COMP0090 Intro to Deep Learning
Assignment 2 - Team RealDeep

Toby Drane
ucabtdr@ucl.ac.uk

Ravi Patel
ucabrp1@ucl.ac.uk

Udi Ibgui
zcemyib@ucl.ac.uk

Louis Prosser
ucabljp@ucl.ac.uk

Daniel May
ucabdd3@ucl.ac.uk

December 10, 2020

Python programming language was used for all code related to this assignment. The full code for each question can be found in the Appendices. All models were implemented using PyTorch machine learning framework.

- Udi Ibgui took primary responsibility for question 1.
- Toby Drane took primary responsibility for question 2.
- Ravi Patel took primary responsibility for question 3.
- Louis Prosser took primary responsibility for question 4.
- Daniel May took primary responsibility for question 5.

Contents

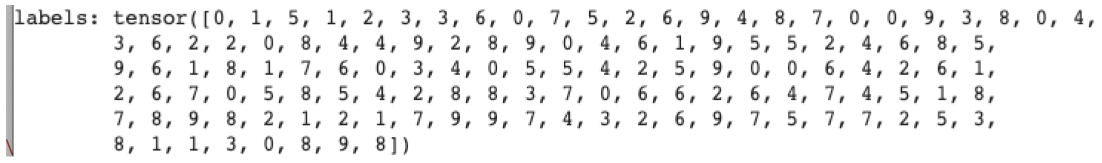
1 Q1: Multiclass MLP

1.1 Part 1: Model Implementation

The Data All Python code corresponding to all aspects of Q1 can be found in Appendix A. We only include a few key relevant snippets in the current section. The code is commented to make it easy to follow and to identify relevant sections.

The Fashion-MNIST dataset is easily accessed within the Pytorch framework. The data was first downloaded, transformed into tensors, and split into training, validation and test sets. By default, the dataset contains 60,000 training images and 10,000 test images. The training and validation sets were split with an 80/20 ratio. The datasets were sent to dataloaders, with batch sizes of 128.

To better understand the dataset, a batch was visualised, showing the input images and their corresponding labels.



```
labels: tensor([0, 1, 5, 1, 2, 3, 3, 6, 0, 7, 5, 2, 6, 9, 4, 8, 7, 0, 0, 9, 3, 8, 0, 4,
        3, 6, 2, 2, 0, 8, 4, 4, 9, 2, 8, 9, 0, 4, 6, 1, 9, 5, 5, 2, 4, 6, 8, 5,
        9, 6, 1, 8, 1, 7, 6, 0, 3, 4, 0, 5, 5, 4, 2, 5, 9, 0, 0, 6, 4, 2, 6, 1,
        2, 6, 7, 0, 5, 8, 5, 4, 2, 8, 8, 3, 7, 0, 6, 6, 2, 6, 4, 7, 4, 5, 1, 8,
        7, 8, 9, 8, 2, 1, 2, 1, 7, 9, 9, 7, 4, 3, 2, 6, 9, 7, 5, 7, 7, 2, 5, 3,
        8, 1, 1, 3, 0, 8, 9, 8])
```

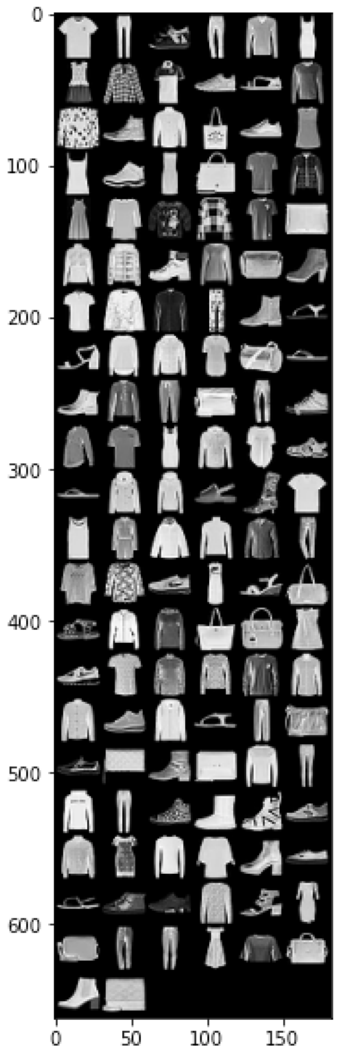
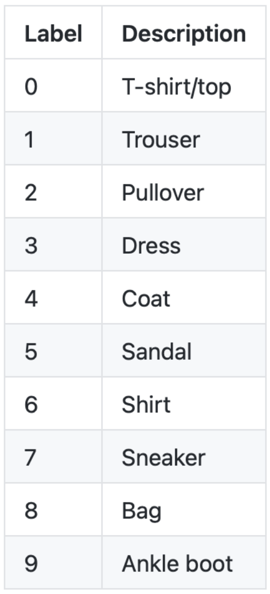


Figure 1: A batch of images with labels

We used numerical categorical labels for the images rather than their English names. Numerical labels ensure the dataset can be used universally, no matter the user's language. The article name corresponding to each number is as shown below.



Label	Description
0	T-shirt/top
1	Trouser
2	Pullover
3	Dress
4	Coat
5	Sandal
6	Shirt
7	Sneaker
8	Bag
9	Ankle boot

The size and balance of the training dataset was also confirmed, using the code below.

```
print('Training dataset size:', len(train_set))
print(train_set.targets.bincount())

Training dataset size: 60000
tensor([6000, 6000, 6000, 6000, 6000, 6000, 6000, 6000, 6000, 6000])
```

We can see that our training dataset contains 60,000 images, and equal amounts of each class. Hence, it is balanced, avoiding class imbalance issues.

The Model Implementation

The images were first flattened, before passing through the MLP layers. Each layer contained nodes which transformed the inputs and activated the next layer. The output layer contained 10 nodes, corresponding to the 10 possible labels. The most activated node in this final layer corresponded to the model's label prediction.

As will be further discussed in subsection 2, different architectures of MLP were implemented and tested as part of hyperparameter search. We used cross-entropy loss as specified in the coursework question, and Adam optimiser. The following MLP PyTorch class was used to implement the MLPs, and the code is written to be adaptive to specified architectural variants during hyperparameter search:

```
class VariableNN(nn.Module):

    def __init__(self, num_of_layers, Dropout, N, O, P):
        super().__init__()
        #Decide number of layers
        self.num_of_layers = num_of_layers
        #Dropout toggle and function, use standard p = 0.7
        self.Dropout = Dropout
        self.dropout = nn.Dropout(p=0.7)
        #Node elements
        self.N = N
        self.O = O
        self.P = P
        #Fully connected layers
        self.fc1 = nn.Linear(28*28, N)
        torch.nn.init.xavier_normal_(self.fc1.weight)
        torch.nn.init.zeros_(self.fc1.bias)

        self.fc2 = nn.Linear(N, O)
        torch.nn.init.xavier_normal_(self.fc2.weight)
        torch.nn.init.zeros_(self.fc2.bias)

        self.fc3a = nn.Linear(O, P)
        torch.nn.init.xavier_normal_(self.fc3a.weight)
        torch.nn.init.zeros_(self.fc3a.bias)

        self.fc3b = nn.Linear(O, 10)
        torch.nn.init.xavier_normal_(self.fc3b.weight)
        torch.nn.init.zeros_(self.fc3b.bias)

        self.fc4 = nn.Linear(P, 10)
        torch.nn.init.xavier_normal_(self.fc4.weight)
        torch.nn.init.zeros_(self.fc4.bias)

    #Define Forwards Pass
    def forward(self, x):
        if self.Dropout == 0:
            if self.num_of_layers == 4:
                # flatten image input
                x = x.view(-1,28*28)
                x = self.fc1(x)
                x = F.relu(x)
                x = self.fc2(x)
                x = F.relu(x)
                x = self.fc3a(x)
                x = F.relu(x)
                x = self.fc4(x)

            elif self.num_of_layers == 3:
                # flatten image input
                x = x.view(-1,28*28)
                x = self.fc1(x)
                x = F.relu(x)
                x = self.fc2(x)
                x = F.relu(x)
                x = self.fc3b(x)

        elif self.Dropout == 1:
            if self.num_of_layers == 4:
                # flatten image input
                x = x.view(-1,28*28)
                x = self.fc1(x)
                x = F.relu(x)
                x = self.dropout(x)
                x = self.fc2(x)
                x = F.relu(x)
                x = self.dropout(x)
                x = self.fc3a(x)
                x = F.relu(x)
                x = self.dropout(x)
                x = self.fc4(x)

            elif self.num_of_layers == 3:
                # flatten image input
                x = x.view(-1,28*28)
                x = self.fc1(x)
                x = F.relu(x)
                x = self.dropout(x)
                x = self.fc2(x)
                x = F.relu(x)
                x = self.dropout(x)
                x = self.fc3b(x)

        return x
```

1.2 Part 2: Hyperparameter search

An MLP was created to identify each article of clothing. To optimise the model we searched the space of hyperparameters, including trying different architectures for the MLP. In particular, different MLP depths and widths were tested, as well as learning rates and dropouts. An MLP PyTorch class containing if statements to adapt to architectural hyperparameters (depth, number of nodes in each layer and dropout setting) was built, and hyperparameter combinations were then looped through. The full code is available in Appendix A, and the results for the different architectures can be seen in Table 2.

In terms of the optimiser and learning rate used, two variants of Adam were tested, one using a learning rate of 0.001 and the other 0.01. During training it was observed that the lower learning rate of 0.001 was too low with an impact on the model's ability to learn within the available computational constraints. We therefore focused on a learning rate of 0.01. The ReLU activation function, $R(z) = max(0, z)$, was used in all models, and the cross-entropy loss function was used as specified in the coursework question. Glorot initialisation was used for all weights and biases in order to avoid exploding and vanishing gradients.

As shown in Table 2, we found that dropout led to worse results, and was not particularly useful in this use case.

1.3 Part 3: Final model training

From the various architectures tested, the highest validation accuracy was achieved by a 3 layer MLP model with layers sizes of 200, 50, 50. The code below shows the code used in this architecture.

```
class FinalNet(nn.Module):
    def __init__(self):
        super().__init__()
        self.dropout = nn.Dropout(p=0.5)

        self.fc1 = nn.Linear(28*28, 200)
        torch.nn.init.xavier_normal_(self.fc1.weight)
        torch.nn.init.zeros_(self.fc1.bias)

        self.fc2 = nn.Linear(200, 50)
        torch.nn.init.xavier_normal_(self.fc2.weight)
        torch.nn.init.zeros_(self.fc2.bias)

        self.fc3 = nn.Linear(50, 50)
        torch.nn.init.xavier_normal_(self.fc3.weight)
        torch.nn.init.zeros_(self.fc3.bias)

        self.fc4 = nn.Linear(50, 10)
        torch.nn.init.xavier_normal_(self.fc4.weight)
        torch.nn.init.zeros_(self.fc4.bias)

    def forward(self, x):
        x = x.view(-1,28*28)
        x = F.relu(self.fc1(x))
        x = F.relu(self.fc2(x))
        x = F.relu(self.fc3(x))
        x = self.fc4(x)
        return F.log_softmax(x, dim=1)
```

1.4 Part 4: Accuracy and loss plots

Figure 2 shows the loss and accuracy on both the training and validation sets for the best model.

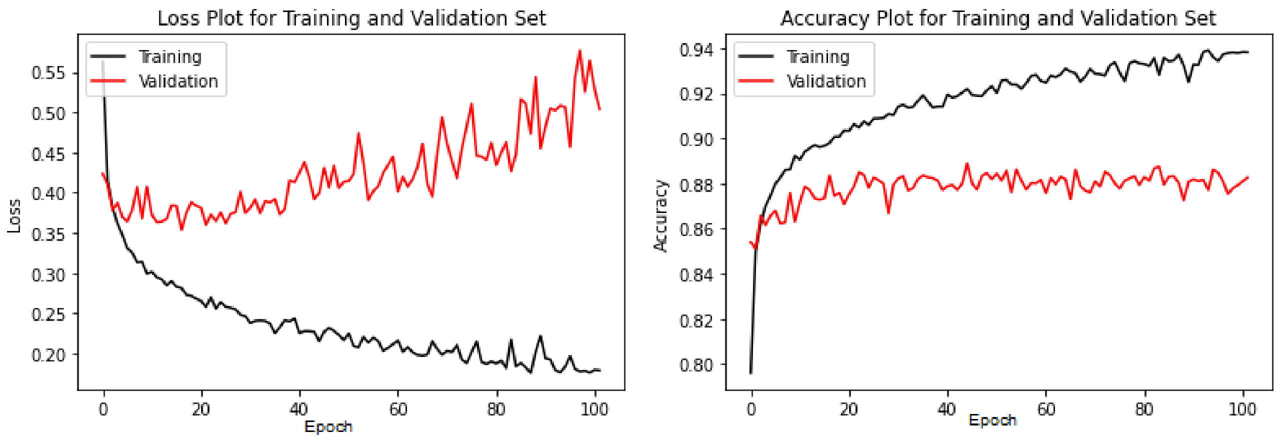


Figure 2: Loss and accuracy evolution of the training and validation sets

1.5 Part 5: Best model final metrics

Table 1 shows the final losses and accuracies for the training, validation and test sets, for the best model.

Dataset	**Accuracy**
Training	0.919
Validation	0.887
Test	0.883

Table 1: The training, validation, and test accuracy at the best epoch (42) for the best MLP architecture

Layer 1	**Layer 2**	**Layer 3**	**Dropout**	**Training Loss**	**Training Accuracy**	**Validation Loss**	**Validation Accuracy**
200	50	50	FALSE	0.248	0.919	0.426	0.887
200	50	/	FALSE	0.243	0.911	0.373	0.881
200	100	/	FALSE	0.249	0.909	0.430	0.881
200	100	/	FALSE	0.245	0.910	0.379	0.880
100	50	50	FALSE	0.252	0.909	0.391	0.878
200	100	50	FALSE	0.267	0.904	0.388	0.876
100	50	/	FALSE	0.257	0.907	0.431	0.875
200	100	25	FALSE	0.267	0.906	0.425	0.873
100	100	50	FALSE	0.262	0.905	0.395	0.870
200	50	25	FALSE	0.262	0.905	0.398	0.870
100	50	25	FALSE	0.271	0.899	0.391	0.869
100	50	/	FALSE	0.267	0.902	0.391	0.868
100	100	/	FALSE	0.269	0.900	0.404	0.867
100	100	/	FALSE	0.279	0.897	0.408	0.867
100	100	25	FALSE	0.247	0.911	0.425	0.867
200	50	/	FALSE	0.254	0.907	0.440	0.858
200	50	/	TRUE	1.284	0.497	0.756	0.715
200	100	/	TRUE	1.320	0.491	0.831	0.696
200	100	/	TRUE	1.297	0.490	0.841	0.654
200	50	/	TRUE	1.318	0.497	0.969	0.648
100	100	/	TRUE	1.616	0.346	1.248	0.534
100	100	/	TRUE	1.740	0.284	1.369	0.483
100	50	/	TRUE	1.649	0.320	1.333	0.469
100	50	/	TRUE	1.791	0.244	1.639	0.389
200	50	25	TRUE	1.780	0.295	1.468	0.385
100	100	25	TRUE	1.964	0.200	1.623	0.369
200	50	50	TRUE	1.938	0.226	1.630	0.321
200	100	50	TRUE	1.811	0.255	1.521	0.321
100	50	25	TRUE	2.094	0.179	1.802	0.319
200	100	25	TRUE	1.818	0.245	1.542	0.295
100	100	50	TRUE	2.068	0.175	1.804	0.264
100	50	50	TRUE	1.947	0.201	1.749	0.243

Table 2: Results of a range of MLP architectures trained on the Fashion-MNIST dataset

1.6 Part 6: Confusion matrix

A confusion matrix on the test set output was created to better understand the model's errors. In particular, the Python pycm package was used, using the following code.

```
from pycm import *
cm = ConfusionMatrix(actual_vector=labels, predict_vector=model_predictions)
cm.relabel(mapping={0:"T-shirt/top",1:"Trouses",2:"Pullover", 3:"Dress",
                4:"Coat", 5:"Sandal", 6:"Shirt", 7:"Sneaker", 8:"Bag",
                9:"Ankle boot"})
items = ['T-shirt/top', 'Trouses', 'Pullover', 'Dress', 'Coat', 'Sandal', 'Shirt', 'Sneaker',
↪  'Bag', 'Ankle boot']
cm.plot(cmap=plt.cm.Blues,number_label=True, class_name=items)
plt.xticks(rotation=90)
```

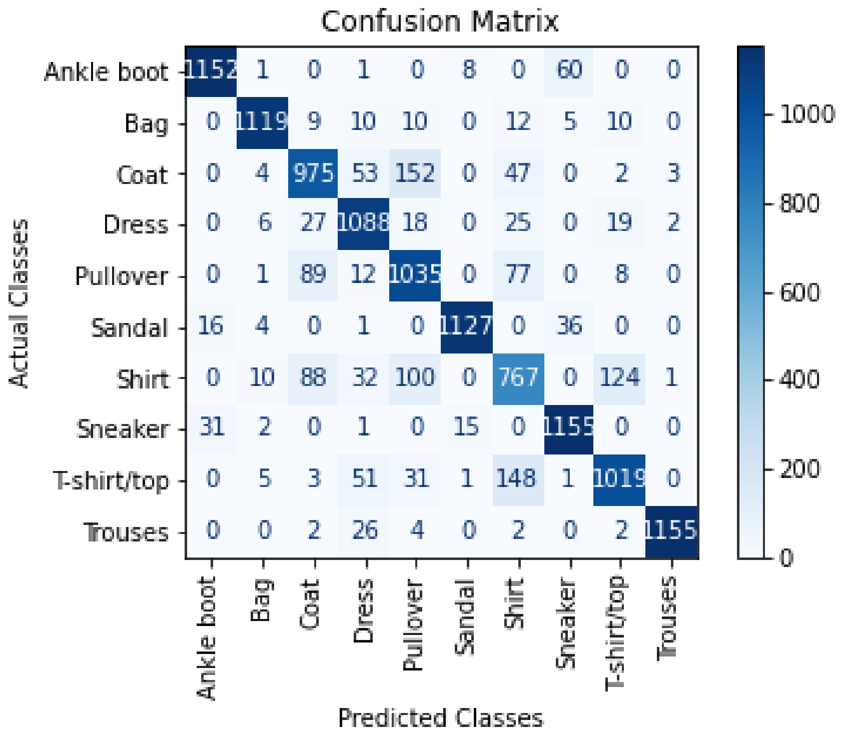


Figure 3: Confusion matrix of test set output

The highest error (152 mistakes) was found to be with coats, which the model identified as pullovers. Similarly, the model confused T-shirts with shirt (148 times) and shirts with T-shirts (124). This can be expected as these articles are quite similar in their overall forms, with both coats and pullovers having hoods, and shirts and T-shirts having the same overall shape. Another common mistaken identification made by the model was ankle boots, which the model sometimes identified as sneakers. Again, these are both 'shoes', and a similar error could be made even by humans.

It is interesting to note that the model confused ankle boots as sneakers more often than sneakers as ankle boots (60 times vs 31 times). A similar pattern is observed with the mistake of coats as pullovers vs pullover as coats (152 times vs 89 times). This could be explained as sneakers and pullovers are missing elements of ankle boots and coats and therefore not identified as such so often. Yet, ankle boots and coats contain the required features of sneakers and pullovers, leading to more errors. Saliency mapping might be one approach of future work to better understand this.

2 Q2: Denoising autoencoder

2.1 Part 1: Model implementation

All Python code corresponding to all aspects of Q2 can be found in Appendix B. We only include a few key relevant snippets in the current section. The code is commented to make it easy to follow and to identify relevant sections.

We apply a noise function which removes one or two quadrants of each image, as for example can be seen in Figure 4. Our aim is to use an autoencoder to reconstruct the original images prior to the application of noise, or "denoise" the image.

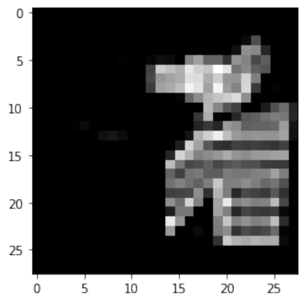


Figure 4: Noisy Image

The autoencoder schema consists of an encoder layer ϕ, and the decoder ψ. Defined as:

$$\phi : \mathbb{X} \rightarrow \mathbb{F} \tag{1}$$

$$\psi : \mathbb{F} \rightarrow \mathbb{X} \tag{2}$$

Our input is passed through an encoder activation such that $h = \sigma(Wx + b)$. This results in a compressed *"encoded"* version of our input, the decoder maps this into our reconstructed $x^{'}$ input: $x^{'} = \sigma^{'}(W^{'}h + b^{'})$. We train our autoencoder to minimize the mean square error loss for $\mathbb{L}(x, x^{'})$. The full code for the autoencoder can be seen in Appendix B.

2.2 Part 2: Hyperparameter search

A grid search approach was taken to optimise hyperparameters for the autoencoder. Particular attention was paid to optimising the structural aspects of the autoencoder.

- Deep compression: A structure was used to scale the image right down to a final 16 layer size. This resulted in a deep compression of the image. The alternative for this was to keep the final hidden layers relatively large to see if this benefited from more information about the structure of the input being stored.
- Large layer differences: Two approaches were chosen to see compare types of step in size between neighbouring hidden layers; first a steady reduction in the layer sizes, and second abrupt large reductions in layer size.

The batch size and learning rate were also varied as a hyperparameter. The grid search setup code can be seen below.

```
hyperparam_grid = {
  'structure': [
    [[28*28, 512], [512, 256], [256, 128], [128, 64], [64, 32], [32, 16]],
    [[28*28, 512], [256, 128], [128, 32], [32, 16]]
    [[28*28, 256], [256, 128], [128, 64], [64, 32], [32, 16]],
    [[28*28, 256], [256, 128], [128, 64]]
    [[28*28, 128], [128, 64], [64, 32], [32, 16]],
    [[28*28, 64], [64, 32], [32, 16]],
  ],
  'batch_size': [16, 32, 64, 128, 256],
  'learning_rate': [0.01, 0.001, 0.0001]
}
```

The best model that was selected had a layers size of $[[28 * 28, 256], [256, 128], [128, 64]]$, batch size of 64 and a learning rate of 0.001. This was then the model selected for the training and testing in the following sections.

2.3 Part 3: Final model training

The autoencoder was trained with the Adam optimiser. The model was trained either to convergence, with the stopping algorithm seen below, or to a maximum of 10,000 epochs. However, the latter was never reached as the model always converged.

```
# Check for early stopping
if current_epoch >= min_epochs:
  if (np.mean(validation_loss_list[-3:]) - np.mean(validation_loss_list[-6:-3])) > -.001:
    converged = True
```

2.4 Part 4: Loss plot

The loss plot for both the training, and validation set for the best model after the grid search can be seen in Figure 5, with the validation loss being 0.015678, and a training loss of 0.015234.

