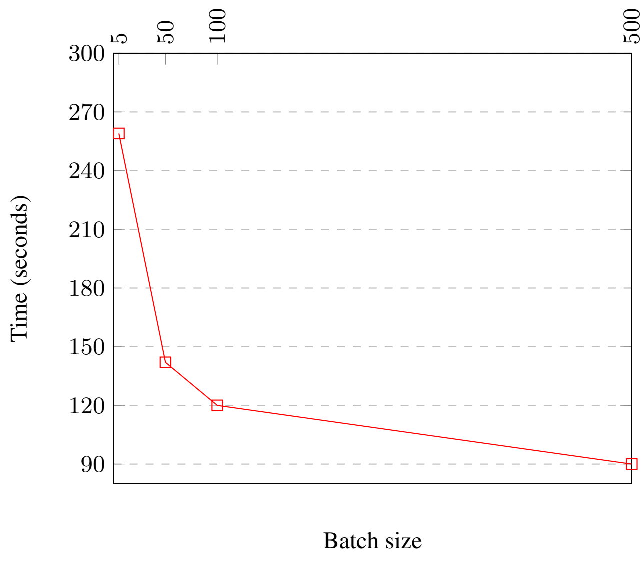

FIG. 4: Train time vs Batch size

3.3: Part 3

Part 3 of this practical took 35 seconds to train, with a training and testing accuracy of 0.9982 and 0.728, respectively. The difference in performance between part 3 and part 1 & 2 is due to the difference in weight initialisation. For parts 1 and 2, the weights were generated randomly and uniformly, whereas for part 3, the weights in the embedding layer were initialized by the GloVe vectors. This can be thought of as initializing the loss function with different initial conditions (the weights), ultimately resulting in part 3 converging to a different minima than parts 1 and 2. Furthermore, both part 1 and 2 converge to a loss value of $\mathcal{O}(20)$, whereas the GLoVe weights in part 3 converge to the loss value of $\mathcal{O}(5)$; this anomaly can be further

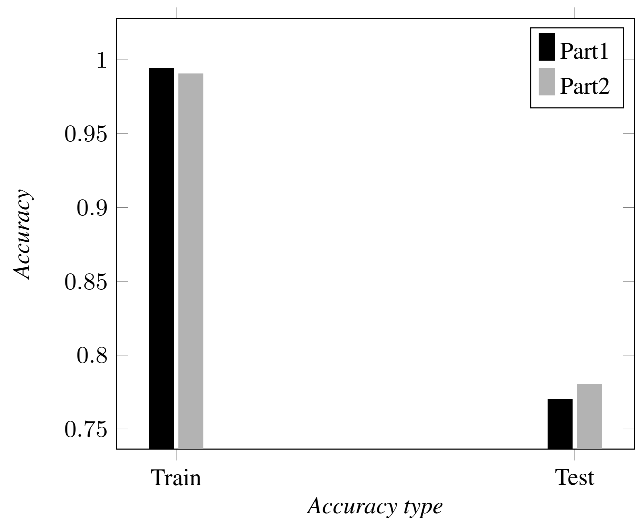

FIG. 5: Accuracy: part 1 vs part 2

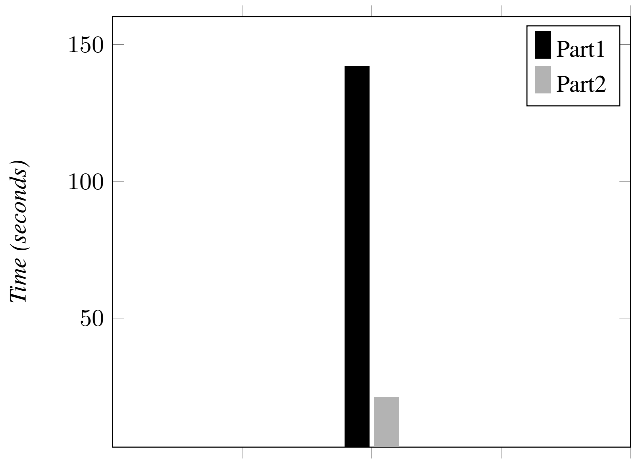

FIG. 6: Training time: part 1 vs part 2

investigated by considering the precision and recall scores.

3.4: Part 4

Part 4 of this practical took 28 seconds to train, with a training and testing accuracy of 0.7962 and 0.586, respectively. The difference in performance between part 4 and part 3 is due to the weights in the embedding layer being frozen (fewer parameters to train/learn).

3.5 Part 5

Part 5 of this practical took 49 seconds to train, with a training and testing accuracy of 0.998 and 0.822, respectively. The difference in performance between part 5 and the rest of the parts is due to the larger network size (**i.e.** the embedding layer has 300 neurons instead of 100). Furthermore, part 5 uses the word2vec embeddings instead of GloVe.

3.6 Part 6

Part 6 of this practical took 40 seconds to train, with a training and testing accuracy of 0.9948 and 0.73, respectively. The difference in performance between part 6 and part 5 is due to the weights in the embedding layer being frozen (fewer parameters to train/learn).

4: CONCLUSION

To conclude, ANNs can classify different questions into different categories by using the bag of words approach. The performance of the network depends on various hyperparameters such as batch size and learning rate. Furthermore, larger networks (such as the one which accompanied the word2vec embedding) can capture more information about the problem space, allowing it to perform better on unseen data.