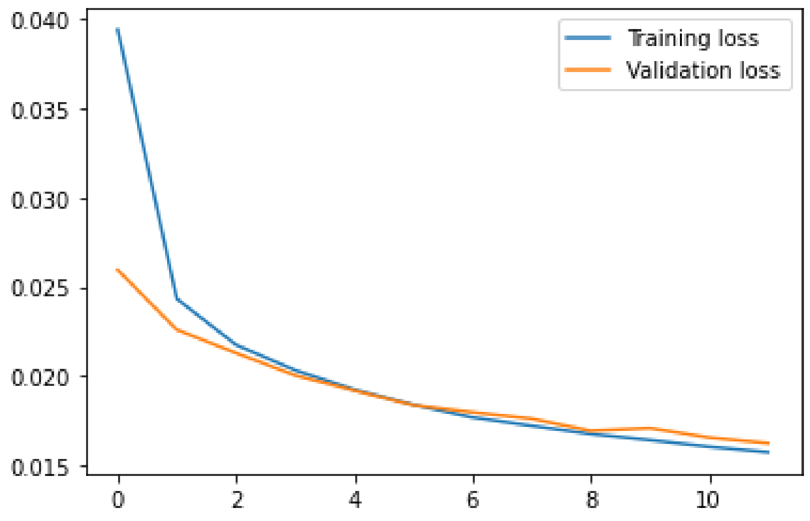


Figure 5: Validation and training loss plots, showing the loss at each epoch

2.5 Part 5: Clean, noisy, and decoded images

A random selection of 32 clean, noisy and decoded images can be seen in Figures 6, 7 and 8. The autoencoder is very good at restoring the image back to its original shape, even when the noisy image has a large chunk of the original structure missing; see the fifth image in Figure 6 as a good example. However, when our autoencoder faces any level of detail in the input it fails to replicate it. All the jumpers, shoes, etc., have the correct shape but lack that level of detail. When we have a zip or a logo the autoencoder fails to replicate the zip or logo. If the task were to predict which brand an item of clothing belonged to, this would not be the model to use.

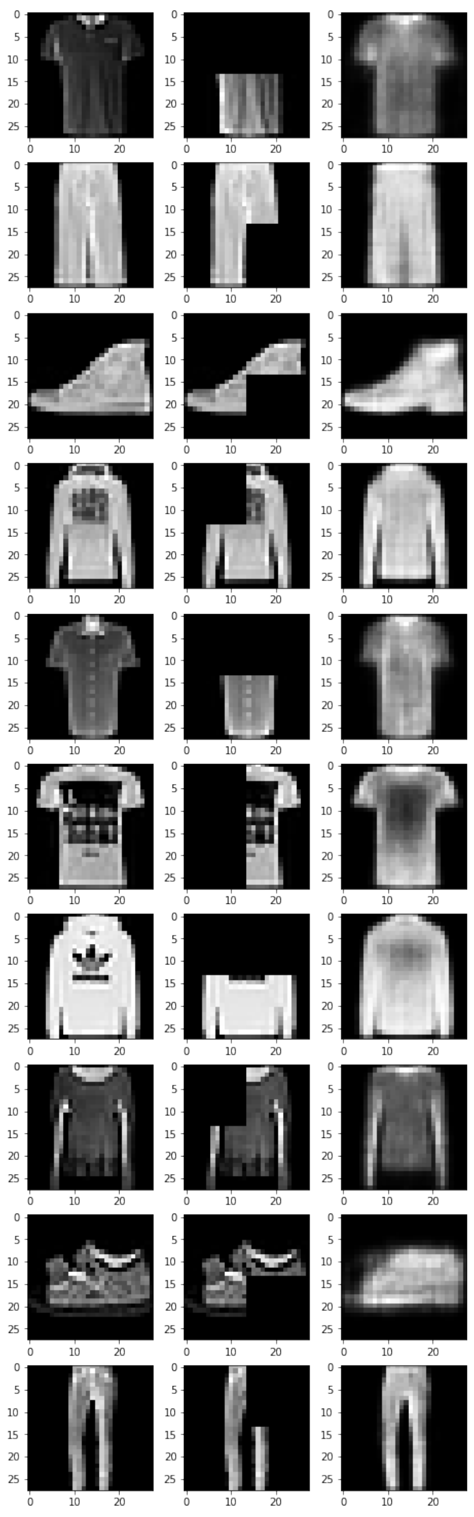


Figure 6: 10 clean, noisy, decoded images

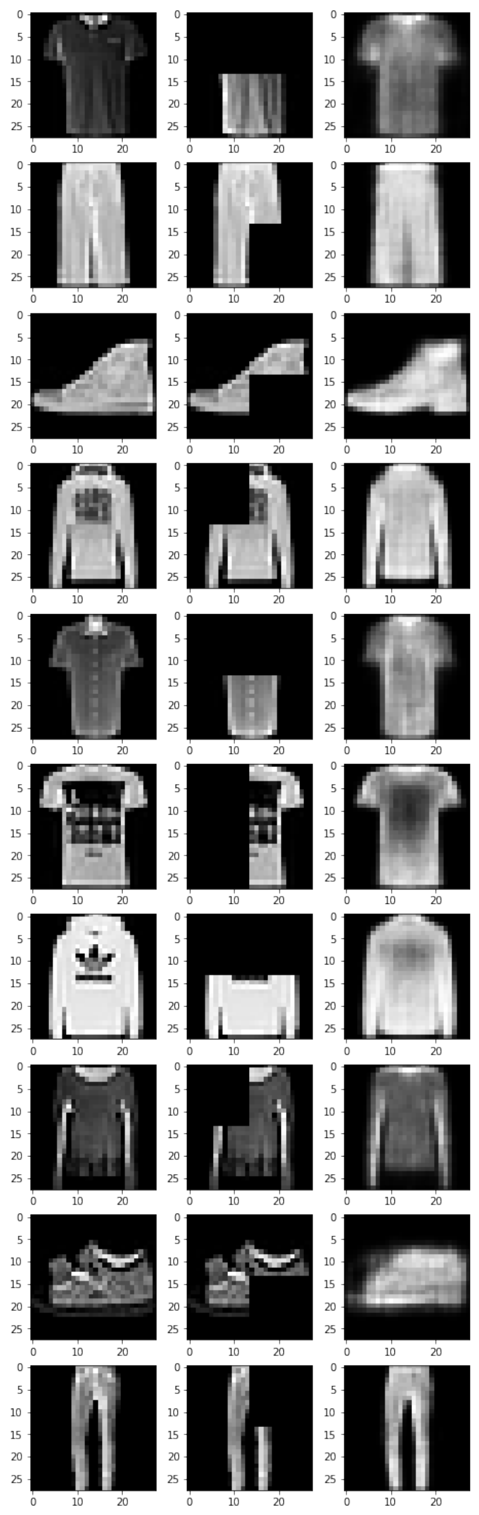


Figure 7: 10 clean, noisy, decoded images

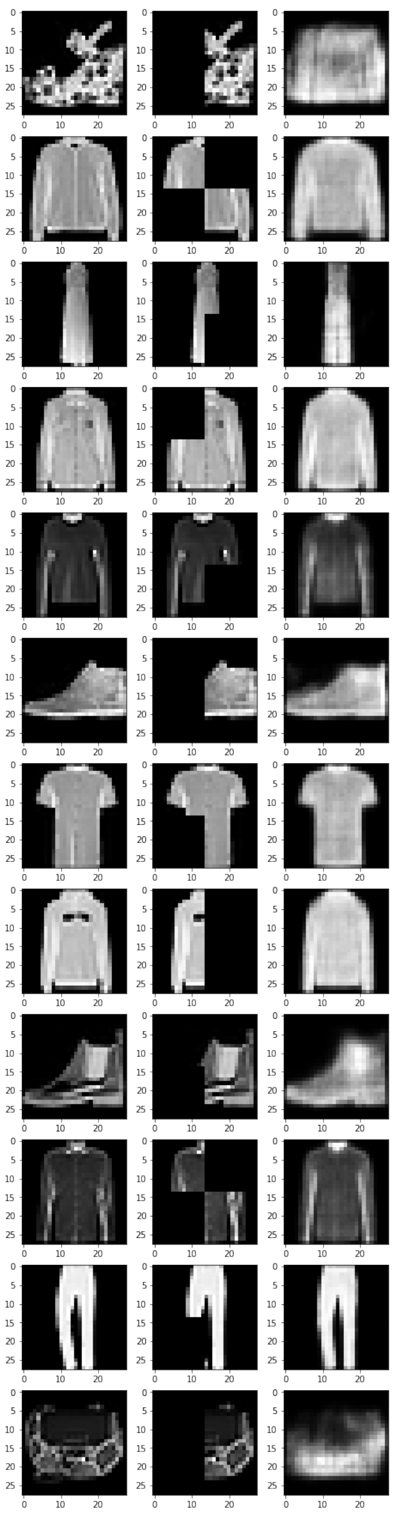


Figure 8: 12 clean, noisy, decoded images

3 Q3: Sequence prediction RNN

3.1 Part 1: Model implementation

All Python code corresponding to all aspects of Q3 can be found in Appendix C. We only include a few key relevant snippets in the current section. The code is commented to make it easy to follow and to identify relevant sections.

We implemented and trained character-level recurrent neural network models with cross-entropy loss, to take an input string and predict the subsequent character. All models were trained, validated, and tested on the textual portion of the Stanford Sentiment Treebank dataset. Our results from a range models trained during grid search iteration over hyperparameters, are summarised in Table 3.

The datasets from the Treebank were parsed to text-only by removing the tree structure. We then cleaned the data set by removing all punctuation except for apostrophes and full stops, removing any other symbols, converting all letters to lowercase, and removing all letters not within the English alphabet.

To enable batching during training, it was necessary to standardise the string length within each batch. To do so, the full corpus was merged into a single text, and batches containing standardised length sequences were randomly drawn from this large corpus. The corresponding target characters were drawn along with the input sequences. Between each batch, the sequence length for the next batch was randomly drawn from a uniform distribution ranging 10 to 50. We did this to ensure the model became robust to a varying range of input sequence lengths. Characters were converted to one-hot encoded vectors for feeding into the model.

The code corresponding to the model structure, loss function, and optimiser for our model is shown below:

```
class characterRNN(nn.Module):

    def __init__(self, input_size, hidden_size, num_layers, num_classes, cell_type,
    ↪   num_dense_layers, hidden_to_dense_size_ratio):
        super(characterRNN, self).__init__()
        self.num_layers = num_layers
        self.hidden_size = hidden_size
        self.cell_type = cell_type
        self.num_dense_layers = num_dense_layers
        if num_dense_layers == 2:
            self.hidden_to_dense_size_ratio = hidden_to_dense_size_ratio

        # only one of these will be used in forward statement (as specified by hyperparam
        ↪   cell_type for current run)
        self.rnn = nn.RNN(input_size, hidden_size, num_layers, batch_first = True)
        self.gru = nn.GRU(input_size, hidden_size, num_layers, batch_first = True)
        self.lstm = nn.LSTM(input_size, hidden_size, num_layers, batch_first = True)

        # number of these linear layers that gets used depends on hyperparam num_dense_layers
        self.linear1 = nn.Linear(hidden_size, num_classes)
        if num_dense_layers == 2:
            self.linear2 = nn.Linear(hidden_size, hidden_size//hidden_to_dense_size_ratio)
            self.linear3 = nn.Linear(hidden_size//hidden_to_dense_size_ratio, num_classes)

    def forward(self, x_batch):

        # Initialise hidden state
        h0 = torch.zeros(self.num_layers, x_batch.size(0), self.hidden_size).to(device)
        # Initialise cell state (used in LSTM only, not GRU or vanilla RNN)
        c0 = torch.zeros(self.num_layers, x_batch.size(0), self.hidden_size).to(device)

        # out shape: (batch_size, seq_length, hidden_size)
        if self.cell_type == "vanillaRNN":
            out, _ = self.rnn(x_batch, h0)
        if self.cell_type == "GRU":
            out, _ = self.gru(x_batch, h0)
        if self.cell_type == "LSTM":
            out, _ = self.lstm(x_batch, (h0, c0))

        # extract the final cells output and feed into a dense layer
        out = out[:, -1, :]
        if self.num_dense_layers == 1:
            out = self.linear1(out)
        if self.num_dense_layers == 2:
            out = F.relu(self.linear2(out))
            out = self.linear3(out)

        return out

# Loss & optimiser
criterion = nn.CrossEntropyLoss() # note that this will apply the softmax for us
optimiser = torch.optim.Adam(model.parameters(), lr=hyperparams['learning_rate'])
```

3.2 Part 2: Hyperparameter search

An initial exploration of reasonable hyperparameter combinations informed the set of hyperparameters which were subsequently iterated through in a grid search fashion. The full results of the grid search are shown in Table 3. Hyperparameters searched through included different recurrent units, hidden layer size, number of RNN layers, number and size of dense layers, learning rate. A search through a wider range of hyperparameter values may have yielded further improvements to the final model, but for the purposes of this assignment we were limited by computer resource and had to be selective. A batch size of 20 decided upon during the initial scoping experiments, cross-entropy loss, and the Adam optimiser were used throughout the grid search experiments. We compared each re-run based upon the best validation accuracies, in order to choose our final best model. The results in Table 3 are ordered from the model with the best validation accuracy down to the worst.

The dictionary specifying our final hyperparameter grid search was as follows:

```
hyperparam_grid = {'num_layers': [1,3],
                   'hidden_size': [128,256],
                   'cell_type': ["vanillaRNN","GRU","LSTM"],
                   'num_dense_layers': [1,2],
                   'learning_rate': [0.0002,0.0005],
                   'hidden_to_dense_size_ratio': [-99,2,4]} #-99 when N/A (when 1 dense layer)
```

3.3 Part 3: Final model training

Each model was trained till convergence on the validation dataset (see exemplar accuracy and loss plots in section Q3 Part 4). We implemented early stopping on detection of convergence. Our criterion for early stopping was a mean validation loss from the most recent 20 epochs being greater than the mean validation loss for the preceding 20 epochs. This averaging in the early stopping metric helped to prevent premature early stopping due to some expected choppiness in the validation loss between single epochs. The training had to precede for a minimum of 50 epochs before early stopping could be triggered. Adam optimiser was chosen, since as expected, it converged significantly more quickly than SGD during initial scoping experiments.

The early-stopping implementation is as shown here:

```
if epochs_counter >= min_epochs:
    if (np.mean(dev_loss_list[-20:]) - np.mean(dev_loss_list[-40:-20])) > 0:
        converged = True
        print("Model Converged")
```

3.4 Part 4: Accuracy and loss plots

The training and validation accuracy and loss plots for our best model are shown in Figure 9 and Figure 10 respectively. Early stopping criteria used were as described in section Q3 Part 3.

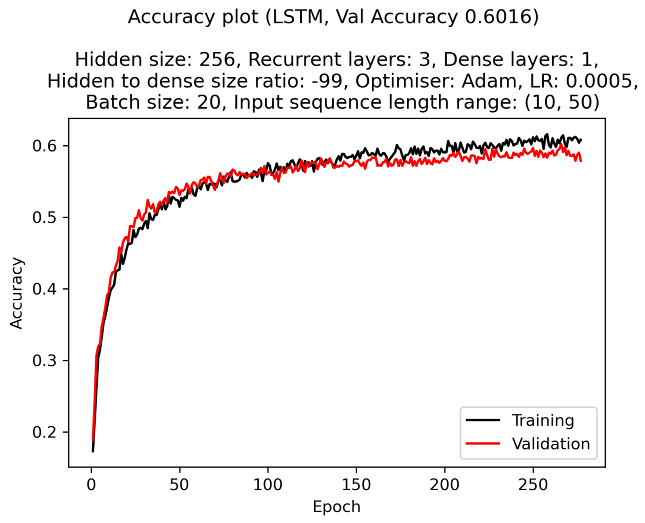


Figure 9: Training and validation accuracy by epoch for the character prediction language model

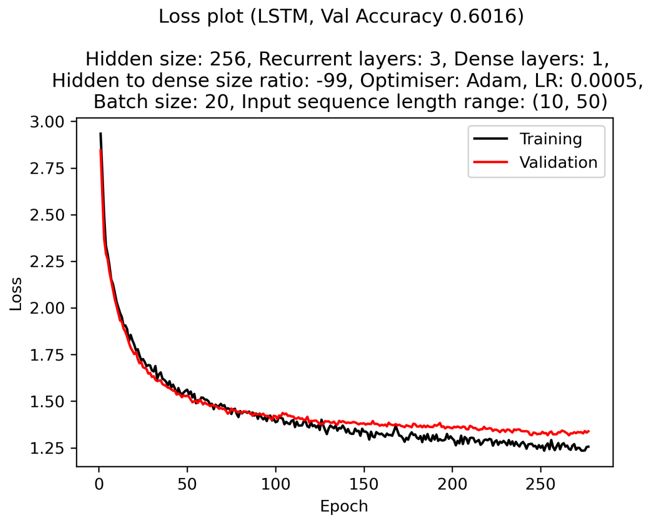


Figure 10: Training and validation loss by epoch for the character prediction language model

Hyperparameters							Metrics			
Cell type	**Num. RNN layers**	**Hidden size**	**Num. dense layers**	**Hidden-dense ratio**	**Learning rate**	**Best epoch**	**Best val acc**	**Associated train acc**	**Associated val loss**	**Associated train loss**
LSTM	3	256	1	N/A	0.0005	266	0.6016	0.6108	1.3171	1.2443
LSTM	3	256	2	4	0.0005	230	0.6016	0.5855	1.3313	1.3093
LSTM	3	128	1	N/A	0.0005	400	0.59	0.5902	1.3435	1.2922
GRU	3	256	1	N/A	0.0002	198	0.5872	0.5862	1.3816	1.3279
GRU	3	256	2	4	0.0002	326	0.5844	0.5965	1.4103	1.3009
LSTM	1	256	2	4	0.0005	251	0.5844	0.5763	1.3704	1.3638
GRU	3	256	2	2	0.0002	322	0.582	0.5914	1.3795	1.3103
LSTM	1	256	1	N/A	0.0005	240	0.58	0.5815	1.3965	1.3471
LSTM	1	256	2	2	0.0005	166	0.5792	0.5721	1.4194	1.387
LSTM	3	128	2	4	0.0005	308	0.578	0.5734	1.41	1.4021
LSTM	3	128	2	2	0.0005	332	0.576	0.5635	1.4188	1.4098
LSTM	3	256	2	2	0.0005	257	0.5748	0.5876	1.3933	1.3198
LSTM	1	128	1	N/A	0.0005	409	0.5728	0.5574	1.4241	1.4357
GRU	3	128	1	N/A	0.0002	253	0.5708	0.5624	1.4287	1.444
LSTM	1	128	2	2	0.0005	346	0.5704	0.5655	1.45	1.4093
vanillaRNN	3	256	1	N/A	0.0002	215	0.566	0.5523	1.4547	1.4559
vanillaRNN	3	256	2	4	0.0002	215	0.5624	0.5422	1.4622	1.4889
GRU	1	256	2	4	0.0002	347	0.5612	0.5765	1.4819	1.3921
GRU	3	128	2	4	0.0002	220	0.5608	0.5414	1.4751	1.4895
LSTM	1	128	2	4	0.0005	488	0.5604	0.5543	1.4621	1.4584
GRU	3	128	2	2	0.0002	191	0.5576	0.5383	1.4966	1.5255
vanillaRNN	3	256	2	2	0.0002	186	0.556	0.5325	1.458	1.5116
GRU	1	256	1	N/A	0.0002	269	0.5536	0.5525	1.5084	1.4589
vanillaRNN	3	128	1	N/A	0.0002	244	0.546	0.529	1.545	1.5394
GRU	1	256	2	2	0.0002	221	0.546	0.5369	1.539	1.5377
GRU	1	128	1	N/A	0.0002	383	0.534	0.5232	1.565	1.5591
vanillaRNN	3	128	2	4	0.0002	247	0.5336	0.5297	1.5576	1.5418
vanillaRNN	3	128	2	2	0.0002	214	0.5304	0.5257	1.5648	1.5759
GRU	1	128	2	4	0.0002	305	0.5256	0.5094	1.6029	1.6275
GRU	1	128	2	2	0.0002	304	0.5244	0.5124	1.5829	1.6098
vanillaRNN	1	256	1	N/A	0.0002	340	0.5164	0.5053	1.6214	1.6352
vanillaRNN	1	256	2	4	0.0002	300	0.4844	0.4707	1.7438	1.7684
vanillaRNN	1	128	1	N/A	0.0002	316	0.4816	0.4577	1.7479	1.7946
vanillaRNN	1	128	2	4	0.0002	332	0.4788	0.4572	1.7571	1.7744
vanillaRNN	1	128	2	2	0.0002	296	0.4748	0.4549	1.7783	1.7938
vanillaRNN	1	256	2	2	0.0002	234	0.4472	0.4352	1.8632	1.874

Table 3: Results of hyperparameter grid search for character completion models trained on the Stanford Sentiment Treebank dataset

3.5 Part 5: Best model final metrics

As shown in Table 4, our best single model achieved a highest accuracy of 0.6016 on the validation dataset after 266 epochs of training, at predicting the next character from an input sequence. This epoch was associated with an accuracy of 0.6108 on the training dataset. Final evaluation of the model was performed on the test dataset, achieving an accuracy of 0.5892.

Dataset	**Accuracy**
Training	0.6108
Validation	0.6016
Test	0.5892

Table 4: The training, validation, and test accuracy at the best epoch (266) for the best character prediction model

3.6 Part 6: Sentence completion

Here we provide 5 sentence completions generated by our model in 3 different ways, for new inputs that we provided. Text generation for each was capped at 100 characters long. We noted that all models tended to output loops of repeated phrases by default. Therefore we show sentences generated in 3 different ways, with the latter two being methods we used to try to prevent this:

1. All characters chosen as those deemed most likely by the model.
2. The first character of each word (i.e. any character following a space) drawn from the distribution over the characters determined by the model. All other characters chosen as those deemed most likely by the model.
3. All characters drawn from the distribution over the characters determined by the model.

The disadvantage of approaches 2 and 3 is that the randomness can still result in low probability characters being selected. This might go some way to explaining why whilst the words in the sentences are spelt correctly in approach 2, the model struggles to output appropriate whole phrases of related words with a clear meaning. In approach 3, we see poorer spelling where low probability letters have been selected within words.

A more robust alternative approach to attempt to resolve this issue of recurring phrases, to be considered in further future work, would be to implement beam search. We also expect the issue is made worse because we have implemented a character-level rather than word-level model here.

In each of the sentences below, the non-bold characters were inputted by the human, and the bold characters were generated by the model.

Approach 1: Sentence predictions when the most likely character is predicted at every iteration:

- dobby squeals as if **it is a film that it is a film that it is a film that it is a film that it is a**
- when dorothy pu**rpose and a film is a film that it is a film that it is a film that it is a film that**
- the incredible hulk ter**ms of the story is a film is a film that it is a film that it is a film that**
- nemo finally finds hi**s story and the film is a film is a film that it is a film that it is a film th**
- james bond's mys**tery and a film is a film that it is a film that it is a film that it is a film that**
- the director's decision to **be a story and the film is a film is a film that it is a film that it is**

Approach 2: Sentence predictions when the first letter after each space is pulled from the probability distribution of the characters at that point:

- dobby squeals as if **not very insightful end the story story of movie about the resonant performance**
- when dorothy pu**rpose and consciousness and just the world be because it all the movie but the experi**

- the incredible hulk ter**ms of what it never rises because under his character is a movie that or and**
- nemo finally finds hi**s performance is more about some insightful story is often acting consciousness**
- james bond's mys**tery and nothing is good that it is explosive and often profice and nothing more abo**
- the director's decision to **incoherent who wants to conscious and fun and well acted and funny with b**

Approach 3: Sentence predictions when the every letter is pulled from the probability distribution of the characters at that point:

- dobby squeals as if **you geer more like the thouty sxasily recilemen that it sidesses its vacual pana**
- when dorothy pu**elography astonio fun of experiment to bad his fudity consulititał.**
- the incredible hulk ter**ms like some singer that's real she's bad.**
- nemo finally finds his **sports after bother.**
- james bond's mys**tery the mast jakes for could ball the moments out taking rewards off with a high wr**
- the director's decision to **very final insament.**

4 Q4: Bag of vectors

4.1 Part 1: Model implementation

All Python code corresponding to all aspects of Q4 can be found in Appendix D. We only include a few key relevant snippets in the current section. The code is commented to make it easy to follow and to identify relevant sections.

Our aim is to implement a bag of vectors model which attempts to predict the sentiment score (0-4) of a movie review by using pre-trained word embeddings, pooling these and feeding them into a multi-layer perceptron to perform multi-class classification.

In order to implement this model, we first loaded Google's pre-trained Word2Vec model and normalised the embedding vectors. Due to memory constraints, we also limited this embedding of 3 million words to include only the 1 million most common words:

```
!wget https://s3.amazonaws.com/dl4j-distribution/GoogleNews-vectors-negative300.bin.gz

# Load Google's pre-trained Word2Vec model
w2v = gensim.models.KeyedVectors.load_word2vec_format("GoogleNews-vectors-negative300.bin.gz",
↪  binary=True, limit = 1000000)

# Normalise the vectors
w2v.init_sims(replace=True)
```

Next, we created a dictionary mapping each of these 1 million tokens to their index in order to retrieve the required embedding vectors using these indices in the model. Unknown tokens are mapped to the token "UNK":

```
token2index_map = {
    word:idx
    for idx, word in enumerate(w2v.vocab)
}

def token2index(token):
  # Use "UNK" as missing token embedding
  default = token2index_map.get("UNK")

  return token2index_map.get(token, default)
```

Next, similarly to in Q3, we parsed the train, validation and test datasets from the Stanford Sentiment Treebank to text-only, removing the tree structure and removing all punctuation including, in this case, apostrophes and full stops, and otherwise cleaned the dataset in order to obtain the sentences in a simple form along with their associated sentiment.

One major issue we faced was the variable sentence lengths in the datasets. Fixed size inputs are required within each batch to feed into the MLP and so, in order to tackle this, we implemented a custom data loader which tokenises the input sentences, maps these tokens to their associated Word2Vec indices, and batches together sentences of equal length. Again, full Python code for the data parsing and cleaning and for the sentence batcher and data loader can be found in Appendix D.

The code corresponding to the model structure, loss function, and optimiser for our model is shown below:

```
class MLP(nn.Module):
  def __init__(self, in_size, hid_size, out_size, num_layers, dropout, pooling_type,
  ↪  fixed_embeddings):
    super().__init__()
    self.layer1 = nn.Linear(in_size, hid_size)
    self.layer2a = nn.Linear(hid_size, out_size)
    self.layer2b = nn.Linear(hid_size, hid_size//2)
    self.layer3a = nn.Linear(hid_size//2, out_size)
    self.layer3b = nn.Linear(hid_size//2, hid_size//4)
    self.layer4 = nn.Linear(hid_size//4, out_size)

    self.num_layers = num_layers
    self.pooling_type = pooling_type
    self.embedding = nn.Embedding.from_pretrained(torch.FloatTensor(w2v.vectors), freeze =
    ↪  fixed_embeddings)
    self.dropout = nn.Dropout(dropout)

  def forward(self, x_batch):

    # Get word embeddings from W2V indices
    emb = self.embedding(x_batch)

    if self.pooling_type == "max":
      # max pooling
      batch_vec = torch.max(emb, dim = 1)[0]

    elif self.pooling_type == "min":
      # min pooling
      batch_vec = torch.min(emb, dim = 1)[0]

    elif self.pooling_type == "mean":
      # mean pooling
      batch_vec = torch.mean(emb, dim = 1)

    elif self.pooling_type == "mean_max_min":
      # mean, max and min pooling concatenated together
      pool_mean = torch.mean(emb, dim = 1)
      pool_max = torch.max(emb, dim = 1)[0]
      pool_min = torch.min(emb, dim = 1)[0]
      batch_vec = torch.cat((pool_mean, pool_max, pool_min), dim=1)

    if self.num_layers == 2:
      x = self.layer1(batch_vec)
      x = torch.relu(x)
      x = self.dropout(x)
      x = self.layer2a(x)

    elif self.num_layers == 3:
      x = self.layer1(batch_vec)
      x = torch.relu(x)
      x = self.dropout(x)
      x = self.layer2b(x)
      x = torch.relu(x)
      x = self.dropout(x)
      x = self.layer3a(x)

    elif self.num_layers == 4:
      x = self.layer1(batch_vec)
      x = torch.relu(x)
      x = self.dropout(x)
      x = self.layer2b(x)
      x = torch.relu(x)
      x = self.dropout(x)
      x = self.layer3b(x)
      x = torch.relu(x)
```

```
        x = self.dropout(x)
        x = self.layer4(x)

    return x

# Loss & optimiser
criterion = nn.CrossEntropyLoss()
optimizer = torch.optim.Adam(model.parameters(), lr=hyperparams['learning_rate'])
```

4.2 Part 2: Hyperparameter search

Now with a model in place, we search over multiple hyperparameter combinations in order to deduce empirically what works best on the validation set. In particular, we vary hidden layer sizes, learning rate, number of layers, dropout probability, pooling type, batch size and whether the embeddings are fixed during training or allowed to vary:

```
hyperparam_grid = {'hid_size': [64, 128],
                    'learning_rate': [0.01, 0.001, 0.0001],
                    'num_layers': [3, 4],
                    'dropout': [0, 0.25],
                    'pooling_type': ["max", "mean", "mean_max_min"],
                    'batch_size': [64, 128],
                    'fixed_embeddings': [True]
 }
```

We selected this particular set of hyperparameters to grid search over after an initial informal scoping of numerous combinations. After this initial investigation, we fixed the Adam optimiser to be used throughout this question as we found it to perform well when compared with other optimisers and chose to reduce hidden layer sizes by a factor of two as you go through the network as this choice also appeared to return the best results. In addition, we found that allowing the embeddings to update during training significantly increased computation time. Therefore, for the sake of increased efficiency, we first varied the above hyperparameters while keeping the embeddings fixed and this allowed us to narrow down the optimal set of hyperparameters and re-ran the grid-search on a smaller subset of these parameters but while updating the embeddings this time. We compared each re-run based upon the best validation accuracies, in order to choose our final best model, with the results again being ordered from the model with the best validation accuracy down to the worst.

After the first, wider grid search with fixed word embeddings, the results of which can be found in Table 5, we found in particular that using mean pooling was clearly providing the best results, with max pooling not even appearing in the top 50 best results, suggesting that mean pooling provides the best summary of the sentences in terms of sentiment, even better than mean, max and min pooling concatenated together. In addition, the top 9 results all had a learning rate of 0.001, suggesting this is optimal. Models using the lower learning rate of 0.0001 took much longer to converge as, for these models, the epoch with the best validation accuracy tended to be in the range of epoch 150-200 rather than 20-40 as would be expected. Using dropout did not seem to improve results much as most of the top results had 0 dropout, perhaps because the network was not large enough to properly warrant its use. Otherwise, hidden layer size, batch size and number of layers all varied among the top results and so in the second, more refined grid search where the embeddings were allowed to vary, we searched over these parameters again. We also did include a nonzero dropout in the second hyperparameter search despite not much improvement when using it the first time around since we thought it may help to prevent overfitting as we found that the model tended to strongly overfit when the embeddings were updated during training.

The second hyperparameter grid we searched over with varied word embeddings can be found below:

```
hyperparam_grid = {'hid_size': [64, 128],
                   'learning_rate': [0.001],
                   'num_layers': [3, 4],
                   'dropout': [0, 0.25],
                   'pooling_type': ["mean"],
                   'batch_size': [64, 128],
                   'fixed_embeddings': [False]
}
```

The results in Table 6 correspond to the second more refined grid search where the embeddings were allowed to vary. In particular, we found that allowing the embeddings to update did not tend to increase validation accuracy and the model often overfitted extremely quickly to the training data, with or without dropout, usually reaching peak validation accuracy within the first couple of epochs.

Hyperparameters								**Metrics**			
Pooling type	**Num. layers**	**Batch size**	**Dropout**	**Fixed embeddings**	**Hidden size**	**Learning rate**	**Best epoch**	**Best val acc**	**Associated train acc**	**Associated val loss**	**Associated train loss**
mean	4	64	0	1	128	0.001	23	0.4514	0.4974	1.3242	1.1403
mean	3	128	0	1	64	0.001	35	0.4487	0.4801	1.2793	1.1554
mean	3	64	0.25	1	128	0.001	30	0.4478	0.493	1.3149	1.1494
mean	3	128	0.25	1	128	0.001	35	0.4478	0.4757	1.2817	1.1586
mean	4	64	0.25	1	128	0.001	30	0.4469	0.4822	1.3287	1.1605
mean	3	64	0	1	64	0.001	33	0.4469	0.4964	1.2828	1.1565
mean	3	128	0	1	128	0.001	42	0.446	0.4981	1.3281	1.1026
mean	3	64	0	1	128	0.001	18	0.445	0.4861	1.2786	1.17
mean	4	128	0	1	128	0.001	29	0.445	0.4837	1.3209	1.1456
mean	3	64	0	1	128	0.0001	162	0.4432	0.483	1.2859	1.1683
mean	3	64	0.25	1	128	0.0001	186	0.4414	0.4909	1.287	1.1812
mean	3	64	0.25	1	64	0.001	38	0.4414	0.4814	1.302	1.1754
mean	3	128	0.25	1	64	0.001	52	0.4405	0.467	1.3142	1.1733
mean	3	128	0.25	1	128	0.0001	199	0.4396	0.4724	1.2863	1.1833
mean_max_min	3	128	0	1	128	0.0001	116	0.4387	0.484	1.3055	1.1167
mean	4	64	0	1	64	0.0001	195	0.4378	0.4643	1.2967	1.2053
mean_max_min	4	64	0.25	1	64	0.0001	144	0.4378	0.4535	1.2946	1.2027
mean	3	128	0	1	128	0.01	24	0.4378	0.5205	1.5072	1.0593
mean_max_min	3	64	0.25	1	64	0.0001	117	0.4369	0.4698	1.2903	1.2016
mean	4	128	0.25	1	64	0.001	36	0.436	0.44	1.3054	1.2238
mean_max_min	3	128	0.25	1	64	0.0001	192	0.436	0.4751	1.2996	1.1599
mean	4	64	0.25	1	64	0.0001	174	0.436	0.4512	1.3047	1.2314
mean	3	64	0.25	1	128	0.01	20	0.4351	0.4477	1.31	1.2542
mean	4	64	0	1	128	0.0001	163	0.4351	0.4721	1.3072	1.1817
mean	3	128	0	1	64	0.01	19	0.4351	0.4863	1.3388	1.1458
mean	4	64	0	1	128	0.01	27	0.4351	0.5229	1.5925	1.1123
mean_max_min	3	64	0.25	1	64	0.001	24	0.4342	0.4542	1.2808	1.2362
mean	3	64	0.25	1	64	0.0001	197	0.4342	0.4636	1.2917	1.2098
mean	4	128	0.25	1	128	0.001	32	0.4342	0.4685	1.3131	1.1679
mean	3	64	0	1	64	0.01	18	0.4342	0.4888	1.3295	1.1566
mean_max_min	4	128	0	1	128	0.0001	72	0.4342	0.4556	1.3116	1.193
mean_max_min	3	64	0	1	64	0.0001	124	0.4342	0.4877	1.3	1.1548
mean	4	64	0.25	1	64	0.001	30	0.4332	0.4583	1.326	1.2121
mean	3	64	0	1	64	0.0001	193	0.4332	0.4723	1.2948	1.1849
mean	4	128	0	1	64	0.01	17	0.4332	0.4652	1.3809	1.1876
mean	4	128	0	1	128	0.01	20	0.4323	0.4888	1.3943	1.1313
mean	4	64	0	1	64	0.001	21	0.4314	0.4684	1.2964	1.2071
mean	4	128	0	1	128	0.0001	199	0.4314	0.4714	1.3037	1.1531
mean_max_min	4	128	0.25	1	64	0.001	57	0.4314	0.44	1.3335	1.2269
mean	3	128	0	1	128	0.0001	190	0.4314	0.4705	1.2903	1.1815
mean	3	128	0.25	1	64	0.01	11	0.4305	0.4346	1.2862	1.279
mean_max_min	3	128	0	1	64	0.0001	141	0.4296	0.4521	1.3086	1.183
mean	3	128	0.25	1	128	0.01	23	0.4287	0.4382	1.3387	1.2459
mean	3	64	0	1	128	0.01	16	0.4287	0.4934	1.3593	1.1526
mean_max_min	4	128	0.25	1	64	0.0001	199	0.4287	0.4356	1.294	1.215
mean_max_min	4	64	0.25	1	64	0.001	35	0.4287	0.4366	1.3021	1.2553
mean_max_min	3	128	0.25	1	128	0.0001	109	0.4287	0.4637	1.2813	1.1824
mean	4	64	0.25	1	128	0.0001	108	0.4287	0.4582	1.3012	1.2134
mean	4	128	0.25	1	128	0.01	31	0.4278	0.4398	1.3421	1.2369
mean	4	64	0	1	64	0.01	23	0.4278	0.5076	1.4508	1.0894

Table 5: Results of hyperparameter grid search for bag of vectors models (fixed word embeddings) trained on the Stanford Sentiment Treebank dataset (top 50 results shown, ordered by best validation accuracy)

Hyperparameters								Metrics			
Pooling type	**Num. layers**	**Batch size**	**Dropout**	**Fixed embeddings**	**Hidden size**	**Learning rate**	**Best epoch**	**Best val acc**	**Associated train acc**	**Associated val loss**	**Associated train loss**
mean	3	64	0.25	0	64	0.001	2	0.4269	0.5487	1.3668	1.0565
mean	3	64	0	0	128	0.001	1	0.4142	0.4702	1.4244	1.1941
mean	3	128	0.25	0	128	0.001	1	0.4105	0.4077	1.3629	1.3718
mean	4	64	0	0	128	0.001	1	0.4096	0.4416	1.3853	1.3038
mean	3	128	0	0	64	0.001	2	0.4087	0.5067	1.4045	1.1094
mean	4	128	0	0	128	0.001	1	0.4051	0.3958	1.3869	1.3918
mean	4	128	0.25	0	128	0.001	2	0.4024	0.4841	1.4114	1.1516
mean	3	128	0	0	128	0.001	7	0.4015	0.9656	3.0211	0.1109
mean	3	128	0.25	0	64	0.001	2	0.3969	0.4643	1.4019	1.2729
mean	3	64	0.25	0	128	0.001	1	0.3951	0.454	1.3341	1.2284
mean	3	64	0	0	64	0.001	1	0.3933	0.4515	1.3514	1.2779
mean	4	64	0.25	0	128	0.001	1	0.3924	0.445	1.4183	1.3075
mean	4	128	0.25	0	64	0.001	2	0.3915	0.4225	1.4558	1.3998
mean	4	64	0.25	0	64	0.001	20	0.3869	0.9695	6.5892	0.0978
mean	4	128	0	0	64	0.001	1	0.3842	0.3942	1.4168	1.458
mean	4	64	0	0	64	0.001	10	0.3806	0.9954	4.5752	0.019

Table 6: Results of hyperparameter grid search for bag of vectors models (varied word embeddings) trained on the Stanford Sentiment Treebank dataset (ordered by best validation accuracy)

4.3 Part 3: Final model training

As in Q3 we train each model to convergence on the validation dataset, implementing an early stopping criterion which defines convergence as being when the mean validation loss from the last 10 epochs minus the mean validation loss from the preceding 10 epochs is greater than 0.01:

```
# Early stopping criteria
      if epoch >= min_epochs:
        if (np.mean(dev_costs[-10:]) - np.mean(dev_costs[-20:-10])) > 0.01:
          converged = True
          print("Model Converged")
```

Choosing the average values over the last 10 epochs as well as using 0.01 as the critical value helps to prevent premature early stopping. For the first run of the model involving a wide grid search using fixed embeddings, we set the minimum number of epochs before the possibility of early stopping being triggered as 30 and the maximum number of epochs for the model to run if early stopping is not triggered as 200. For the second grid search, due to the fact that the model overfitted extremely quickly with the embeddings being updated, we slightly reduced these bounds (while still giving the model ample time to run) to be a minimum of 20 epochs and a maximum of 100. In all the scenarios we ran in which the embeddings were allowed to update, the early stopping was triggered at epoch 20, the earliest possible time.

The final model we chose was the one which reached the highest validation accuracy overall: 0.4514 as can be seen from Table 5. The hyperparameters of this model were:

Hyperparameter	**Value**
Pooling type	Mean
Number of layers	4
Batch size	64
Dropout probability	0
Fixed embeddings	True
Hidden layer size	128
Learning rate	0.001

Table 7: The hyperparameters for the best performing bag of vectors model

4.4 Part 4: Accuracy and loss plots

The training and validation accuracy and loss plots for our best model are shown in Figure 11, and Figure 12 respectively. Early stopping criteria used were as described in section Q4 Part 3.

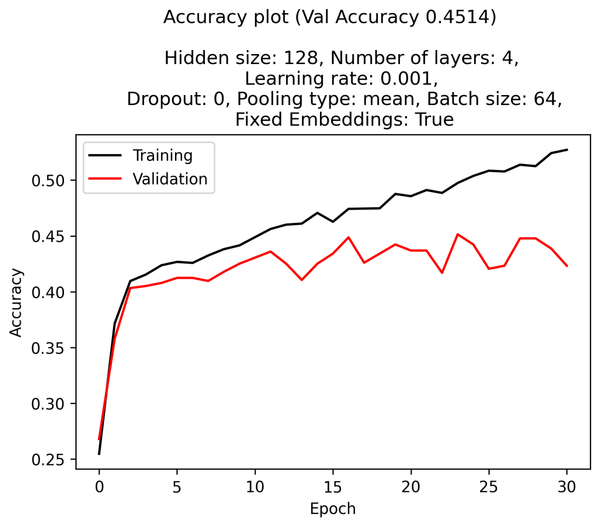


Figure 11: Training and validation accuracy by epoch for the best bag of vectors model

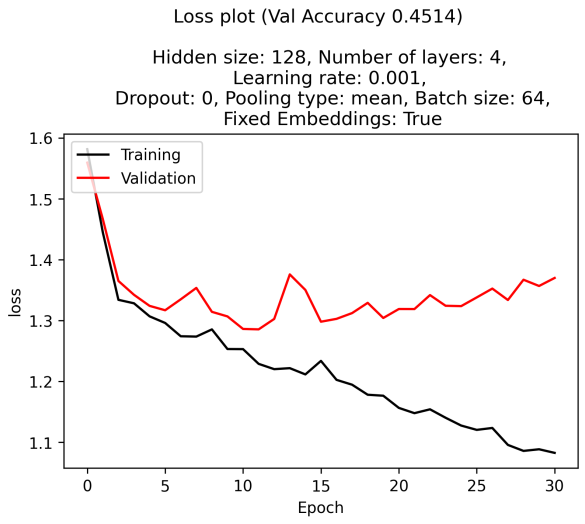


Figure 12: Training and validation loss by epoch for the best bag of vectors model

4.5 Part 5: Best model final metrics

As shown in Table 5, our best model reached its highest accuracy of 0.4514 on the validation dataset after 23 epochs of training. This epoch was associated with an accuracy of 0.4974 on the training dataset. Final evaluation of the model was performed on the test dataset, achieving an accuracy of 0.4240.

Dataset	**Accuracy**
Training	0.4974
Validation	0.4514
Test	0.4240

Table 8: The training, validation, and test accuracy at the best epoch (23) for the best bag of vectors model

4.6 Part 6: Online review classifications

We obtained a selection of opinionated online reviews, and pre-processed these according to the same rules as the training, validation, and test datasets.

We then fed these into our best model to obtain the following classification decisions.

> To this day, this is still my favorite pixar film. The animation is stellar, its heartwarming, funny and proves that pixar movies are always bound to be great (except for cars 2 but thats a different story). This has a shot at the title "best movie of the century"
>
> `https://www.imdb.com/review/rw5485122`

Prediction: 4

> But the worst thing of all with this film is the mangling of Austen's dialogue and the atrocious modern dialogue. Austen's dialogue needs no assistance from a writer who thinks he/she can write like Austen.
>
> `https://www.imdb.com/review/rw1213354/`

Prediction: 1

> It sucked. It was boring, it was contrived, it was cheaply done and the characters made zero sense. All the coincidences and stupid behavior almost put me to sleep. If there is any learning that can happen in this world it will be the end of sequels forever and ever.
>
> `https://www.imdb.com/review/rw4229247/`

Prediction: 0

> One of the all time great feel good movies, incredible story- brilliant writing and directing all around. Don't think that there is a single actor in the world who could've done a better job than Jim Carrey. A must watch.
>
> `https://www.imdb.com/review/rw5873851/`

Prediction: 4

> One of the best romcom movies of all time. Great cast and superb acting. Hard not to get emotional and invested once it starts.
>
> `https://www.imdb.com/review/rw5895794/`

Prediction: 4

5 Q5: Sequence encoding RNN

5.1 Part 1: Model implementation

All Python code corresponding to all aspects of Q5 can be found in Appendix E. We only include a few key relevant snippets in the current section. The code is commented to make it easy to follow and to identify relevant sections.

We implemented and trained recurrent neural networks with cross-entropy loss, to predict the sentiment score (0-4) of a movie review. We used Google's pre-trained Word2Vec embeddings, normalizing these as in Q4, and including only the 1 million most common words due to memory constraints.

All models were trained, validated, and tested on the textual portion of the Stanford Sentiment Treebank dataset. Our results from a range models trained during grid search iteration over hyperparameters are summarised in Table 9.

The datasets from the Treebank were parsed to text-only by removing the tree structure. We then cleaned the data set by removing all punctuation except for apostrophes and full stops, removing any other symbols, changing all letters to lowercase, and removing all letters not within the English alphabet.

As in Q4, we faced the issue of variable sentence lengths in the datasets. In order to enable batching during training, we implemented a custom data loader which tokenises the input sentences, maps these to Word2Vec indices, and batches together sentences of equal length.

The code corresponding to the model structure, loss function, and optimiser for our model is shown below:

```
class RNN(nn.Module):
  def __init__(self, embedding_size, hidden_size, num_layers, num_classes, cell_type,
   ↪  num_dense_layers, hidden_to_dense_size_ratio=2, dropout=0, fixed_embeddings=True):
    super(RNN, self).__init__()
    self.num_layers = num_layers
    self.hidden_size = hidden_size
    self.cell_type = cell_type
    self.num_dense_layers = num_dense_layers

    self.embedding = nn.Embedding.from_pretrained(torch.FloatTensor(w2v.vectors))
    self.embedding.weight.requires_grad = not fixed_embeddings

    # Only one of these will be used in forward statement (as specified by hyperparam
    ↪  cell_type for current run)
    self.rnn = nn.RNN(embedding_size, hidden_size, num_layers, dropout=dropout, batch_first =
    ↪  True)
    self.gru = nn.GRU(embedding_size, hidden_size, num_layers, dropout=dropout, batch_first =
    ↪  True)
    self.lstm = nn.LSTM(embedding_size, hidden_size, num_layers, dropout=dropout, batch_first
    ↪  = True)

    # Number of these linear layers that gets used depends on hyperparam num_dense_layers
    if num_dense_layers == 1:
      self.linear = nn.Linear(hidden_size, num_classes)
    elif num_dense_layers == 2:
      self.linear1 = nn.Linear(hidden_size, hidden_size//hidden_to_dense_size_ratio)
      self.linear2 = nn.Linear(hidden_size//hidden_to_dense_size_ratio, num_classes)
      self.dropout = nn.Dropout(dropout)

  def forward(self, x_batch):
    # Get the word embeddings
    embedded = self.embedding(x_batch)

    if self.cell_type == 'VanillaRNN':
      packed_out, _ = self.rnn(embedded)
    elif self.cell_type == 'GRU':
      packed_out, _ = self.gru(embedded)
    elif self.cell_type == 'LSTM':
      packed_out, _ = self.lstm(embedded)

    # Extract the final output and feed into a dense layer
    out = packed_out[:, -1, :]
    if self.num_dense_layers == 1:
      out = self.linear(out)
    elif self.num_dense_layers == 2:
      out = self.linear1(out)
      out = torch.relu(out)
      out = self.dropout(out)
      out = self.linear2(out)
    return out

    # Loss & optimiser
    criterion = nn.CrossEntropyLoss() # note that this will apply the softmax for us
    optimiser = torch.optim.Adam(model.parameters(), lr=hyperparams['learning_rate'])
```

5.2 Part 2: Hyperparameter search

We performed grid search to find the hyperparameters that worked best on the validation set. We varied a range of hyperparameters, including the number of recurrent layers, number of multi-layer perceptron layers, hidden layer sizes, learning rate, dropout probability, and whether the embeddings are fixed during training or allowed to vary.

The dictionary specifying our final hyperparameter grid search was as follows:

```
hyperparam_grid = {'cell_type': ['VanillaRNN', 'GRU', 'LSTM'],
                   'num_layers': [1, 3],
                   'num_dense_layers': [1, 2],
                   'hidden_size': [128, 256],
                   'hidden_to_dense_size_ratio': [2], # applies when num_dense_layers > 1
                   'learning_rate': [0.001, 0.01],
                   'dropout': [0, 0.25], # applies when num_layers and/or num_dense_layers > 1
                   'fixed_embeddings': [True, False],
                   'batch_size': [128]
}
```

We selected this particular set of hyperparameters to grid-search over after an initial informal scoping of numerous combinations. After this initial investigation, we fixed the Adam optimiser to be used throughout this question as we found it to perform well when compared with other optimisers, and chose to reduce hidden layer sizes by a factor of two as you go through the network as this choice also appeared to return the best results.

The results in Table 9 show the top 50 models from this grid-search, measured and ordered by their validation set accuracy, from highest to lowest. This shows a number of patterns:

- Of the top 50 models, only one allowed the embedding weights to update. These models tended to overfit the training data, reaching their peak validation accuracy within the first couple of epochs.
- GRU and LSTM models are both well-represented in the table. The best vanilla RNN ranked 54th, with a validation set accuracy of 0.4124.
- The best models had the lower learning rate of 0.001.
- In terms of our particular set of hyperparameters, the hidden size, number of recurrent or dense layers, and dropout probability had a relatively small effect compared to the cell type and fixed embedding weights.

Hyperparameters							**Metrics**				
Cell type	**RNN layers**	**Hidden size**	**Dense layers**	**Learning rate**	**Dropout**	**Fixed embeddings**	**Best epoch**	**Best val acc**	**Assoc. train acc**	**Assoc. val loss**	**Assoc. train loss**
GRU	3	128	2	0.001	0.25	1	21	0.4505	0.4896	1.1033	1.1033
LSTM	3	256	2	0.001	0	1	30	0.4505	0.5455	1.1638	1.3451
LSTM	3	256	1	0.001	0	1	27	0.4487	0.529	1.1651	2.3536
GRU	1	128	2	0.001	0	1	16	0.4469	0.4993	0.9076	1.1668
GRU	3	256	1	0.001	0	1	11	0.4469	0.4751	1.3375	1.4611
LSTM	3	128	2	0.001	0	1	27	0.4432	0.5064	1.1583	1.4965
GRU	1	128	2	0.001	0.25	1	17	0.4414	0.4952	1.1106	1.6344
GRU	3	256	1	0.001	0.25	1	11	0.4405	0.4781	1.2116	1.2627
GRU	3	128	2	0.001	0	1	26	0.4387	0.588	1.2216	1.4029
GRU	1	256	2	0.01	0	1	6	0.4387	0.4854	1.2356	1.2014
GRU	3	256	2	0.001	0	1	11	0.4378	0.4783	1.0388	1.347
GRU	1	256	2	0.001	0.25	1	10	0.4378	0.4629	1.3178	1.1669
LSTM	1	128	1	0.01	0.25	1	6	0.4369	0.4812	1.1737	1.3881
GRU	3	128	1	0.001	0	1	13	0.436	0.4846	1.2251	0.9714
GRU	1	128	1	0.01	0	1	3	0.436	0.4363	1.2775	2.7006
GRU	1	128	2	0.01	0	1	10	0.436	0.5926	0.8751	1.9659
GRU	3	256	2	0.001	0.25	1	13	0.436	0.4703	1.0676	1.1432
LSTM	1	128	2	0.001	0	1	26	0.436	0.519	0.8957	1.5731
GRU	1	256	1	0.01	0	1	4	0.4351	0.461	1.2506	1.5046
LSTM	1	256	2	0.001	0	1	19	0.4351	0.4781	1.5481	1.6099
LSTM	3	256	1	0.001	0.25	1	18	0.4351	0.4706	1.2208	1.193
GRU	1	128	1	0.001	0	1	14	0.4342	0.4841	1.1685	1.1923
GRU	3	128	1	0.001	0.25	1	10	0.4342	0.4548	1.2729	1.1344
GRU	1	128	2	0.01	0.25	1	7	0.4342	0.5012	0.9056	1.381
LSTM	3	128	1	0.001	0.25	1	34	0.4332	0.533	1.111	1.212
LSTM	1	128	2	0.001	0.25	1	22	0.4323	0.4919	1.1299	1.5672
LSTM	3	128	1	0.01	0	1	12	0.4323	0.4751	1.2865	1.1957
GRU	3	128	1	0.01	0	1	9	0.4314	0.4745	0.7536	1.6324
GRU	1	128	1	0.001	0.25	1	13	0.4314	0.4689	1.4302	1.6635
LSTM	1	256	2	0.001	0.25	1	14	0.4296	0.4539	1.1424	1.2512
GRU	1	256	2	0.01	0.25	1	31	0.4287	0.5625	1.0536	4.2523
LSTM	1	256	2	0.01	0.25	1	10	0.4287	0.4216	1.3257	1.1481
LSTM	1	128	1	0.01	0	1	5	0.4287	0.4675	1.217	1.4673
LSTM	3	256	2	0.001	0.25	1	29	0.4287	0.5687	0.8473	1.5408
GRU	3	128	1	0.01	0.25	1	9	0.4278	0.4581	1.2957	1.5064
GRU	1	256	1	0.001	0.25	1	10	0.4278	0.4611	1.2809	1.3976
LSTM	1	128	1	0.001	0.25	1	24	0.4278	0.5219	1.1025	1.1691
LSTM	1	256	1	0.01	0.25	1	13	0.4278	0.4725	1.2413	0.7944
GRU	3	128	2	0.01	0	1	19	0.4269	0.5202	1.1038	1.298
GRU	1	256	2	0.001	0	1	13	0.426	0.4875	1.1886	1.4131
GRU	1	128	1	0.01	0.25	1	6	0.426	0.5459	0.9748	1.4814
LSTM	3	128	2	0.001	0.25	1	24	0.426	0.4622	1.5079	1.3763
GRU	1	256	1	0.001	0	1	12	0.4242	0.4718	0.5621	1.2012
LSTM	1	256	1	0.01	0	1	7	0.4242	0.4604	1.3112	1.2159
LSTM	1	128	2	0.01	0.25	1	5	0.4242	0.4232	1.3108	1.1252
LSTM	3	128	1	0.001	0	1	22	0.4223	0.4923	1.2071	1.179
LSTM	1	256	2	0.01	0	1	9	0.4214	0.4329	1.2253	1.2675
LSTM	3	128	1	0.01	0.25	1	31	0.4205	0.4647	1.1916	1.3396
GRU	3	128	1	0.001	0	0	1	0.4187	0.4347	1.1729	1.7696
GRU	1	256	1	0.01	0.25	1	6	0.4187	0.5829	1.0295	1.7597

Table 9: Results of hyperparameter grid search for RNN model variants trained on the Stanford Sentiment Treebank dataset (top 50 results shown, ordered by best validation accuracy)

5.3 Part 3: Final model training

We trained each model in the hyperparameter grid-search to convergence on the validation dataset, implementing an early stopping criterion which defines convergence as being when the mean validation loss from the last 15 epochs is greater than the mean validation loss from the preceding 15 epochs. We set the maximum number of epochs before early stopping could be triggered to 20, and the maximum number of epochs to 100.

Table 9 shows that there are two models with the highest validation accuracy of 0.4505, so we choose the one with the lowest validation loss as our final model. The hyperparameters of this model were:

Hyperparameter	**Value**
Cell type	GRU
Number of recurrent layers	3
Number of dense layers	2
Hidden layer size	128
Dropout	0.25
Fixed embeddings	True
Learning rate	0.001

Table 10: The hyperparameters for the best performing recurrent neural network model

5.4 Part 4: Accuracy and loss plots

The training and validation accuracy and loss plots for our best model are shown in Figure 13 and Figure 14 respectively. Early stopping criteria used were as described in section Q5 Part 3.

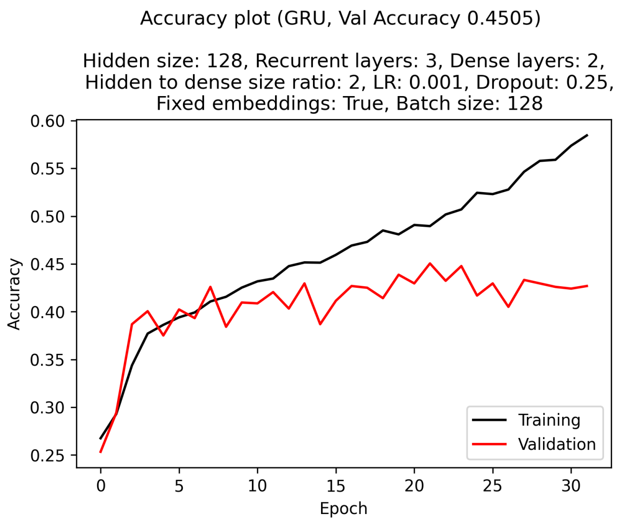


Figure 13: Training and validation accuracy by epoch for the best RNN model

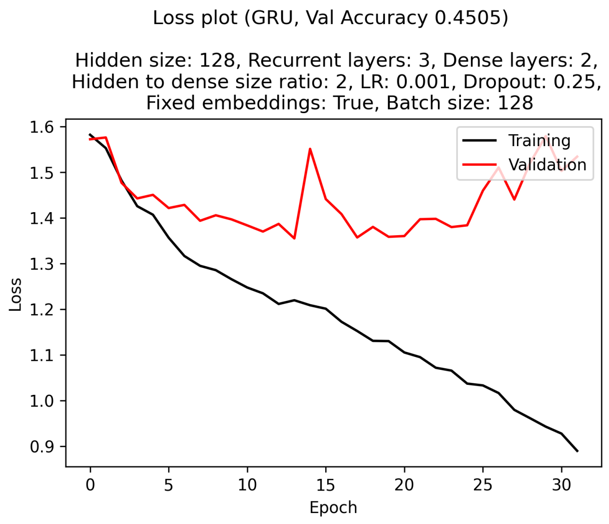


Figure 14: Training and validation loss by epoch for the best RNN model

5.5 Part 5: Best model final metrics

As shown in Table 9, our best model reached its highest accuracy of 0.4504 on the validation dataset after 21 epochs of training. This epoch was associated with an accuracy of 0.4896 on the training dataset. Final evaluation of the model was performed on the test dataset, achieving an accuracy of 0.4394.

Dataset	**Accuracy**
Training	0.4896
Validation	0.4504
Test	0.4394

Table 11: The training, validation, and test accuracy at the best epoch (21) for the best RNN model

5.6 Part 6: Online review classifications

We obtained a selection of online reviews with a mixture of positive, neutral, and negative sentiments, and pre-processed these according to the same rules as the training, validation, and test datasets.

We then fed these into our best model to obtain the following classification decisions.

> To this day, this is still my favorite pixar film. The animation is stellar, its heartwarming, funny and proves that pixar movies are always bound to be great (except for cars 2 but thats a different story). This has a shot at the title ”best movie of the century”
>
> `https://www.imdb.com/review/rw5485122`

Prediction: 4

> This is just a wonderful telling of Charles Dickens great Christmas story. The story being so good, you would have to try had to make a bad movie out of it.
>
> `https://www.imdb.com/review/rw0310420`

Prediction: 4

> Honestly, I really should be giving this film a lower score. Somehow I enjoyed it quite a bit even in the face of the many fundamental issues, which is a testament to the strength of the best sequences.
>
> `https://www.imdb.com/review/rw4075393`

Prediction: 4

2 1/2 hours of Boredom. Half the audience fell asleep, including most of the kiddies. Beautiful to look at, but that does not make for a interesting film. Rather spend your money on Lord of the Rings.

`https://www.imdb.com/review/rw0717356/`

Prediction: 3

But the worst thing of all with this film is the mangling of Austen's dialogue and the atrocious modern dialogue. Austen's dialogue needs no assistance from a writer who thinks he/she can write like Austen.

`https://www.imdb.com/review/rw1213354/`

Prediction: 0

Appendix A Q1 code

```
import numpy as np

import torch
import torchvision
import torchvision.transforms as transforms

import torch.nn as nn
import torch.nn.functional as F
import torch.optim as optim
from torch.utils.data.sampler import SubsetRandomSampler

import matplotlib.pyplot as plt
%matplotlib inline

if torch.cuda.is_available():
    device = torch.device('cuda:0')
    print('GPU on')
else:
    device = torch.device('cpu')
    print('Using CPU')


#Download the training and testing data sets and transform to tensor
train_set = torchvision.datasets.FashionMNIST(
    "",
    train=True,
    download=True,
    transform=transforms.Compose([
        transforms.ToTensor()
    ])
)

test_set = torchvision.datasets.FashionMNIST(
    "",
    train=False,
    download =True,
    transform = transforms.Compose([
        transforms.ToTensor()
    ])
)

# obtain training indices that will be used for validation
train_size = len(train_set)
indices = list(range(train_size))
np.random.shuffle(indices)
#Validation size of 20%
split = int(np.floor(train_size * 0.2))
train_index, valid_index = indices[split:], indices[:split]

# define samplers for obtaining training and validation batches
train_sampler = SubsetRandomSampler(train_index)
validation_sampler = SubsetRandomSampler(valid_index)

# prepare data loaders
```

```
train_set_loader = torch.utils.data.DataLoader(train_set, batch_size = 128,
                                           sampler = train_sampler, num_workers = 1)
validation_set_loader = torch.utils.data.DataLoader(train_set, batch_size = 128,
                                          sampler = validation_sampler, num_workers = 1)
test_set_loader = torch.utils.data.DataLoader(test_set, batch_size = 128,
                                         num_workers = 2)


print('Training dataset size:', len(train_set))
print(train_set.targets)
print(train_set.targets.bincount())

#Obtain a batch
batch = next(iter(train_set_loader))

#The batch is a tuple. It ocntains our image tensors in one list and the labels in another.
↪  Unpack them
images, labels = batch

#Pytorch function to create an image grid
grid = torchvision.utils.make_grid(images, nrow=6)

#Show images
plt.figure(figsize=(10,10))
plt.imshow(np.transpose(grid, (1, 2, 0)))

print('labels:', labels)

class VariableNN(nn.Module):

    def __init__(self, num_of_layers, Dropout, N, O, P):
        super().__init__()
        #Decide number of layers
        self.num_of_layers = num_of_layers
        #Dropout toggle and function, use standard p = 0.7
        self.Dropout = Dropout
        self.dropout = nn.Dropout(p=0.7)
        #Node elements
        self.N = N
        self.O = O
        self.P = P
        #Fully connected layers
        self.fc1 = nn.Linear(28*28, N)
        torch.nn.init.xavier_normal_(self.fc1.weight)
        torch.nn.init.zeros_(self.fc1.bias)

        self.fc2 = nn.Linear(N, O)
        torch.nn.init.xavier_normal_(self.fc2.weight)
        torch.nn.init.zeros_(self.fc2.bias)

        self.fc3a = nn.Linear(O, P)
        torch.nn.init.xavier_normal_(self.fc3a.weight)
        torch.nn.init.zeros_(self.fc3a.bias)

        self.fc3b = nn.Linear(O, 10)
        torch.nn.init.xavier_normal_(self.fc3b.weight)
        torch.nn.init.zeros_(self.fc3b.bias)

        self.fc4 = nn.Linear(P, 10)
        torch.nn.init.xavier_normal_(self.fc4.weight)
        torch.nn.init.zeros_(self.fc4.bias)

    #Define Forwards Pass
    def forward(self, x):
        if self.Dropout == 0:
            if self.num_of_layers == 4:
                # flatten image input
                x = x.view(-1,28*28)
                x = self.fc1(x)
                x = F.relu(x)
                x = self.fc2(x)
                x = F.relu(x)
                x = self.fc3a(x)
                x = F.relu(x)
                x = self.fc4(x)

            elif self.num_of_layers == 3:
                # flatten image input
                x = x.view(-1,28*28)
                x = self.fc1(x)
                x = F.relu(x)
                x = self.fc2(x)
                x = F.relu(x)
                x = self.fc3b(x)

        elif self.Dropout == 1:
            if self.num_of_layers == 4:
                # flatten image input
                x = x.view(-1,28*28)
                x = self.fc1(x)
                x = F.relu(x)
                x = self.dropout(x)
                x = self.fc2(x)
                x = F.relu(x)
                x = self.dropout(x)
                x = self.fc3a(x)
                x = F.relu(x)
                x = self.dropout(x)
                x = self.fc4(x)

            elif self.num_of_layers == 3:
                # flatten image input
                x = x.view(-1,28*28)
                x = self.fc1(x)
                x = F.relu(x)
                x = self.dropout(x)
                x = self.fc2(x)
                x = F.relu(x)
                x = self.dropout(x)
                x = self.fc3b(x)

        return x
```

```
#Cross entropy loss
loss_function = nn.CrossEntropyLoss()


def train_network(train_loader, test_loader, num_of_layers, dropout, nodes1, nodes2, nodes3,
↪   epochs):
    #Our NN
    net = VariableNN(num_of_layers, dropout, nodes1, nodes2, nodes3)
    optimizer = optim.Adam(net.parameters(), lr=0.01)
    train_losses = []
    train_accuracies = []

    valid_losses = []
    valid_accuracies = []


    for epoch in range(epochs):  # loop over the dataset multiple times

        # monitor losses
        running_train_loss = 0
        running_train_accuracy = 0
        train_batches = 0

        net.train()
        for data, label in train_loader:
            # clear the gradients of all optimized variables
            optimizer.zero_grad()
            # forward pass: compute predicted outputs by passing inputs to the model
            output = net(data)
            # calculate the loss
            loss = loss_function(output,label)
            # backward pass: compute gradient of the loss with respect to model parameters
            loss.backward()
            # perform a single optimization step (parameter update)
            optimizer.step()
            # update running training loss
            running_train_loss += loss.item() #* data.size(0)
            # get accuracies
            _, predictions = torch.max(output.data, 1)
            correct_predictions = (predictions.int() == label.int()).sum().numpy()
            length = label.size()[0]
            running_train_accuracy += correct_predictions/length

            train_batches += 1

        train_losses.append(running_train_loss/train_batches)
        train_accuracies.append(running_train_accuracy/train_batches)

        running_valid_loss = 0
        running_valid_accuracy = 0
        valid_batches = 0


        net.eval()
        for data, label in test_loader:
```

```
        # forward pass: compute predicted outputs by passing inputs to the model
        output = net(data)
        # calculate the loss
        loss = loss_function(output,label)
        # update running validation loss
        running_valid_loss += loss.item() #* data.size(0)
        # get accuracies
        _, predictions = torch.max(output.data, 1)
        correct_predictions = (predictions.int() == label.int()).sum().numpy()
        length = label.size()[0]
        running_valid_accuracy += correct_predictions/length

        valid_batches += 1

    valid_losses.append(running_valid_loss / valid_batches)
    valid_accuracies.append(running_valid_accuracy / valid_batches)

    if epoch % 10 == 0:
        print('Epoch: {}/{}, training loss: {:05.3f}, training accuracy: {:05.3f},
        ↪  validation loss: {:05.3f}, validation accuracy: {:05.3f}'.format(epoch,
        ↪  epochs, train_losses[-1],
                                                            train_accuracies[-1],
                                                            ↪  valid_losses[-1],
                                                            valid_accuracies[-1])
             )


    # save model if validation loss has decreased
    if epoch > 20:
        if (np.mean(valid_losses[-10:]) - np.mean(valid_losses[-20:-10])) > 0.01 or epoch
        ↪  == epochs:
            print('The model has converged')
            torch.save(net.state_dict(), 'net.pt')

            train_loss_min = train_losses[-1]
            train_acc_min = train_accuracies[-1]
            valid_loss_min = valid_losses[-1]
            valid_acc_min = valid_accuracies[-1]
            print('Convergence Epoch: {}. \nFinal training loss and accuracy: {:05.3f},
            ↪  {:05.3f}. \nFinal validation loss and accuracy: {:05.3f},
            ↪  {:05.3f}'.format(epoch, train_loss_min,
                                                              train_acc_min,
                                                              valid_loss_min,
                                                              valid_acc_min)
                 )
            break


# plot the loss
plt.plot(np.squeeze(train_losses), color='k', linestyle='-')
plt.plot(np.squeeze(valid_losses), color='r', linestyle='-')
plt.legend(['Training', 'Validation'], loc='upper left')
plt.ylabel('Loss')
plt.xlabel('Iteration')
plt.title("Loss Plot for Training and Validation Set")
plt.show()
```

```
    # plot the accuracy
    plt.plot(np.squeeze(train_accuracies), color='k')
    plt.plot(np.squeeze(valid_accuracies), color='r')
    plt.legend(['Training', 'Validation'], loc='upper left')
    plt.ylabel('Accuracy')
    plt.xlabel('iteration')
    plt.title("Accuracy Plot for Training and Validation Set")
    plt.show()

    final_cost = valid_losses[-1]


    return final_cost, net

dropouts = [0 , 1]
depths = [3, 4]
Ns = [100, 200]
Os = [50, 100]
Ps = [25, 50]

epochs = 5000

results = {}
## loop through dropouts
for i,depth in enumerate(depths,0):
  #Loop through size:
  for j,dropout in enumerate(dropouts, 0):
    #Loop through 1st layer number of nodes:
    for k, N in enumerate(Ns, 0):
        for l, O in enumerate(Os, 0):
            for m, P in enumerate(Ps, 0):
                drops = str(bool(j))

                key = "Dropout = {0!s}, number of layers = {1},  Layer 1 = {2} , Layer2 = {3},
                ↪  Layer3 = {4}".format(drops, depth, N, O, P)

                print('\nCROSS VALIDATION for Dropouts = %s, number of layers = %d, where
                ↪  Layer 1 = %d , Layer2 = %d and Layer3 = %d'
                      %(drops, depth, N, O, P)
                     )

                ## add the right arguments for function train_network
                final_cost, _ = train_network(train_set_loader, validation_set_loader, depth,
                ↪  dropout, N, O, P, epochs)


#Final MLP Architecture
class FinalNet(nn.Module):
    def __init__(self):
        super().__init__()
        self.dropout = nn.Dropout(p=0.5)

        self.fc1 = nn.Linear(28*28, 200)
        torch.nn.init.xavier_normal_(self.fc1.weight)
        torch.nn.init.zeros_(self.fc1.bias)
        self.fc2 = nn.Linear(200, 50)
        torch.nn.init.xavier_normal_(self.fc2.weight)
        torch.nn.init.zeros_(self.fc2.bias)

        self.fc3 = nn.Linear(50, 50)
        torch.nn.init.xavier_normal_(self.fc3.weight)
        torch.nn.init.zeros_(self.fc3.bias)

        self.fc4 = nn.Linear(50, 10)
        torch.nn.init.xavier_normal_(self.fc4.weight)
        torch.nn.init.zeros_(self.fc4.bias)

    def forward(self, x):
        x = x.view(-1,28*28)
        x = F.relu(self.fc1(x))
        x = F.relu(self.fc2(x))
        x = F.relu(self.fc3(x))
        x = self.fc4(x)
        return F.log_softmax(x, dim=1)




#Final training loop
net = FinalNet()
print(net)

#maximum number of epochs to train the model
epochs = 5000

train_losses = []
train_accuracies = []

valid_losses = []
valid_accuracies = []

for epoch in range(epochs):

    #monitor loss and accuracy
    running_train_loss = 0
    running_train_accuracy = 0
    train_batches = 0


    net.train() # prep model for training
    for data,label in train_set_loader:
        #Clear the gradients of all optimized variables
        optimizer.zero_grad()
        #Forward pass
        output = net(data)
        #Loss
        loss = loss_function(output,label)
        #Backward pass
        loss.backward()
        #Parameter update
        optimizer.step()
        #Running training loss
        running_train_loss += loss.item()
        #Running training accuracies
        _, predictions = torch.max(output.data, 1)
        correct_predictions = (predictions.int() == label.int()).sum().numpy()
        length = label.size()[0]
        running_train_accuracy += correct_predictions/length

        train_batches += 1

    #Total training loss and accuracy for epoch
    train_losses.append(running_train_loss/train_batches)
    train_accuracies.append(running_train_accuracy/train_batches)


    running_valid_loss = 0
    running_valid_accuracy = 0
    valid_batches = 0


    net.eval()  # prep model for evaluation
    for data,label in validation_set_loader:
        output = net(data)
        loss = loss_function(output,label)
        running_valid_loss += loss.item()
        _, predictions = torch.max(output.data, 1)
        correct_predictions = (predictions.int() == label.int()).sum().numpy()
        length = label.size()[0]
        running_valid_accuracy += correct_predictions/length

        valid_batches += 1

    valid_losses.append(running_valid_loss / valid_batches)
    valid_accuracies.append(running_valid_accuracy / valid_batches)

    if epoch % 10 == 0:
        print('Epoch: {}/{}, training loss: {:05.3f}, training accuracy: {:05.3f}, validation
            ↪  loss: {:05.3f}, validation accuracy: {:05.3f}'.format(epoch, epochs,
            ↪  train_losses[-1],
                                                            train_accuracies[-1],
                                                             ↪  valid_losses[-1],
                                                            valid_accuracies[-1])
             )

    # save model if validation loss has decreased
    if epoch > 20:
        if (np.mean(valid_losses[-15:]) - np.mean(valid_losses[-30:-15])) > 0.01 or epoch ==
         ↪  epochs:
            print('The model has converged')
            torch.save(net.state_dict(), 'net.pt')

            train_loss_min = train_losses[-1]
            train_acc_min = train_accuracies[-1]
            valid_loss_min = valid_losses[-1]
            valid_acc_min = valid_accuracies[-1]
```

```
            print('Convergence Epoch: {}. \nFinal training loss and accuracy: {:05.3f},
            ↪  {:05.3f}. \nFinal validation loss and accuracy: {:05.3f},
            ↪  {:05.3f}'.format(epoch, train_loss_min,
                                                              train_acc_min,
                                                              valid_loss_min,
                                                              valid_acc_min)
                 )
            break

# plot the loss
plt.plot(np.squeeze(train_losses), color='k', linestyle='-')
plt.plot(np.squeeze(valid_losses), color='r', linestyle='-')
plt.legend(['Training', 'Validation'], loc='upper left')
plt.ylabel('Loss')
plt.xlabel('Iteration')
plt.title("Loss Plot for Training and Validation Set")
plt.show()

# plot the accuracy
plt.plot(np.squeeze(train_accuracies), color='k')
plt.plot(np.squeeze(valid_accuracies), color='r')
plt.legend(['Training', 'Validation'], loc='upper left')
plt.ylabel('Accuracy')
plt.xlabel('iteration')
plt.title("Accuracy Plot for Training and Validation Set")
plt.show()
```

Appendix B Q2 code

```
import numpy as np
import torchvision
import torchvision.transforms as transforms
import torch
import torch.nn as nn
import torch.optim as optim
import matplotlib.pyplot as plt
from matplotlib import gridspec
import random
from tqdm import tqdm
from torch.utils.data.sampler import SubsetRandomSampler
from sklearn.model_selection import ParameterGrid

trainset = torchvision.datasets.FashionMNIST(root = "./data", train = True, download = True,
↪   transform = transforms.ToTensor())
testset = torchvision.datasets.FashionMNIST(root = "./data", train = False, download = True,
↪   transform = transforms.ToTensor())

#loading the training & test data from trainset
trainloader = torch.utils.data.DataLoader(trainset, shuffle = True)
testloader = torch.utils.data.DataLoader(testset, shuffle = False)

# Obtain training indices that will be used for validation
train_size = len(trainset)
indices = list(range(train_size))
np.random.shuffle(indices)

# Validation size of 20%
split = int(np.floor(train_size * 0.2))
train_index, valid_index = indices[split:], indices[:split]

# Define samplers for obtaining training and validation
train_sampler = SubsetRandomSampler(train_index)
validation_sampler = SubsetRandomSampler(valid_index)

# prepare data loaders
train_set_loader = torch.utils.data.DataLoader(trainset, sampler = train_sampler, num_workers
↪   = 2)
validation_set_loader = torch.utils.data.DataLoader(trainset, sampler = validation_sampler,
↪   num_workers = 2)

def noise(x):
    q = random.choice([1, 2])
    for i in range(q):
        xoffset = random.choice([0, 14])
        yoffset = random.choice([0, 14])
        for i in range(xoffset, xoffset+14):
            for j in range(yoffset, yoffset+14):
                x[:, :, i, j] = 0
    return x

for xb, yb in train_set_loader:
    x = noise(xb)
    x = x.view(28, 28)
    break

plt.imshow(x, cmap="gray")

# Train Dataset
traindata_clean_x = np.zeros((len(train_set_loader), 28, 28))
traindata_x = np.zeros((len(train_set_loader), 28, 28))

traindata_y = np.zeros((len(train_set_loader), 1))

# Validation Dataset
validationdata_clean_x = np.zeros((len(validation_set_loader), 28, 28))
validationdata_x = np.zeros((len(validation_set_loader), 28, 28))

index = 0
for i in tqdm(iter(train_set_loader)):
    (x, y) = i
    traindata_clean_x[index] = x.numpy()

    x_noise = torch.reshape(noise(x), (28, 28))
    traindata_x[index] = x_noise.numpy()

    traindata_y[index] = y.numpy()
    index += 1

index = 0
for i in tqdm(iter(validation_set_loader)):
  (x, y) = i
  validationdata_clean_x[index] = x.numpy()

  x_noise = torch.reshape(noise(x), (28, 28))
  validationdata_x[index] = x_noise.numpy()

  index += 1

# Test Dataset
testdata_clean_x = np.zeros((len(testloader), 28, 28))
testdata_x = np.zeros((len(testloader), 28, 28))
testdata_y = np.zeros((len(testloader), 1))

index = 0
for i in tqdm(iter(testloader)):
  (x, y) = i
  testdata_clean_x[index] = x.numpy()
  x_noise = torch.reshape(noise(x), (28, 28))
  testdata_x[index] = x_noise.numpy()

  index += 1

# Custom PyTorch dataset
class create_dataset(torch.utils.data.Dataset):

  # Pass a noise x, and a clean x, aswell as a tranform if needed
  # Functions follow the PyTorch Dataset class pattern
  def __init__(self, noiseX, cleanX, transform):
    self.noise = noiseX
    self.clean = cleanX
    self.transform = transform

  def __len__(self):
    return len(self.noise)

  # Returns a noise, clean
  def __getitem__(self,idx):
    xNoise = self.noise[idx]
    xClean = self.clean[idx]

    if self.transform != None:
        xNoise = self.transform(xNoise)
        xClean = self.transform(xClean)

    return (xNoise, xClean)

tsfms=transforms.Compose([
    transforms.ToTensor()
])

train_dataset = create_dataset(traindata_x, traindata_clean_x, tsfms)
validation_dataset = create_dataset(validationdata_x, validationdata_clean_x, tsfms)
test_dataset = create_dataset(testdata_x, testdata_clean_x, tsfms)

test_dataset_loader = torch.utils.data.DataLoader(test_dataset, shuffle = True)

class AutoEncoder(nn.Module):
    def __init__(self, structure):
        super().__init__()

        encoder_layers = []
        decoder_layers = []

        # Encoder
        for l in structure:
          encoder_layers.append(nn.Linear(l[0], l[1]))
          encoder_layers.append(nn.ReLU(True))

        # Decoder
        for index, l in reversed(list(enumerate(structure))):
          decoder_layers.append(nn.Linear(l[1], l[0]))
          # Last index, thus need Sigmoid
          if (index == (0)):
            decoder_layers.append(nn.Sigmoid())
          else:
            decoder_layers.append(nn.ReLU(True))

        self.encoder_struct = nn.Sequential(*encoder_layers)
        self.decoder_struct = nn.Sequential(*decoder_layers)

    def forward(self, x):
        x = self.encoder_struct(x)
        x = self.decoder_struct(x)

        return x
```

```
if torch.cuda.is_available() == True:
    print("Using CUDA")
    device="cuda:0"
else:
    print("Using CPU")
    device ="cpu"

"""
hyperparam_grid = {
  'structure': [
    [[28*28, 512], [512, 256], [256, 128], [128, 64], [64, 32], [32, 16]],
    [[28*28, 256], [256, 128], [128, 64], [64, 32], [32, 16]],
    [[28*28, 256], [256, 128], [128, 64]]
    [[28*28, 128], [128, 64], [64, 32], [32, 16]],
    [[28*28, 64], [64, 32], [32, 16]],
  ],
  'batch_size': [16, 32, 64, 128, 256],
  'learning_rate': [0.01, 0.001, 0.0001]
}
"""

hyperparam_grid = {
  'structure': [
    [[28*28, 256], [256, 128], [128, 64]],
  ],
  'batch_size': [64],
  'learning_rate': [0.001]
}

for run_index, hyperparams in enumerate(ParameterGrid(hyperparam_grid)):
  train_dataset_loader = torch.utils.data.DataLoader(train_dataset, batch_size =
   ↪  hyperparams['batch_size'], shuffle = True)
  validation_dataset_loader = torch.utils.data.DataLoader(validation_dataset, batch_size =
   ↪  hyperparams['batch_size'], shuffle = True)

  model = AutoEncoder(hyperparams['structure']).to(device)
  c = nn.MSELoss()
  optimizier = optim.Adam(model.parameters(), lr = hyperparams['learning_rate'])

  current_epoch = 0
  max_epochs = 10000
  min_epochs = 5
  running_loss = 0
  running_validation_loss = 0
  epoch_loss = 0

  best_loss = 100000

  loss_list = []
  validation_loss_list = []
  converged = False

  while (converged == False) and (current_epoch <= max_epochs):
    current_epoch += 1
    model.train()
    for xd, xcd in tqdm((train_dataset_loader)):
      x = xd.view(xd.size(0), -1).type(torch.FloatTensor)
      xc = xcd.view(xcd.size(0), -1).type(torch.FloatTensor)
      x, xc = x.to(device), xc.to(device)

      # Forward
      output = model(x)
      loss = c(output, xc)

      # Backward
      optimizier.zero_grad()
      loss.backward()
      optimizier.step()

      running_loss += loss.item()

    # Now handle validation testing
    model.eval()
    with torch.no_grad():
      for vd, vcd in tqdm(validation_dataset_loader):
        v = vd.view(vd.size(0), -1).type(torch.FloatTensor)
        vc = vcd.view(vcd.size(0), -1).type(torch.FloatTensor)
        v, vc = v.to(device), vc.to(device)

        v_output = model(v)
        v_loss = c(v_output, vc)

        running_validation_loss += v_loss.item()

    loss_list.append(running_loss / len(train_dataset_loader))
    validation_loss_list.append(running_validation_loss / len(validation_dataset_loader))

    if (running_validation_loss / len(validation_dataset_loader) < best_loss):
      best_loss = running_validation_loss / len(validation_dataset_loader)
      best_model = model

    running_loss = 0
    running_validation_loss = 0

    #print("=> epoch ", current_epoch, "Loss: ", loss.item(), " Validation Loss: ",
    ↪  v_loss.item())

    # Check for early stopping
    if current_epoch >= min_epochs:
      if (np.mean(validation_loss_list[-3:]) - np.mean(validation_loss_list[-6:-3])) > -.001:
        converged = True
        print("Model Converged")
        print("Epoch ", current_epoch, " Validation Loss: ", v_loss.item())
        print(hyperparams)

  # save the model
  print(best_model)
  torch.save(best_model, f"best_model.pth")

plt.plot(range(len(loss_list)),loss_list, label='Training loss')
plt.plot(range(len(validation_loss_list)), validation_loss_list, label='Validation loss')
plt.legend()
plt.show()

nrow = 32
ncol = 3

fig = plt.figure(figsize=((ncol+1)*2, (nrow+1)*2))

gs = gridspec.GridSpec(nrow, ncol,
         wspace=0.2, hspace=0.2,
         top=1.-0.5/(nrow+1), bottom=0.5/(nrow+1),
         left=0.5/(ncol+1), right=1-0.5/(ncol+1))

for i in range(nrow):
  for xd, xcd in test_dataset_loader:
    output_x = xd[0].view(xd[0].size(0), -1).type(torch.FloatTensor)
    output_x = output_x.to(device)
    output = model(output_x)

    output = output.view(1, 28, 28)
    output = output.detach().cpu().numpy()

    ax= plt.subplot(gs[i,0]).imshow(xcd[0].view(28, 28), cmap='gray')
    ax2= plt.subplot(gs[i,1]).imshow(xd[0].view(28, 28), cmap='gray')
    ax3= plt.subplot(gs[i,2]).imshow(output[0], cmap='gray')

    break

plt.show()
```

Appendix C Q3 code

```
# SETUP ENVIRONMENT

# install pytorch

!pip3 install torch


# import relevant libraries

import numpy as np
import pandas as pd
import matplotlib.pyplot as plt

import torch
import torch.nn as nn
import torch.optim as optim
import torch.nn.functional as F

import re

from sklearn.model_selection import ParameterGrid


# Mount google drive
from google.colab import drive
drive.mount('/content/drive')

# Set path for output files
path = "/content/drive/My
↪  Drive/Machine_learning/UCL/Modules/DL/dl_coursework2/Q3_saved_models/"


# LOAD AND PREPARE DATASETS

# load stanford sentiment treebank dataset
# site for more information on data set: https://nlp.stanford.edu/sentiment/

!wget https://nlp.stanford.edu/sentiment/trainDevTestTrees_PTB.zip
!unzip trainDevTestTrees_PTB.zip

%cd "/content/trees/"


# functions for use in rest of notebook

def parse_tree(sentence_list_tree):
    """
    Function for extracting text only from sentiment tree (remove tree structure)
    Input: list of strings of trees of sentences/lines
    Output: x_list is a list of sentences, y_list is a list of sentence sentiment classes
    """
    # remove empty strings from list
    while("" in sentence_list_tree):
        sentence_list_tree.remove("")
    # initialise lists to store x (the sentence), y (the sentiment)
    x_list = []
    y_list = []
    for sentence in sentence_list_tree:
        y_list.append(int(sentence[1])) # y = sentiment class of sentence
        # remove digits and parentheses using regular expression
        patterns = r'[0-9]|\(|\)'
        sentence = re.sub(patterns, '', sentence)
        # remove -RRB- and -LRB-
        sentence = re.sub(r"-RRB-", "", sentence)
        sentence = re.sub(r"-LRB-", "", sentence)
        # remove duplicate spaces
        sentence = " ".join(sentence.split())
        # remove spaces before some types of punctuation
        sentence = re.sub(r"\s([?.!;\,](?:\s|$))", r'\1', sentence)
        # remove spaces before apostrophes
        sentence = re.sub(r" '\b", "'", sentence)
        # remove spaces before n't
        sentence = re.sub(r" n't\b", "n't", sentence)
        # remove space before ...
        sentence = re.sub(r" \.\.\.", r"...", sentence)
        # make all lower case
        sentence = sentence.lower()
        # remove slashes (replace with space)
        sentence = sentence.replace("\\"," ")

        # remove dashes (replace with space)
        sentence = re.sub("-", " ", sentence)
        # remove all other punctuation / non-standard characters
        sentence = re.sub('[¼¶´³±¯\xad©¨§¦£¢¡!"#$%&()*+,/:;<=>?@[\]^_`{|}~]', "", sentence)
        # remove double quotation marks
        sentence = re.sub("''", "", sentence)
        # remove single quotation marks (space beforehand so as to not replace apostrophes)
        sentence = re.sub(" '", " ", sentence)
        # remove duplicate spaces
        sentence = " ".join(sentence.split())

        # replace letters with accents, with standard non-accented english letters
        sentence = re.sub('[àáâã]', "a", sentence)
        sentence = re.sub('[æ]', "ae", sentence)
        sentence = re.sub('[ç]', "c", sentence)
        sentence = re.sub('[èé]', "e", sentence)
        sentence = re.sub('[íï]', "i", sentence)
        sentence = re.sub('[ñ]', "n", sentence)
        sentence = re.sub('[óôö]', "o", sentence)
        sentence = re.sub('[ûü]', "u", sentence)

        x_list.append(sentence)

    return x_list, y_list


# load and parse string data

train_file = open("train.txt", "r")
```

```
train_tree = train_file.read()
train_tree_sentence_list = train_tree.split("\n")
x_train_list, y_train_list = parse_tree(train_tree_sentence_list)

dev_file = open("dev.txt","r")
dev_tree = dev_file.read()
dev_tree_sentence_list = dev_tree.split("\n")
x_dev_list, y_dev_list = parse_tree(dev_tree_sentence_list)

test_file = open("test.txt", "r")
test_tree = test_file.read()
test_tree_sentence_list = test_tree.split("\n")
x_test_list, y_test_list = parse_tree(test_tree_sentence_list)


# DISPLAY DATA

# display example data for training set

start_index = 0
end_index = 30

for sentiment, sentence in
↪  zip(y_train_list[start_index:end_index],x_train_list[start_index:end_index]):
    print(f"y = {sentiment} | x = ({sentence})")


# display number of sentences in each dataset

print(f"Number of sentences training set: {len(x_train_list)}")
print(f"Number of sentences validation set: {len(x_dev_list)}")
print(f"Number of sentences test set: {len(x_test_list)}")


# merge sentences into a single long string
# this is so that we can pull out strings of random length (within specified range)
# to train/validate/test our character completion model

train_merged_string = ""
for sentence in x_train_list:
    train_merged_string += sentence + " "

dev_merged_string = ""
for sentence in x_dev_list:
    dev_merged_string += sentence + " "

test_merged_string = ""
for sentence in x_test_list:
    test_merged_string += sentence + " "


# ONE-HOT-ENCODING OF CHARACTERS

# specify functions for switching between onehot and index and text forms of data

def index_to_onehot_singlecharacter(index, number_unique_characters):
    """Convert a single index value corresponding to a single character into a one-hot
    ↪  vector"""

    one_hot_vector = torch.zeros(number_unique_characters)
    one_hot_vector[index] = 1

    return one_hot_vector

def onehot_to_index_singlecharacter(onehot_vector):
    """Convert a single one-hot list or numpy array corresponding to a single character into a
    ↪  single scalar index"""

    return torch.argmax(onehot_vector)

def test_charindex_mapping(line, char_to_index_dict, index_to_char_dict):
    """
    Check that the character-index mapping is working as expected
    Input: string of a single line
    """

    line_indices = []
    for char in line:
        line_indices.append(char_to_index.get(char))

    line_characters = ""
    for index in line_indices:
        line_characters += str(index_to_char.get(index))

    if line == line_characters:
        print("character-index and index-character mapping: WORKING")
    else:
        print("character-index or index-character mapping: NOT WORKING")


def text_to_onehot_multiline(line_or_lines, char_to_index_dict, number_unique_characters):
    """
    Convert each line in a list of lines to its onehot encoded form
    Input: line_or_lines = the dataset (list of lines, or a single line) to onehot encode
           number_unique_characters = how many unique characters in the full dataset
           char_to_index_dict = dictionary of character to index mapping for all unique
↪   characters
    Output: list of onehot encoded lines
            Where each element of list contains one line, in matrix onehot form
            Where each row in the matrix corresponds to a onehot encoding of a character
    Dependencies: Depends on index_to_onehot_singlecharacter function
    """

    # if a single line is passed to line_or_lines rather than a list:
    if isinstance(line_or_lines, str):
        line = line_or_lines
        char_indices = []
        for char in line:
            char_indices.append(char_to_index.get(char))

        line_onehot_matrix = torch.zeros((1, len(char_indices), number_unique_characters))
        for i, index in enumerate(char_indices):
            line_onehot_matrix[0, i,:] = index_to_onehot_singlecharacter(index,
            ↪  number_unique_characters)

        data_onehot = line_onehot_matrix

    # if a list of lines are passed to line_or_sline:
    else:
        data_onehot = []
        # loop through each line in the list
        for line in line_or_lines:

            # loop through each character in the line and append index to char_indices
            char_indices = []
            for char in line:
                char_indices.append(char_to_index.get(char))

            line_onehot_matrix = torch.zeros((len(char_indices), number_unique_characters))
            for i, index in enumerate(char_indices):
                line_onehot_matrix[i,:] = index_to_onehot_singlecharacter(index,
                ↪  number_unique_characters)

            data_onehot.append(line_onehot_matrix)
        data_onehot = torch.stack(data_onehot)

    return data_onehot


def onehot_to_text_singleline(onehot_line_matrix, index_to_char_dict):
    """
    Convert a onehot matrix version of a line back into text
    Input: onehot_line_matrix = the onehot matrix corresponding to a line (each row is onehot
↪   vector for a single character)
           index_to_char_dict = dictionary of index to character mapping for all unique
↪   characters
    Output: string of line
    Dependencies: Depends on onehot_to_index_singlecharacter function
    """

    num_characters_in_line = onehot_line_matrix.shape[0]
    line = ""

    for row_index in range(num_characters_in_line):
        char_index = onehot_to_index_singlecharacter(onehot_line_matrix[row_index]).item()
        line += index_to_char_dict.get(char_index)

    return line


# Specify function for generating random minibatch of data

def random_xy_batch(merged_string, seq_length_range=(20,100), batch_size=1):
  """
  Input:
  merged_string - a string of the full dataset from which to draw examples
  seq_length_range - tuple to specify range of string lengths of each output training example
  batch_size - number of string examples to generate
  Ouptut:
  X - a string or list of strings randomly drawn from the dataset, of specified length
↪  seq_length (the input)
  y - the letter that followed each string in X (the target)
  """

  dataset_length = len(merged_string)

  seq_length = np.random.randint(seq_length_range[0], seq_length_range[1])

  # indices
  if batch_size == 1:
      x_start_index = np.random.randint(0, dataset_length-seq_length-1)
      X = merged_string[x_start_index : x_start_index + seq_length]
      y = merged_string[x_start_index + seq_length]
  else:
      X = []
      y = []
      for i in range(batch_size):
          x_start_index = np.random.randint(0, dataset_length-seq_length-1)
          X.append(merged_string[x_start_index : x_start_index + seq_length])
          y.append(merged_string[x_start_index + seq_length])

  return (X, y)


# Make list of all unique characters in datasets
all_text = "".join(x_train_list) + "".join(x_dev_list) + "".join(x_test_list)
unique_characters = sorted(list(set(all_text)))

# Calculte number of unique characters
number_unique_characters = len(unique_characters)

# Print results
print(f"List of unique characters: {unique_characters}")
print(f"Number of unique characters: {number_unique_characters}")


# make dictionaries for changing between characters and their assigned index, and vice versa
char_to_index = {key:value for value,key in enumerate(unique_characters)}
index_to_char = {key:value for key,value in enumerate(unique_characters)}

# Check char_to_index and index_to_char dictionaries are mapping correctly
test_charindex_mapping(x_train_list[0], char_to_index, index_to_char)


# SENTENCE COMPLETION FUNCTIONS

# functions to predict rest of sentence

def predict_next_character(input_string, char_to_index_dict, number_unique_characters):
  """
    Predicts next character given an arbitrary length input string
    Input:
    - input_string = string for which to predict the next character
    - char_to_index_dict = Dictionary mapping characters to their index
    - number_unique_characters = number of potential unique characters in dataset
    Output:
    - Predicted next character
    """
    with torch.no_grad():
        one_hot_string = text_to_onehot_multiline(input_string, char_to_index_dict,
        ↪  number_unique_characters)
        one_hot_string = one_hot_string.to(device)
        next_character_pred = best_model(one_hot_string)
        next_character_pred = torch.argmax(next_character_pred, 1)
        next_character_pred = index_to_char.get(next_character_pred.item())

    return next_character_pred


def predict_next_character_randomness(input_string, char_to_index_dict,
↪   number_unique_characters):
    """
    Predicts next character given an arbitrary length input string, but chooses the character
    randomly according the distribution predicted by the RNN. This randomness helps to avoid
    the sentence prediction ending up in loops of repeated words.
    Input:
    - input_string = string for which to predict the next character
    - char_to_index_dict = Dictionary mapping characters to their index
    - number_unique_characters = number of potential unique characters in dataset
    Output:
    - Predicted next character
    """

    with torch.no_grad():
        one_hot_string = text_to_onehot_multiline(input_string, char_to_index_dict,
        ↪  number_unique_characters)
        one_hot_string = one_hot_string.to(device)
        next_character_pred = F.softmax(torch.squeeze(best_model(one_hot_string).to('cpu')),
        ↪  dim=0).numpy()
        next_character_pred = np.random.choice([i for i in range(29)],
        ↪  p=np.squeeze(next_character_pred))
        next_character_pred = index_to_char.get(next_character_pred.item())

    return next_character_pred


def play_complete_sentence():
    """
    Asks for user input for the start of the a sentence
    Returns a completed sentence
    """
    with torch.no_grad():
        sentence_start = input("Please enter the start of a sentence: ")
        sentence = sentence_start
        sentence = complete_sentences(sentence, randomness=True)

        print(f"Completed sentence, by Mr Robot: {sentence}")
```

```
def complete_sentences(sentence_start_list, randomness=False, to_randomise="first_letter",
↪   max_length=100):
    """
    Input:
    - sentence_start_list: list of strings each containing the start of a sentence
    - randomness: Whether to include randomness in sentence prediction to help prevent the
    sentence prediction from looping round specific phrases (i.e. whether predict next
↪   character
    by sampling from discrete distribution  over the characters, as predicted from the model).
    - to_randomise: Options are "first_letter" or "all" - this determines which letters to
↪   apply the
    randomness too
    - max_length: max length of predicted sentences
    Output:
    - List of predicted completed sentences
    """
    with torch.no_grad():

        # if the input is a single string
        if isinstance(sentence_start_list, list):
            for sentence_start in sentence_start_list:
                sentence = sentence_start

                # continue predicting new characters until a fullstop or upto the max_length
                while sentence[-1] != "." and len(sentence) < max_length:
                    if randomness == False: # always just predict the most likely character
                        sentence += predict_next_character(sentence, char_to_index,
                        ↪  number_unique_characters)
                    elif randomness == True: # incorporate randomness
                        if to_randomise=="first_letter": # randomness for first letter of each
                        ↪  word only
                            if sentence[-1] == " ":
                                sentence += predict_next_character_randomness(sentence,
                                ↪  char_to_index, number_unique_characters)
                            else:
                                sentence += predict_next_character(sentence, char_to_index,
                                ↪  number_unique_characters)
                        elif to_randomise=="all": # randomness for all characters
                            sentence += predict_next_character_randomness(sentence,
                            ↪  char_to_index, number_unique_characters)
                print(f"Start of sentence, by Mr Human: {sentence_start}")
                print(f"Completed sentence, by Mr Robot: {sentence}")

        # if the input is a list of strings
        else:
            sentence_start = sentence_start_list
            sentence = sentence_start

            # continue predicting new characters until a fullstop or upto the max_length
            while sentence[-1] != "." and len(sentence) < max_length:
                if randomness == False: # always just predict the most likely character
                    sentence += predict_next_character(sentence, char_to_index,
                    ↪  number_unique_characters)
                elif randomness == True: # incorporate randomness
                    if to_randomise=="first_letter": # randomness for first letter of each
                    ↪  word only
```

```
                        if sentence[-1] == " ":
                            sentence += predict_next_character_randomness(sentence,
                            ↪  char_to_index, number_unique_characters)
                        else:
                            sentence += predict_next_character(sentence, char_to_index,
                            ↪  number_unique_characters)
                    elif to_randomise=="all": # randomness for all characters
                        sentence += predict_next_character_randomness(sentence, char_to_index,
                        ↪  number_unique_characters)

            print(f"Start of sentence, by Mr Human: {sentence_start}")
            print(f"Completed sentence, by Mr Robot: {sentence}")


# Write some sentence starts to try the trained models on

sentence_start_list = []
sentence_start_list.append("dobby squeals as if ")
sentence_start_list.append("when dorothy pu")
sentence_start_list.append("the incredible hulk ter")
sentence_start_list.append("nemo finally finds hi")
sentence_start_list.append("james bond's mys")
sentence_start_list.append("the director's decision to  ")


# HYPERPARAMETERS

# Hyperparameters to grid search over (GRID SEARCH SCENARIO)
# Note that some combinations of these hyperparameters have been excluded
# This is to reduce computational cost by removing combinations expected to
# perform less well after initial scoping experiments
# The exclusion takes place towards the start of the main grid search for loop

hyperparam_grid = {'num_layers': [1,3],
                   'hidden_size': [128,256],
                   'cell_type': ["vanillaRNN","GRU","LSTM"],
                   'num_dense_layers': [1,2],
                   'learning_rate': [0.0002,0.0005],
                   'batch_size': [20],
                   'hidden_to_dense_size_ratio': [-99,2,4] #-99 when N/A (when 1 dense layer)
}

# cell_type_current = "vanillaRNN" # this will exclude all except the specified cell type from
↪  the grid search
optim_choice = "Adam" # note changing this will not change the optimiser in code below (just
↪  here for plotting)


# range of epochs (note we also use early stopping)
min_epochs = 50
max_epochs = 800

# how many characters to have in training and validation
seq_length_range = (10, 50)

# define number of training sequences to count as a single epoch
```

```
length_train_set = 10000

# initiate dataframe to store results of grid search
results_df = pd.DataFrame()


for run_index, hyperparams in enumerate(ParameterGrid(hyperparam_grid)):

    # At first run generate the validation dataset
    # The validation dataset stays the same for each re-run to enable comparison between
    ↪  models
    # validation set is 1/4 the size of the training set
    if run_index == 0:

        X_dev_ = []
        y_dev_ = []

        for i in range(length_train_set//4):
            X_dev_single, y_dev_single = random_xy_batch(dev_merged_string, seq_length_range,
            ↪  1)
            X_dev_.append(X_dev_single)
            y_dev_.append(y_dev_single)

        length_dev_set = len(y_dev_)


    # Certain combinations of hyperparams to exclude from grid search:
    if hyperparams['cell_type']=="LSTM" and hyperparams['learning_rate'] != 0.0005:
        continue
    if hyperparams['cell_type']!="LSTM" and hyperparams['learning_rate'] == 0.0005:
        continue
    if hyperparams['num_dense_layers']==1 and hyperparams['hidden_to_dense_size_ratio']!=-99:
        continue
    if hyperparams['num_dense_layers']==2 and hyperparams['hidden_to_dense_size_ratio']==-99:
        continue


    # specify model architecture

    input_size = number_unique_characters
    num_classes = number_unique_characters

    class characterRNN(nn.Module):

        def __init__(self, input_size, hidden_size, num_layers, num_classes, cell_type,
        ↪  num_dense_layers, hidden_to_dense_size_ratio):
            super(characterRNN, self).__init__()
            self.num_layers = num_layers
            self.hidden_size = hidden_size
            self.cell_type = cell_type
            self.num_dense_layers = num_dense_layers
            if num_dense_layers == 2:
                self.hidden_to_dense_size_ratio = hidden_to_dense_size_ratio

            # only one of these will be used in forward statement (as specified by hyperparam
            ↪  cell_type for current run)
            self.rnn = nn.RNN(input_size, hidden_size, num_layers, batch_first = True)
            self.gru = nn.GRU(input_size, hidden_size, num_layers, batch_first = True)
            self.lstm = nn.LSTM(input_size, hidden_size, num_layers, batch_first = True)

            # number of these linear layers that gets used depends on hyperparam
             ↪  num_dense_layers
            self.linear1 = nn.Linear(hidden_size, num_classes)
            if num_dense_layers == 2:
                self.linear2 = nn.Linear(hidden_size, hidden_size//hidden_to_dense_size_ratio)
                self.linear3 = nn.Linear(hidden_size//hidden_to_dense_size_ratio, num_classes)

        def forward(self, x_batch):

            # Initialise hidden state
            h0 = torch.zeros(self.num_layers, x_batch.size(0), self.hidden_size).to(device)
            # Inisitialise cell state (used in LSTM only, not GRU or vanilla RNN)
            c0 = torch.zeros(self.num_layers, x_batch.size(0), self.hidden_size).to(device)

            # out shape: (batch_size, seq_length, hidden_size)
            if self.cell_type == "vanillaRNN":
                out, _ = self.rnn(x_batch, h0)
            if self.cell_type == "GRU":
                out, _ = self.gru(x_batch, h0)
            if self.cell_type == "LSTM":
                out, _ = self.lstm(x_batch, (h0, c0))

            # extract the final cells output and feed into a dense layer
            out = out[:, -1, :]
            if self.num_dense_layers == 1:
                out = self.linear1(out)
            if self.num_dense_layers == 2:
                out = F.relu(self.linear2(out))
                out = self.linear3(out)

            return out


    # Initialise model and send to GPU

    model = characterRNN(input_size,
                        hyperparams['hidden_size'],
                        hyperparams['num_layers'],
                        num_classes,
                        hyperparams['cell_type'],
                        hyperparams['num_dense_layers'],
                        hyperparams['hidden_to_dense_size_ratio'])

    device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
    model.to(device)


    # Loss & optimiser
    criterion = nn.CrossEntropyLoss() # note that this will apply the softmax for us
    optimiser = torch.optim.Adam(model.parameters(), lr=hyperparams['learning_rate'])
```

```
# training loop

running_loss = 0
loss_list = []
running_corrects = 0
training_accuracy_list = []

best_dev_accuracy = 0
best_epoch = 0
best_train_accuracy = 0

dev_loss_list = []
dev_accuracy_list = []

converged = False
iters_counter = 0
prev_epoch_counter = 0
epochs_counter = 0
epochs_to_plot = []

while (converged == False) and (epochs_counter <= max_epochs):


    model.train()

    # data prep

    ## generate batch
    X_batch_, y_batch_ = random_xy_batch(train_merged_string, seq_length_range,
    ↪  hyperparams['batch_size'])
    ## one-hot encode X_batch
    X_batch = text_to_onehot_multiline(X_batch_, char_to_index, number_unique_characters)
    ## convert y_batch to target indices in the appropriate shape (batch_size, 1)
    y_batch = torch.zeros((len(y_batch_)), dtype=torch.int)
    for i, char in enumerate(y_batch_):
          y_batch[i] = char_to_index.get(char)
    ## Send to GPU
    X_batch = X_batch.to(device)
    y_batch = y_batch.to(device)

    # forward pass, loss
    y_batch_pred = model(X_batch)

    # training loss
    loss = criterion(y_batch_pred, y_batch.long())
    running_loss += loss

    # backward pass
    loss.backward()

    # updates
    optimiser.step()
    optimiser.zero_grad()

    model.eval()
```

```
with torch.no_grad():

    # training accuracy
    y_batch_pred_indices = torch.argmax(y_batch_pred, 1)
    running_corrects += (y_batch_pred_indices == y_batch).sum().item()

    if prev_epoch_counter < epochs_counter:

        # training loss
        loss_avg = running_loss/iters_counter
        loss_list.append(loss_avg.item())

        # training accuracy
        training_accuracy_avg = running_corrects/length_train_set
        training_accuracy_list.append(training_accuracy_avg)

        # validation dataset evaluation

        dev_running_loss = 0
        dev_running_corrects = 0

        for i in range(length_dev_set):
            # generate validation example
            X_dev_i = X_dev_[i]
            y_dev_i = y_dev_[i]
            # one-hot encode X_dev
            X_dev = text_to_onehot_multiline(X_dev_i, char_to_index,
            ↪  number_unique_characters)
            # convert y_dev to target indices in the appropriate shape
            y_dev = torch.zeros((len(y_dev_i)), dtype=torch.int)
            for j, char in enumerate(y_dev_i):
                y_dev[j] = char_to_index.get(char)

            # Send to GPU
            X_dev = X_dev.to(device)
            y_dev = y_dev.to(device)

            # forward pass
            y_dev_pred = model(X_dev)

            # validation loss
            loss_dev = criterion(y_dev_pred, y_dev.long())
            dev_running_loss += loss_dev

            # training accuracy
            y_dev_pred_indices = torch.argmax(y_dev_pred, 1)
            dev_running_corrects += (y_dev_pred_indices == y_dev).sum().item()

        # calculate validation loss and store in list ready for plotting
        dev_loss_avg = dev_running_loss/length_dev_set
        dev_loss_list.append(dev_loss_avg.item())

        # calculate validation accuracy and store in list ready for plotting
        dev_accuracy_avg = dev_running_corrects/length_dev_set
        dev_accuracy_list.append(dev_accuracy_avg)
```

```
            if epochs_counter%20 == 0:
                print(f"epoch: {epochs_counter}, train_loss: {loss_avg.item():.4f},
                ↪  train_acc: {training_accuracy_avg:.4f}, val_loss:
                ↪  {dev_loss_avg.item():.4f}, val_acc: {dev_accuracy_avg:.4f}")

            # zero running metric values ready for next run
            running_loss = 0
            running_corrects = 0
            dev_running_loss = 0
            dev_running_corrects = 0

            epochs_to_plot.append(epochs_counter)

            ## Store the current results if it is the highest validation accuracy so far
            if dev_accuracy_avg > best_dev_accuracy:
                best_dev_accuracy = dev_accuracy_avg
                best_dev_loss = dev_loss_avg.item()
                best_train_accuracy = training_accuracy_avg
                best_loss = loss_avg.item()
                best_epoch = epochs_counter
                best_model = model

            ## Early stopping criteria
            if epochs_counter >= min_epochs:
                if (np.mean(dev_loss_list[-20:]) - np.mean(dev_loss_list[-40:-20])) > 0:
                    converged = True
                    print("Model Converged")

            iters_counter = 0

        iters_counter += 1
        prev_epoch_counter = epochs_counter
        if iters_counter*hyperparams['batch_size']%length_train_set == 0:
            epochs_counter += 1

# Display results from epoch with best validation accuracy, for current run and
↪  hyperparams

hyperparams['model_index'] = run_index + 1
hyperparams['best_epoch'] = best_epoch
hyperparams['best_validation_accuracy'] = best_dev_accuracy
hyperparams['associated_validation_loss'] = best_dev_loss
hyperparams['associated_training_accuracy'] = best_train_accuracy
hyperparams['associated_training_loss'] = best_loss

# Store the results for the best epoch in a dataframe and save as csv file
results_df = results_df.append(hyperparams , ignore_index=True)
results_df = results_df.sort_values("best_validation_accuracy", ascending=False).round(4)
results_df.to_csv(f"{path}RESULTS_TABLE_{hyperparams['cell_type']}.csv")

print(f"\n RUN_INDEX: {run_index + 1} \n")
print(hyperparams)

# save the model
torch.save(best_model,
↪  f"{path}{hyperparams['cell_type']}_{best_dev_accuracy:.4f}_devacc_{best_train_accuracy:.4f}_traina
```

```
    # # Loss plot
    plt.plot(epochs_to_plot, loss_list, color='k', linestyle='-')
    plt.plot(epochs_to_plot, dev_loss_list, color='r', linestyle='-')
    plt.legend(['Training', 'Validation'], loc='upper right')
    plt.ylabel('Loss', color='k')
    plt.xlabel('Epoch', color='k')
    plt.title(f'''Loss plot ({hyperparams['cell_type']}, Val Accuracy {best_dev_accuracy})
    Hidden size: {hyperparams['hidden_size']}, Recurrent layers: {hyperparams['num_layers']},
↪   Dense layers: {hyperparams['num_dense_layers']},
    Hidden to dense size ratio: {hyperparams['hidden_to_dense_size_ratio']}, Optimiser:
↪   {optim_choice}, LR: {hyperparams['learning_rate']},
    Batch size: {hyperparams['batch_size']}, Input sequence length range:
↪   {seq_length_range}''', color='k')


     ↪  plt.savefig(f"{path}{hyperparams['cell_type']}_{best_dev_accuracy:.4f}_devacc_{best_train_accuracy
     ↪  dpi=300, bbox_inches = "tight")
    plt.show()

    # # Accuracy plot
    plt.plot(epochs_to_plot, training_accuracy_list, color='k', linestyle='-')
    plt.plot(epochs_to_plot, dev_accuracy_list, color='r', linestyle='-')
    plt.legend(['Training', 'Validation'], loc='lower right')
    plt.ylabel('Accuracy', color='k')
    plt.xlabel('Epoch', color='k')
    plt.title(f'''Accuracy plot ({hyperparams['cell_type']}, Val Accuracy {best_dev_accuracy})
    Hidden size: {hyperparams['hidden_size']}, Recurrent layers: {hyperparams['num_layers']},
↪   Dense layers: {hyperparams['num_dense_layers']},
    Hidden to dense size ratio: {hyperparams['hidden_to_dense_size_ratio']}, Optimiser:
↪   {optim_choice}, LR: {hyperparams['learning_rate']},
    Batch size: {hyperparams['batch_size']}, Input sequence length range:
↪   {seq_length_range}''', color='k')


     ↪  plt.savefig(f"{path}{hyperparams['cell_type']}_{best_dev_accuracy:.4f}_devacc_{best_train_accuracy
     ↪  dpi=300, bbox_inches = "tight")
    plt.show()

    # # Display some sentence predictions
    print("without randomness:")
    complete_sentences(sentence_start_list)
    print("With randomness:")
    complete_sentences(sentence_start_list, randomness=29)
    print("From test set:")
    for i in range(5):
        xsample, ysample = random_xy_batch(test_merged_string, seq_length_range, 1)
        complete_sentences(xsample, randomness=5)


# EVALUATE ON TEST SET

# generate the test set 1/4 size of training set

X_test_ = []
y_test_ = []
```

```
for i in range(length_train_set//4):
    X_test_single, y_test_single = random_xy_batch(test_merged_string, seq_length_range, 1)
    X_test_.append(X_test_single)
    y_test_.append(y_test_single)

length_test_set = len(y_test_)


# model input and output size

input_size = number_unique_characters
num_classes = number_unique_characters

# class correspond to structure of the best model

class characterRNN(nn.Module):

    def __init__(self, input_size, hidden_size, num_layers, num_classes, cell_type,
     ↪  num_dense_layers, hidden_to_dense_size_ratio):
        super(characterRNN, self).__init__()
        self.num_layers = num_layers
        self.hidden_size = hidden_size
        self.cell_type = cell_type
        self.num_dense_layers = num_dense_layers
        self.lstm = nn.LSTM(input_size, hidden_size, num_layers, batch_first = True)
        self.linear1 = nn.Linear(hidden_size, num_classes)

    def forward(self, x_batch):

        # Initialise hidden state
        h0 = torch.zeros(self.num_layers, x_batch.size(0), self.hidden_size).to(device)
        # Inisitialise cell state
        c0 = torch.zeros(self.num_layers, x_batch.size(0), self.hidden_size).to(device)

        # out shape: (batch_size, seq_length, hidden_size)
        out, _ = self.lstm(x_batch, (h0, c0))

        # extract the final cells output and feed into a dense layer
        out = out[:, -1, :]
        out = self.linear1(out)

        return out


# load best model (highest validation accuracy) for further evaluation

best_model = torch.load(f"{path}LSTM_0.6016_devacc_0.6108_trainacc.pth")
best_model.eval()

device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
best_model.to(device)


# Final performance of best model on test set

test_running_loss = 0
```

```
test_running_corrects = 0

# Loss & optimiser
criterion = nn.CrossEntropyLoss() # note that this will apply the softmax for us

for i in range(length_test_set):
    ## take a test example
    X_test_i = X_test_[i]
    y_test_i = y_test_[i]
    ## one-hot encode X_test_i
    X_test = text_to_onehot_multiline(X_test_i, char_to_index, number_unique_characters)
    ## convert y_test to target indices in the appropriate shape
    y_test = torch.zeros((len(y_test_i)), dtype=torch.int)
    for j, char in enumerate(y_test_i):
        y_test[j] = char_to_index.get(char)

    ## Send to GPU
    X_test = X_test.to(device)
    y_test = y_test.to(device)

    ## forward pass
    y_test_pred = best_model(X_test)

    ## test loss
    loss_test = criterion(y_test_pred, y_test.long())
    test_running_loss += loss_test

    ## test accuracy
    y_test_pred_indices = torch.argmax(y_test_pred, 1)
    test_running_corrects += (y_test_pred_indices == y_test).sum().item()

test_loss_avg = test_running_loss/length_test_set
test_accuracy_avg = test_running_corrects/length_test_set

# print results
print(f"Final test accuracy: {test_accuracy_avg}")
print(f"Final test loss: {test_loss_avg}")


# GENERATE WHOLE SENTENCES

# Write some sentence starts to try the trained model on

sentence_start_list = []
sentence_start_list.append("dobby squeals as if ")
sentence_start_list.append("when dorothy pu")
sentence_start_list.append("the incredible hulk ter")
sentence_start_list.append("nemo finally finds hi")
sentence_start_list.append("james bond's mys")
sentence_start_list.append("the director's decision to  ")

print("without randomness:")
complete_sentences(sentence_start_list)
print("With randomness:")
complete_sentences(sentence_start_list, randomness=True, to_randomise="first_letter",
↪  max_length=100)
```

```
print("From test set (with randomness):")
for i in range(5):
    xsample, ysample = random_xy_batch(test_merged_string, seq_length_range, 1)
    complete_sentences(xsample, randomness=True, to_randomise="first_letter")


# ask the user for an input sentence and return the completion

play_complete_sentence()
```

Appendix D Q4 code

```
# Import libraries
import torch
import gzip
import gensim
import numpy as np
import pandas as pd
import re
import string
import torch.nn as nn
import torch.optim as optim
import torch.nn.functional as F
from torch.utils.data import TensorDataset
from torch.utils.data import DataLoader
from math import ceil
from sklearn.model_selection import ParameterGrid
import matplotlib.pyplot as plt
import random

# Mount Google Drive
from google.colab import drive
drive.mount('/content/drive')
path = "/content/drive/My Drive/Q4_saved_models/"

### Load Word2Vec embeddings ###

# Load Google's pre-trained Word2Vec model
!wget https://s3.amazonaws.com/dl4j-distribution/GoogleNews-vectors-negative300.bin.gz
w2v = gensim.models.KeyedVectors.load_word2vec_format("GoogleNews-vectors-negative300.bin.gz",
↪   binary=True, limit = 1000000)

# Normalise the vectors
w2v.init_sims(replace=True)

# Create token to index dictionary
token2index_map = {
    word:idx
    for idx, word in enumerate(w2v.vocab)
}

def token2index(token):
  """
  Get the word2vec embedding index for a token
  Input:
  - Token = a string
  Output:
  - The index of the word2vec embedding of the token"""
  # Use "UNK" as missing token embedding
  default = token2index_map.get("UNK")

  return token2index_map.get(token, default)

### Load Stanford Sentiment Treebank dataset ###

# Site for more information on data set: https://nlp.stanford.edu/sentiment/
```

```python
!wget https://nlp.stanford.edu/sentiment/trainDevTestTrees_PTB.zip
!unzip trainDevTestTrees_PTB.zip

# Parse the tree into sentences and classifications
def parse_tree(sentence_list_tree):
    """
    Function for extracting text only from sentiment tree (remove tree structure)
    Input: list of strings of trees of sentences/lines
    Output: x_list is a list of sentences, y_list is a list of sentence sentiment classes
    """
    # remove empty strings from list
    while("" in sentence_list_tree):
        sentence_list_tree.remove("")

    # initialise lists to store x (the sentence), y (the sentiment)
    x_list = []
    y_list = []
    for sentence in sentence_list_tree:
        y_list.append(int(sentence[1])) # y = sentiment class of sentence
        # remove digits and parentheses using regular expression
        patterns = r'[0-9]|\(|\)'
        sentence = re.sub(patterns, '', sentence)
        # remove -RRB- and -LRB-
        sentence = re.sub(r"-RRB-", "", sentence)
        sentence = re.sub(r"-LRB-", "", sentence)
        # remove duplicate spaces
        sentence = " ".join(sentence.split())
        # remove spaces before some types of punctuation
        sentence = re.sub(r"\s([?.!;\,](?:\s|$))", r'\1', sentence)
        # remove spaces before apostrophes
        sentence = re.sub(r" '\b", "'", sentence)
        # remove spaces before n't
        sentence = re.sub(r" n't\b", "n't", sentence)
        # remove space before ...
        sentence = re.sub(r" \.\.\.", r"...", sentence)

        # Additional edits to get words only

        # remove dashes (replace with space)
        sentence = re.sub("-", " ", sentence)
        # remove slashes (replace with space)
        sentence = sentence.replace("\\"," ")
        # remove all other punctuation
        sentence = re.sub('[!"#$%&()*+,./:;<=>?@[\]^_`{|}~]', "", sentence)
        # remove double quotation marks
        sentence = re.sub("''", "", sentence)
        # remove single quotation marks (space beforehand so as to not replace apostrophes)
        sentence = re.sub(" '", " ", sentence)
        # remove duplicate spaces
        sentence = " ".join(sentence.split())

        x_list.append(sentence)

    return x_list, y_list
```

```
# load and parse string data
train_file = open("train.txt", "r")
train_tree = train_file.read()
train_tree_sentence_list = train_tree.split("\n")
x_train_list, y_train_list = parse_tree(train_tree_sentence_list)

dev_file = open("dev.txt","r")
dev_tree = dev_file.read()
dev_tree_sentence_list = dev_tree.split("\n")
x_dev_list, y_dev_list = parse_tree(dev_tree_sentence_list)

test_file = open("test.txt", "r")
test_tree = test_file.read()
test_tree_sentence_list = test_tree.split("\n")
x_test_list, y_test_list = parse_tree(test_tree_sentence_list)

# display example data for training set
start_index = 60
end_index = 65

for sentiment, sentence in
↪  zip(y_train_list[start_index:end_index],x_train_list[start_index:end_index]):
  print(f"y = {sentiment} | x = ({sentence})")

### Data processing ###

class SentenceBatcher:
    def __init__(self, inputs, labels, batch_size=128, drop_last=False):
        # Tokenise the input sentences
        tokenised_inputs = np.array([sentence.split() for sentence in inputs])

        # Map the tokens to word2vec indices
        indices_mapping = [torch.tensor([token2index(token) for token in tokens]) for tokens
        ↪  in tokenised_inputs]

        # Store sentences by length
        self.sentences_by_length = {}
        for input, label in zip(indices_mapping, labels):
            length = input.shape[0]

            if length not in self.sentences_by_length:
                self.sentences_by_length[length] = []
            self.sentences_by_length[length].append([input, label])

        # Create a DataLoader for each set of sentences of the same length
        self.loaders = {length : torch.utils.data.DataLoader(
                                  sentences,
                                  batch_size=batch_size,
                                  shuffle=True,
                                  drop_last=drop_last)
          for length, sentences in self.sentences_by_length.items()}

    def __iter__(self):
        # Create an iterator for each sentence length
        iters = [iter(loader) for loader in self.loaders.values()]
        while iters:
            # Get a random iterator
            i = random.choice(iters)
            try:
                yield next(i)
            except StopIteration:
                iters.remove(i)

### Model ###

class MLP(nn.Module):
  def __init__(self, in_size, hid_size, out_size, num_layers, dropout, pooling_type,
   ↪  fixed_embeddings):
    super().__init__()
    self.layer1 = nn.Linear(in_size, hid_size)
    self.layer2a = nn.Linear(hid_size, out_size)
    self.layer2b = nn.Linear(hid_size, hid_size//2)
    self.layer3a = nn.Linear(hid_size//2, out_size)
    self.layer3b = nn.Linear(hid_size//2, hid_size//4)
    self.layer4 = nn.Linear(hid_size//4, out_size)

    self.num_layers = num_layers
    self.pooling_type = pooling_type
    self.embedding = nn.Embedding.from_pretrained(torch.FloatTensor(w2v.vectors), freeze =
     ↪  fixed_embeddings)
    self.dropout = nn.Dropout(dropout)

  def forward(self, x_batch):

    # Get word embeddings from W2V indices
    emb = self.embedding(x_batch)

    if self.pooling_type == "max":
      # max pooling
      batch_vec = torch.max(emb, dim = 1)[0]

    elif self.pooling_type == "min":
      # min pooling
      batch_vec = torch.min(emb, dim = 1)[0]

    elif self.pooling_type == "mean":
      # mean pooling
      batch_vec = torch.mean(emb, dim = 1)

    elif self.pooling_type == "mean_max_min":
      # mean, max and min pooling concatenated together
      pool_mean = torch.mean(emb, dim = 1)
      pool_max = torch.max(emb, dim = 1)[0]
      pool_min = torch.min(emb, dim = 1)[0]
      batch_vec = torch.cat((pool_mean, pool_max, pool_min), dim=1)

    if self.num_layers == 2:
      x = self.layer1(batch_vec)
      x = torch.relu(x)
      x = self.dropout(x)
      x = self.layer2a(x)
    elif self.num_layers == 3:
      x = self.layer1(batch_vec)
      x = torch.relu(x)
      x = self.dropout(x)
      x = self.layer2b(x)
      x = torch.relu(x)
      x = self.dropout(x)
      x = self.layer3a(x)

    elif self.num_layers == 4:
      x = self.layer1(batch_vec)
      x = torch.relu(x)
      x = self.dropout(x)
      x = self.layer2b(x)
      x = torch.relu(x)
      x = self.dropout(x)
      x = self.layer3b(x)
      x = torch.relu(x)
      x = self.dropout(x)
      x = self.layer4(x)

    return x

# Define minumum and maximum epochs to run model for
min_epochs = 30
max_epochs = 200

# First do grid search over this hyperparameter grid, keeping embeddings fixed
hyperparam_grid = {'hid_size': [64, 128],
                   'learning_rate': [0.01, 0.001, 0.0001],
                   'num_layers': [3, 4],
                   'dropout': [0, 0.25],
                   'pooling_type': ["max", "mean", "mean_max_min"],
                   'batch_size': [64, 128],
                   'fixed_embeddings': [True]
}

# Then do grid search over this hyperparameter grid, allowing the embeddings to update
# hyperparam_grid = {'hid_size': [64, 128],
#                    'learning_rate': [0.001],
#                    'num_layers': [3, 4],
#                    'dropout': [0, 0.25],
#                    'pooling_type': ["mean"],
#                    'batch_size': [64, 128],
#                    'fixed_embeddings': [False]
# }

# Initiate dataframe to store results of grid search
results_df = pd.DataFrame()

# Initialise device
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')

# Define output size
out_size = 5
```

```
for run_index, hyperparams in enumerate(ParameterGrid(hyperparam_grid)):

  # Get training and validation data loaders
  train_loader = SentenceBatcher(x_train_list, y_train_list,
  ↪  batch_size=hyperparams['batch_size'], drop_last=False)
  dev_loader = SentenceBatcher(x_dev_list, y_dev_list, batch_size=hyperparams['batch_size'],
  ↪  drop_last=False)

  # Set input size to be 900 if concatenating mean, max and min pooling vectors otherwise just
  ↪  300 for one of them
  in_size = 900 if hyperparams['pooling_type'] == "mean_max_min" else 300

  # Initialise model
  model = MLP(in_size,
              hyperparams['hid_size'],
              out_size,
              hyperparams['num_layers'],
              hyperparams['dropout'],
              hyperparams['pooling_type'],
              hyperparams['fixed_embeddings']).to(device)

  # Loss & optimiser
  criterion = nn.CrossEntropyLoss()
  optimizer = torch.optim.Adam(model.parameters(), lr=hyperparams['learning_rate'])

  dev_length = len(x_dev_list)
  train_length = len(x_train_list)

  train_costs = []
  train_accuracies = []
  dev_costs = []
  dev_accuracies = []

  best_dev_accuracy = 0
  best_epoch = 0
  best_train_accuracy = 0
  epoch = 0

  converged = False

  while (converged == False) and (epoch <= max_epochs):
    model.train()

    running_train_loss = 0.0
    running_dev_loss = 0.0
    num_of_train_batches = 0
    num_of_dev_batches = 0
    correct_train_pred = 0
    correct_dev_pred = 0

    # Training loop
    for inputs, labels in train_loader:

      # Send to GPU
      inputs = inputs.to(device)
      labels = labels.to(device)
      # Forward pass
      train_predicted = model(inputs)

      # Loss
      loss = criterion(train_predicted, labels)
      running_train_loss += loss.item()
      num_of_train_batches +=1

      # Backward pass
      optimizer.zero_grad()
      loss.backward()
      optimizer.step()

      # Get number of correct training predictions per batch and sum
      with torch.no_grad():
        _, train_pred = torch.max(train_predicted, 1)
        correct_train_pred += (train_pred == labels).sum().item()

    model.eval()
    with torch.no_grad():

      # Calculate training accuracy per epoch and record
      train_accuracy = correct_train_pred/train_length
      train_accuracies.append(train_accuracy)

      # Record loss
      train_costs.append(running_train_loss/num_of_train_batches)

      # Test on validation set
      for dev_inputs, dev_labels in dev_loader:

        # Send to GPU
        dev_inputs = dev_inputs.to(device)
        dev_labels = dev_labels.to(device)

        # Forward pass
        dev_predicted = model(dev_inputs)

        # Loss
        dev_loss = criterion(dev_predicted, dev_labels)
        running_dev_loss += dev_loss.item()
        num_of_dev_batches +=1

        # Get number of correct validation predictions per batch and sum
        _, dev_pred = torch.max(dev_predicted, 1)
        correct_dev_pred += (dev_pred == dev_labels).sum().item()

      # Calculate and record validation accuracy and loss
      dev_accuracy = correct_dev_pred/dev_length
      dev_accuracies.append(dev_accuracy)
      dev_costs.append(running_dev_loss/num_of_dev_batches)

      train_loss_avg = running_train_loss/num_of_train_batches
      dev_loss_avg = running_dev_loss/num_of_dev_batches
```

```
        # Print results every 20 epochs
        if epoch%(max_epochs//20) == 0:
            print(f'Epoch: {epoch + 1}/{max_epochs + 1}, train loss: {train_loss_avg:.4f}, dev
             ↪  loss: {dev_loss_avg:.4f} train acc: {train_accuracy:.4f}, dev acc:
             ↪  {dev_accuracy:.4f}')

        # Store the current results if it is the highest validation accuracy so far
        if dev_accuracy > best_dev_accuracy:
            best_dev_accuracy = dev_accuracy
            best_dev_loss = dev_loss_avg
            best_train_accuracy = train_accuracy
            best_loss = train_loss_avg
            best_epoch = epoch
            best_model = model

        # Early stopping criteria
        if epoch >= min_epochs:
          if (np.mean(dev_costs[-10:]) - np.mean(dev_costs[-20:-10])) > 0.01:
            converged = True
            print("Model Converged")

        epoch += 1

    # Display results from epoch with best validation accuracy, for current run and hyperparams
    hyperparams['model_index'] = run_index + 1
    hyperparams['best_epoch'] = best_epoch
    hyperparams['best_validation_accuracy'] = best_dev_accuracy
    hyperparams['associated_validation_loss'] = best_dev_loss
    hyperparams['associated_training_accuracy'] = best_train_accuracy
    hyperparams['associated_training_loss'] = best_loss

    results_df = results_df.append(hyperparams , ignore_index=True)
    results_df = results_df.sort_values("best_validation_accuracy", ascending=False).round(4)
    results_df.to_csv(f"{path}RESULTS_TABLE.csv")

    print(f"\n RUN_INDEX: {run_index + 1} \n")
    print(hyperparams)

    # Save the model
    torch.save(best_model,
     ↪  f"{path}{hyperparams['pooling_type']}_{best_dev_accuracy:.4f}_devacc_{best_train_accuracy:.4f}_train

    # Accuracy plot
    plt.plot(train_accuracies, color='k', linestyle='-')
    plt.plot(dev_accuracies, color='r', linestyle='-')
    plt.title(f'Model accuracy',  color='k')
    plt.ylabel('Accuracy',  color='k')
    plt.xlabel('Epoch',  color='k')
    plt.legend(['Training', 'Validation'], loc='upper left')
    plt.tick_params(colors='k')
    plt.title(f'''Accuracy plot (Val Accuracy {best_dev_accuracy:.4f}) \n
      Hidden size: {hyperparams['hid_size']}, Number of layers: {hyperparams['num_layers']},
      Learning rate: {hyperparams['learning_rate']},
      Dropout: {hyperparams['dropout']}, Pooling type: {hyperparams['pooling_type']}, Batch
  ↪   size: {hyperparams['batch_size']},
      Fixed Embeddings: {hyperparams['fixed_embeddings']}''', color='k')
        ↪  plt.savefig(f"{path}{hyperparams['pooling_type']}_{best_dev_accuracy:.4f}_devacc_{best_train_accurac
        ↪  dpi=300, bbox_inches = "tight")
    plt.show()


    # Loss plot
    plt.plot(train_costs, color='k', linestyle='-')
    plt.plot(dev_costs, color='r', linestyle='-')
    plt.title(f'Model loss',  color='k')
    plt.ylabel('loss',  color='k')
    plt.xlabel('Epoch',  color='k')
    plt.legend(['Training', 'Validation'], loc='upper left')
    plt.tick_params(colors='k')
    plt.title(f'''Loss plot (Val Accuracy {best_dev_accuracy:.4f}) \n
      Hidden size: {hyperparams['hid_size']}, Number of layers: {hyperparams['num_layers']},
      Learning rate: {hyperparams['learning_rate']},
      Dropout: {hyperparams['dropout']}, Pooling type: {hyperparams['pooling_type']}, Batch
   ↪   size: {hyperparams['batch_size']},
      Fixed Embeddings: {hyperparams['fixed_embeddings']}''', color='k')

        ↪  plt.savefig(f"{path}{hyperparams['pooling_type']}_{best_dev_accuracy:.4f}_devacc_{best_train_accurac
        ↪  dpi=300, bbox_inches = "tight")
    plt.show()

  ### Evaluate on test dataset ###

  # Load best model
  best_model = torch.load(f"{path}mean_0.4514_devacc_0.4974_trainacc.pth")

  # Get test dataloader
  test_loader = SentenceBatcher(x_test_list, y_test_list, batch_size=20, drop_last=False)

  # Specify device
  device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')

  test_length = len(x_test_list)
  test_accuracies = []

  best_model.eval()
  with torch.no_grad():

    correct_test_pred = 0

    for test_inputs, test_labels in test_loader:

      # Send to GPU
      test_inputs = test_inputs.to(device)
      test_labels = test_labels.to(device)

      # Forward pass
      test_predicted = best_model(test_inputs)

      # Get number of correct test predictions per batch and sum
      _, test_pred = torch.max(test_predicted, 1)
      correct_test_pred += (test_pred == test_labels).sum().item()
    # Record test accuracy
    test_accuracy = correct_test_pred/test_length
    test_accuracies.append(test_accuracy)

  print(f"Final test accuracy: {test_accuracy}")

  ### Online review classifications ###

  # Online reviews, pre-processed as in parse_tree
  x_online_list = [
                  "To this day, this is still my favorite pixar film. The animation is stellar,
                   ↪  its heartwarming, funny and proves that pixar movies are always bound to
                   ↪  be great (except for cars 2 but thats a different story). This has a shot
                   ↪  at the title 'best movie of the century'", #
                   ↪  https://www.imdb.com/review/rw5485122
                  "But the worst thing of all with this film is the mangling of Austen's
                   ↪  dialogue and the atrocious modern dialogue. Austen's dialogue needs no
                   ↪  assistance from a writer who thinks he/she can write like Austen.", #
                   ↪  https://www.imdb.com/review/rw1213354/
                  "It sucked. It was boring, it was contrived, it was cheaply done and the
                   ↪  characters made zero sense. All the coincidences and stupid behavior
                   ↪  almost put me to sleep. If there is any learning that can happen in this
                   ↪  world it will be the end of sequels forever and ever.", #
                   ↪  https://www.imdb.com/review/rw4229247/
                  "One of the all time great feel good movies, incredible story- brilliant
                   ↪  writing and directing all around. Don't think that there is a single
                   ↪  actor in the world who could've done a better job than Jim Carrey. A must
                   ↪  watch.", # https://www.imdb.com/review/rw5873851/
                  "One of the best romcom movies of all time. Great cast and superb acting.
                   ↪  Hard not to get emotional and invested once it starts." #
                   ↪  https://www.imdb.com/review/rw5895794/
  ]

  # SentenceBatcher will randomly order the reviews, so we keep track of their indices
  x_online_index = range(len(x_online_list))

  # Get online data loader
  online_loader = SentenceBatcher(x_online_list, x_online_index, batch_size=128,
   ↪  drop_last=False)

  # Evaluate model on online reviews

  for online_inputs, online_index in online_loader:
    online_inputs = online_inputs.to(device)

    online_predicted = best_model(online_inputs)
    _, online_pred = torch.max(online_predicted, 1)

    for index, pred in zip(online_index, online_pred):
      print(f"Review: {x_online_list[index]}")
      print(f"Prediction: {pred}")
```

Appendix E Q5 code

```
import collections
import gensim
import gzip
import matplotlib.pyplot as plt
import numpy as np
import pandas as pd
import random
import re
import string
import torch
import torch.nn as nn
import torch.optim as optim
from google.colab import drive
from sklearn.model_selection import ParameterGrid

# Mount google drive
drive.mount('/content/drive')
path = "/content/drive/My Drive/UCL/Modules/DL/assignment2/"

"""# Load Word2Vec Embedding"""

!wget https://s3.amazonaws.com/dl4j-distribution/GoogleNews-vectors-negative300.bin.gz

# Load Google's pre-trained Word2Vec model
w2v = gensim.models.KeyedVectors.load_word2vec_format("GoogleNews-vectors-negative300.bin.gz",
↪  binary=True, limit=1000000)

# Normalise the vectors
w2v.init_sims(replace=True)

token2index_map = {
    word: idx for idx, word in enumerate(w2v.vocab)
}

def token2index(token):
  """
  Get the word2vec embedding index for a token
  Input:
  - Token = a string
  Output:
  - The index of the word2vec embedding of the token"""
  # Use "UNK" as missing token embedding
  default = token2index_map.get("UNK")

  # Remove multi-word tokens to match embeddings in Julia starter code
  if "_" in token:
    return default

  return token2index_map.get(token, default)

"""# Load Stanford sentiment treebank dataset"""

# Load stanford sentiment treebank dataset
# Site for more information on data set: https://nlp.stanford.edu/sentiment/
```

```
!wget https://nlp.stanford.edu/sentiment/trainDevTestTrees_PTB.zip
!unzip trainDevTestTrees_PTB.zip

# Parse the tree into sentences and classifications

def parse_tree(sentence_list_tree):
    """
    Function for extracting text only from sentiment tree (remove tree structure)
    Input: list of strings of trees of sentences/lines
    Output: x_list is a list of sentences, y_list is a list of sentence sentiment classes
    """
    # remove empty strings from list
    while("" in sentence_list_tree):
        sentence_list_tree.remove("")

    # initialise lists to store x (the sentence), y (the sentiment)
    x_list = []
    y_list = []
    for sentence in sentence_list_tree:
        y_list.append(int(sentence[1])) # y = sentiment class of sentence
        # remove digits and parentheses using regular expression
        patterns = r'[0-9]|\(|\)'
        sentence = re.sub(patterns, '', sentence)
        # remove -RRB- and -LRB-
        sentence = re.sub(r"-RRB-", "", sentence)
        sentence = re.sub(r"-LRB-", "", sentence)
        # remove duplicate spaces
        sentence = " ".join(sentence.split())
        # remove spaces before some types of punctuation
        sentence = re.sub(r"\s([?.!;\,](?:\s|$))", r'\1', sentence)
        # remove spaces before apostrophes
        sentence = re.sub(r" '\b", "'", sentence)
        # remove spaces before n't
        sentence = re.sub(r" n't\b", "n't", sentence)
        # remove space before ...
        sentence = re.sub(r" \.\.\.", r"...", sentence)
        # make all lower case
        sentence = sentence.lower()
        # remove slashes (replace with space)
        sentence = sentence.replace("\\"," ")

        # remove dashes (replace with space)
        sentence = re.sub("-", " ", sentence)
        # remove all other punctuation
        sentence = re.sub('[!"#$%&()*+,/:;<=>?@[\]^_`{|}~]', "", sentence)
        # remove double quotation marks
        sentence = re.sub("''", "", sentence)
        # remove single quotation marks (space beforehand so as to not replace apostrophes)
        sentence = re.sub(" '", " ", sentence)
        # remove duplicate spaces
        sentence = " ".join(sentence.split())

        # replace letters with accents, with standard non-accented english letters
        sentence = re.sub('[àáâã]', "a", sentence)
        sentence = re.sub('[æ]', "ae", sentence)
        sentence = re.sub('[ç]', "c", sentence)
        sentence = re.sub('[èé]', "e", sentence)
        sentence = re.sub('[íï]', "i", sentence)
        sentence = re.sub('[ñ]', "n", sentence)
        sentence = re.sub('[óôö]', "o", sentence)
        sentence = re.sub('[ûü]', "u", sentence)

        x_list.append(sentence)

    return x_list, y_list

# Load and parse string data

train_file = open("/content/trees/train.txt", "r")
train_tree = train_file.read()
train_tree_sentence_list = train_tree.split("\n")
x_train_list, y_train_list = parse_tree(train_tree_sentence_list)
train_length = len(x_train_list)

dev_file = open("/content/trees/dev.txt","r")
dev_tree = dev_file.read()
dev_tree_sentence_list = dev_tree.split("\n")
x_dev_list, y_dev_list = parse_tree(dev_tree_sentence_list)
dev_length = len(x_dev_list)

test_file = open("/content/trees/test.txt", "r")
test_tree = test_file.read()
test_tree_sentence_list = test_tree.split("\n")
x_test_list, y_test_list = parse_tree(test_tree_sentence_list)
test_length = len(x_test_list)

"""# Data processing"""

class SentenceBatcher:
    def __init__(self, inputs, labels, batch_size=128, drop_last=False):
        # Tokenise the input sentences
        tokenised_inputs = np.array([sentence.split() for sentence in inputs])

        # Map the tokens to word2vec indices
        indices_mapping = [torch.tensor([token2index(token) for token in tokens]) for tokens
        ↪  in tokenised_inputs]

        # Store sentences by length
        self.sentences_by_length = {}
        for input, label in zip(indices_mapping, labels):
            length = input.shape[0]

            if length not in self.sentences_by_length:
                self.sentences_by_length[length] = []
            self.sentences_by_length[length].append([input, label])

        #  Create a DataLoader for each set of sentences of the same length
        self.loaders = {length : torch.utils.data.DataLoader(
                                 sentences,
                                 batch_size=batch_size,
                                 shuffle=True,
                                   drop_last=drop_last)
            for length, sentences in self.sentences_by_length.items()}

    def __iter__(self):
        # Create an iterator for each sentence length
        iters = [iter(loader) for loader in self.loaders.values()]
        while iters:
            # Get a random iterator
            i = random.choice(iters)
            try:
                yield next(i)
            except StopIteration:
                iters.remove(i)

"""# Data exploration"""

# Display example data for training set

start_index = 0
end_index = 30

for sentence, sentence in zip(y_train_list[start_index:end_index],
↪   x_train_list[start_index:end_index]):
  print(f"y = {sentiment} | x = ({sentence})")

# Number of examples in each dataset

print(f"Number of sentences training set: {len(x_train_list)}")
print(f"Number of sentences validation set: {len(x_dev_list)}")
print(f"Number of sentences test set: {len(x_test_list)}")

# Plot distribution of sentiment classifications by dataset
from collections import Counter

train_counter = Counter(y_train_list)
plt.bar(train_counter.keys(), train_counter.values())
plt.title("Number of examples per classification")
plt.ylabel('Count')
plt.xlabel('Classification')
plt.savefig(f"sentence_lengths.png", dpi=300)
plt.show()

dev_counter = Counter(y_dev_list)
plt.bar(dev_counter.keys(), dev_counter.values())
plt.title("Number of examples per classification")
plt.ylabel('Count')
plt.xlabel('Classification')
plt.savefig(f"sentence_lengths.png", dpi=300)
plt.show()

# Plot distribution of sentence lengths by dataset

train_sentence_lengths = Counter(map(lambda s : len(s.split()), x_train_list))
plt.bar(train_sentence_lengths.keys(), train_sentence_lengths.values())
plt.title("Number of tokens per sentence")
plt.ylabel('Count')
plt.xlabel('Number of tokens')
plt.savefig(f"sentence_lengths.png", dpi=300)
plt.show()

"""# Models"""

class RNN(nn.Module):
  def __init__(self, embedding_size, hidden_size, num_layers, num_classes, cell_type,
   ↪  num_dense_layers, hidden_to_dense_size_ratio=2, dropout=0, fixed_embeddings=True):
    super(RNN, self).__init__()
    self.num_layers = num_layers
    self.hidden_size = hidden_size
    self.cell_type = cell_type
    self.num_dense_layers = num_dense_layers

    self.embedding = nn.Embedding.from_pretrained(torch.FloatTensor(w2v.vectors))
    self.embedding.weight.requires_grad = not fixed_embeddings

    # Only one of these will be used in forward statement (as specified by hyperparam
    ↪  cell_type for current run)
    self.rnn = nn.RNN(embedding_size, hidden_size, num_layers, dropout=dropout, batch_first =
    ↪  True)
    self.gru = nn.GRU(embedding_size, hidden_size, num_layers, dropout=dropout, batch_first =
    ↪  True)
    self.lstm = nn.LSTM(embedding_size, hidden_size, num_layers, dropout=dropout, batch_first
    ↪  = True)

    # Number of these linear layers that gets used depends on hyperparam num_dense_layers
    if num_dense_layers == 1:
      self.linear = nn.Linear(hidden_size, num_classes)
    elif num_dense_layers == 2:
      self.linear1 = nn.Linear(hidden_size, hidden_size//hidden_to_dense_size_ratio)
      self.linear2 = nn.Linear(hidden_size//hidden_to_dense_size_ratio, num_classes)
      self.dropout = nn.Dropout(dropout)

  def forward(self, x_batch):
    # Get the word embeddings
    embedded = self.embedding(x_batch)

    if self.cell_type == 'VanillaRNN':
      packed_out, _ = self.rnn(embedded)
    elif self.cell_type == 'GRU':
      packed_out, _ = self.gru(embedded)
    elif self.cell_type == 'LSTM':
      packed_out, _ = self.lstm(embedded)

    # Extract the final output and feed into a dense layer
    out = packed_out[:, -1, :]
    if self.num_dense_layers == 1:
      out = self.linear(out)
    elif self.num_dense_layers == 2:
      out = self.linear1(out)
      out = torch.relu(out)
      out = self.dropout(out)
      out = self.linear2(out)
    return out
```

```python
"""# Model, loss, optimizer, and training loop"""

# Hyperparameters to grid search over
hyperparam_grid = {'cell_type': ['GRU'],
                   'num_layers': [3],
                   'num_dense_layers': [2],
                   'hidden_size': [128],
                   'hidden_to_dense_size_ratio': [2],
                   'learning_rate': [0.001],
                   'dropout': [0.25],
                   'fixed_embeddings': [True],
                   'batch_size': [128]
}

# Initialise dataframe to store results of grid search
results_df = pd.DataFrame()

# Range of epochs
MIN_EPOCHS = 20
MAX_EPOCHS = 100

for run_index, hyperparams in enumerate(ParameterGrid(hyperparam_grid)):

  # Get training and validation data loaders
  train_loader = SentenceBatcher(x_train_list, y_train_list, hyperparams['batch_size'],
  ↪  drop_last=False)
  dev_loader = SentenceBatcher(x_dev_list, y_dev_list, hyperparams['batch_size'],
  ↪  drop_last=False)

  # Initialise model
  model = RNN(embedding_size=300,
              hidden_size=hyperparams['hidden_size'],
              num_layers=hyperparams['num_layers'],
              num_classes=5,
              cell_type=hyperparams['cell_type'],
              num_dense_layers=hyperparams['num_dense_layers'],
              hidden_to_dense_size_ratio=hyperparams['hidden_to_dense_size_ratio'],
              dropout=hyperparams['dropout'],
              fixed_embeddings=hyperparams['fixed_embeddings'])
  device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
  model.to(device)

  # Loss & optimizer
  criterion = nn.CrossEntropyLoss() # note that this will apply the softmax for us
  optimizer = torch.optim.Adam(model.parameters(), lr=hyperparams['learning_rate'])

  # Training loop
  running_loss = 0
  running_corrects = 0
  training_accuracy_list = []
  training_loss_list = []

  best_epoch = 0
  best_train_accuracy = 0
  best_dev_accuracy = 0
```

```
dev_accuracy_list = []
dev_loss_list = []

converged = False
epoch = 0
epochs_to_plot = []

while (converged == False) and (epoch <= MAX_EPOCHS):

  model.train()

  running_loss = 0
  running_corrects = 0
  num_of_batches = 0

  # Generate batch
  for inputs, labels in train_loader:
    model.train()

    optimizer.zero_grad()

    # Send to GPU
    inputs = inputs.to(device)
    labels = labels.to(device)

    # Forward pass
    predicted = model(inputs)

    # Loss
    loss = criterion(predicted, labels)
    running_loss += loss.item()

    # Backward pass
    loss.backward()

    # Updates
    optimizer.step()

    num_of_batches +=1

    # Get number of correct training predictions per batch and sum
    _, train_pred = torch.max(predicted, 1)
    running_corrects += (train_pred == labels).sum().item()

  model.eval()
  with torch.no_grad():
    # Calculate training accuracy per epoch
    training_accuracy = running_corrects/train_length
    training_accuracy_list.append(training_accuracy)

    training_loss_list.append(running_loss/num_of_batches)

    # Evaluate on validation dataset

    dev_running_loss = 0
    dev_running_corrects = 0
    num_of_batches = 0

    for dev_inputs, dev_labels in dev_loader:

        # Send to GPU
        dev_inputs = dev_inputs.to(device)
        dev_labels = dev_labels.to(device)

        # Forward pass
        dev_predicted = model(dev_inputs)

        # Loss
        dev_loss = criterion(dev_predicted, dev_labels)
        dev_running_loss += dev_loss.item()

        num_of_batches +=1

        # Get number of correct validation predictions per batch and sum
        _, dev_pred = torch.max(dev_predicted, 1)
        dev_running_corrects += (dev_pred == dev_labels).sum().item()

    dev_accuracy = dev_running_corrects/dev_length
    dev_accuracy_list.append(dev_accuracy)

    dev_loss_list.append(dev_running_loss/num_of_batches)

    print(f"Epoch: {epoch}, train_loss: {loss.item():.4f}, train_acc:
    ↪   {training_accuracy:.4f}, val_loss: {dev_loss.item():.4f}, val_acc:
    ↪   {dev_accuracy:.4f}")

    epochs_to_plot.append(epoch)

    # Store the current results if it is the highest validation accuracy so far
    if dev_accuracy > best_dev_accuracy:
        best_dev_accuracy = dev_accuracy
        best_dev_loss = dev_loss.item()
        best_train_accuracy = training_accuracy
        best_loss = loss.item()
        best_epoch = epoch
        best_model = model

    # Early stopping criteria
    if epoch >= MIN_EPOCHS:
      if (np.mean(dev_loss_list[-15:]) - np.mean(dev_loss_list[-30:-15])) > 0:
        converged = True
        print("Model Converged")

  epoch += 1

hyperparams['model_index'] = run_index + 1
hyperparams['best_epoch'] = best_epoch
hyperparams['best_validation_accuracy'] = best_dev_accuracy
hyperparams['associated_validation_loss'] = best_dev_loss
hyperparams['associated_training_accuracy'] = best_train_accuracy
hyperparams['associated_training_loss'] = best_loss
```

```
    results_df = results_df.append(hyperparams , ignore_index=True)
    results_df = results_df.sort_values("best_validation_accuracy", ascending=False).round(4)
    results_df.to_csv(f"{path}RESULTS_TABLE_{hyperparams['cell_type']}.csv")

    print(f"\n RUN_INDEX: {run_index + 1} \n")
    print(hyperparams)

    # Save the model
    torch.save(best_model,
    ↪   f"{path}{hyperparams['cell_type']}_{best_dev_accuracy:.4f}_devacc_{best_train_accuracy:.4f}_trainacc

    # Loss plot
    plt.plot(epochs_to_plot, training_loss_list, color='k', linestyle='-')
    plt.plot(epochs_to_plot, dev_loss_list, color='r', linestyle='-')
    plt.legend(['Training', 'Validation'], loc='upper right')
    plt.ylabel('Loss', color='k')
    plt.xlabel('Epoch', color='k')
    plt.title(f'''Loss plot ({hyperparams['cell_type']}, Val Accuracy {best_dev_accuracy:.4f})
    ↪   \n
    Hidden size: {hyperparams['hidden_size']}, Recurrent layers: {hyperparams['num_layers']},
  ↪   Dense layers: {hyperparams['num_dense_layers']},
    Hidden to dense size ratio: {hyperparams['hidden_to_dense_size_ratio']}, LR:
  ↪   {hyperparams['learning_rate']}, Dropout: {hyperparams['dropout']},
    Fixed embeddings: {hyperparams['fixed_embeddings']}, Batch size:
  ↪   {hyperparams['batch_size']}''', color='k')


    ↪   plt.savefig(f"{path}{hyperparams['cell_type']}_{best_dev_accuracy:.4f}_devacc_{best_train_accuracy:.
    ↪   dpi=300, bbox_inches = "tight")
    plt.show()

    # Accuracy plot
    plt.plot(epochs_to_plot, training_accuracy_list, color='k', linestyle='-')
    plt.plot(epochs_to_plot, dev_accuracy_list, color='r', linestyle='-')
    plt.legend(['Training', 'Validation'], loc='lower right')
    plt.ylabel('Accuracy', color='k')
    plt.xlabel('Epoch', color='k')
    plt.title(f'''Accuracy plot ({hyperparams['cell_type']}, Val Accuracy
    ↪   {best_dev_accuracy:.4f}) \n
    Hidden size: {hyperparams['hidden_size']}, Recurrent layers: {hyperparams['num_layers']},
  ↪   Dense layers: {hyperparams['num_dense_layers']},
    Hidden to dense size ratio: {hyperparams['hidden_to_dense_size_ratio']}, LR:
  ↪   {hyperparams['learning_rate']}, Dropout: {hyperparams['dropout']},
    Fixed embeddings: {hyperparams['fixed_embeddings']}, Batch size:
  ↪   {hyperparams['batch_size']}''', color='k')


    ↪   plt.savefig(f"{path}{hyperparams['cell_type']}_{best_dev_accuracy:.4f}_devacc_{best_train_accuracy:.
    ↪   dpi=300, bbox_inches = "tight")
    plt.show()

"""# Evaluate on test dataset"""

# Load best model (highest validation accuracy) for further evaluation
```

```
best_model = torch.load(f"{path}GRU_0.4505_devacc_0.4896_trainacc.pth")
best_model.eval()

device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
best_model.to(device)

# Get test data loader
test_loader = SentenceBatcher(x_test_list, y_test_list, hyperparams['batch_size'],
↪  drop_last=False)

# Loss & optimiser
criterion = nn.CrossEntropyLoss() # note that this will apply the softmax for us

# Final performance of best model on test set

test_running_loss = 0
test_running_corrects = 0

for test_inputs, test_labels in test_loader:

    # Send to GPU
    test_inputs = test_inputs.to(device)
    test_labels = test_labels.to(device)

    # Forward pass
    test_predicted = best_model(test_inputs)

    # Test loss
    loss_test = criterion(test_predicted, test_labels)
    test_running_loss += loss_test

    # Test accuracy
    _, test_pred = torch.max(test_predicted, 1)
    test_running_corrects += (test_pred == test_labels).sum().item()

test_loss_avg = test_running_loss/test_length
test_accuracy_avg = test_running_corrects/test_length

# print results
print(f"Final test accuracy: {test_accuracy_avg}")
print(f"Final test loss: {test_loss_avg}")

"""# Online review classifications"""

# Online reviews, pre-processed as in parse_tree
x_online_list = [
                 "to this day this is still my favorite pixar film the animation is stellar
                 ↪  its heartwarming funny and proves that pixar movies are always bound to
                 ↪  be great except for cars but thats a different story this has a shot at
                 ↪  the title best movie of the century", #
                 ↪  https://www.imdb.com/review/rw5485122
                 "this is just a wonderful telling of charles dickens great christmas story
                 ↪  the story being so good you would have to try had to make a bad movie out
                 ↪  of it", # https://www.imdb.com/review/rw0310420
                 "honestly i really should be giving this film a lower score somehow i enjoyed
                 ↪  it quite a bit even in the face of the many fundamental issues which is a
                 ↪  testament to the strength of the best sequences", #
                 ↪  https://www.imdb.com/review/rw4075393
                 "but the worst thing of all with this film is the mangling of austen's
                 ↪  dialogue and the atrocious modern dialogue austen's dialogue needs no
                 ↪  assistance from a writer who thinks he she can write like austen", #
                 ↪  https://www.imdb.com/review/rw1213354
                 "hours of boredom half the audience fell asleep including most of the kiddies
                 ↪  beautiful to look at but that does not make for a interesting film" #
                 ↪  https://www.imdb.com/review/rw0717356
]
# SentenceBatcher will randomly order the reviews, so we keep track of their indices
x_online_index = range(len(x_online_list))

# Get online data loader
online_loader = SentenceBatcher(x_online_list, x_online_index, batch_size=128,
↪  drop_last=False)

# Evaluate model on online reviews

for online_inputs, online_index in online_loader:
  online_inputs = online_inputs.to(device)

  online_predicted = best_model(online_inputs)
  _, online_pred = torch.max(online_predicted, 1)

  for index, pred in zip(online_index, online_pred):
    print(f"Review: {x_online_list[index]}")
    print(f"Prediction: {pred}")
```