Post Block Assessment 3

Francois van Zyl : 18620426

24/07/2020

Convolutional Neural Network (CNN) [44]

Question 1) What are the four most important layered concepts in convolutional neural networks (CNNs)? Briefly discuss the functionality of each of these in a CNN. [8]

1. In contrast to standard dense layers, convolutional layers are employed within CNNs to learn local patterns within the input feature space, where the former layer would typically learn global patterns. The convolutional layer achieves this by using convolution; a mathematical operator that enables the filtering of spatial frequency of images by sliding convolutional windows over the input feature space which outputs some transformation of the initial image. In the case of image related operations, the convolutional layer is therefore a set of filters that modify the existing image that allows for a efficient and generally powerful form of learning local patterns from target space, which is the main reason why CNNs are infamous within the field of computer vision. The purpose of training CNN's is therefore to adjust these aforementioned filters, such that the network can most adequately identify the relevant local patterns, and these layers are stacked to typically identify low and high level features as one moves deeper into the network respectively.

2. ReLu activation functions are considered the standard selection of activation functions within CNNs, as they exhibit three beneficial charactersistics. ReLu activation functions are generally computationally less expensive than other activation functions, and due to their shape they can be used to introduce non-linearity after the convolutional layer, which has been proved to be beneficial to the performance of CNNs. Recalling that the errors are backpropagated with weight gradient computation, the ReLu activation functions also solve the problem of the vanishing gradient problem.

3. Pooling layers are layers that are introduced after convolutional layers that serve to reduce the dimensionality of the output received from the convolutional layer. More specifically, pooling layers reduce the spatial size of the convoluted representation received, effectively reducing the number of parameters and computational expense, which simultaneously allows for the extraction of the most important features such that the model can be more efficiently trained.

4. Fully connected layers are layers that contain neurons that are fully connected to all output layers received from the convolutional and pooling layers. This effectively enables the model to learn non-linear mappings received from the output of the learnt features, which enables the model to perform classification tasks. The convolutional network can be thought of as a dual-network, where the first section of the network is responsible for the extraction of important features, and the fully-connected section is the section of the network that enables the network to perform some classification regarding these features.

Question 2) What does Pooling do in a CNN? Apply max pooling to the feature map in the figure below with a 2x2 filter and stride of 2. Create a figure showing the resulting pooled feature map. [2] Pooling layers reduce the spatial size of the convoluted representation received. Two main types of pooling exist, namely max pooling and average pooling. Max pooling extracts the maximum value found within the specified pooling size window that is scanned over the output of the convoluted layer, alternatively referred to as the feature map, and average pooling returns the average value found within this same pooling layer from the feature map output from the convolutional layer. Furthermore, a stride length is specified, which indicates how finely the window will move along the output received from the convoluted layer in the horizontal and verical directions. This allows for control over the movement of scanning sections of the output of the convolutional layer that have been scanned already. This is perhaps explained best visually, and by inspecting the array specified, a max pooling layer with a size of 2 x 2 is applied with a stride length of 2.

```
setwd("/Users/jd-vz/Desktop/MEng/Deep Learning/Assignments/PBA3/")
library(keras)
data <- array(data = c(9,0,1,3,3,2,3,0,5,0,4,5,0,1,1,1), dim = c(4,4))#By col
arr <- array_reshape(data, dim = c(1,4,4,1))
model <- keras_model_sequential() %>% layer_max_pooling_2d(pool_size = c(2,2), strides = 2)
result <- model %>% predict(arr)
result <- array_reshape(result, dim = c(2,2))
knitr::kable(data,  caption = "Initial Feature Map")
```

Table 1: Initial Feature Map

9	3	5	0
0	2	0	1
1	3	4	1
3	0	5	1

```
knitr::kable(result, caption = "Pooled Feature Map")
```

Table 2: Pooled Feature Map

9	5
3	5

The initial and max pooled feature maps are displayed above. The inital feature map can be seen to be 4 x 4, and a 2 x 2 pooled map with a stride of 2 was scanned over it. By splitting the initial feature map into 4 equal sections, and selecting the maximum found within each section, the output from the pooled layer can be quantified. Note that the pooling layer moved from the first two upper sections with no overlap, before moving on to the bottom two sections. This is attributed to the stride of the pooling layer.

Question 3) Suppose, for a CNN, the input size is 6 $\times$ 6 and the filter size is 8 $\times$ 8. What would the size of the output be? [2] Assuming valid padding is used, the size of the output is given by $(n+2p-f+1)(n+2p-f+1) = (6+2*0-8+1)(6+2*0-8+1)$ which would result in an output of -1 x -1. This is clearly infeasible and is due to the filter size exceeding the input size without the addition of any sort of padding. However, if **same** padding is used, this will be rectified since $2p = (f-1) = 7$, and therefore the input will be the same as the output, as seen by calculating the following equation $(n+2p-f+1)(n+2p-f+1)$ $= (6+7-8+1)(6+7-8+1) = 6$ x 6

Question 4) Explain what the difference is between valid padding and same padding is in a CNN? [2] Padding is required since the process of convolution could reduce or even enlarge the size of an image depending on the size of the input and filter. This could lead to potential information loss as well as an explosion in dimensionality. To overcome this, valid and same padding can be applied. Same padding alters the dimensions of the matrix or image with so-called padding to prevent information being lost and keeps the input dimensions the same as the output dimensions. If this reduction in dimensionality is desirable, valid padding can be applied as it does not alter the dimensions of the matrix or image with any form of padding.

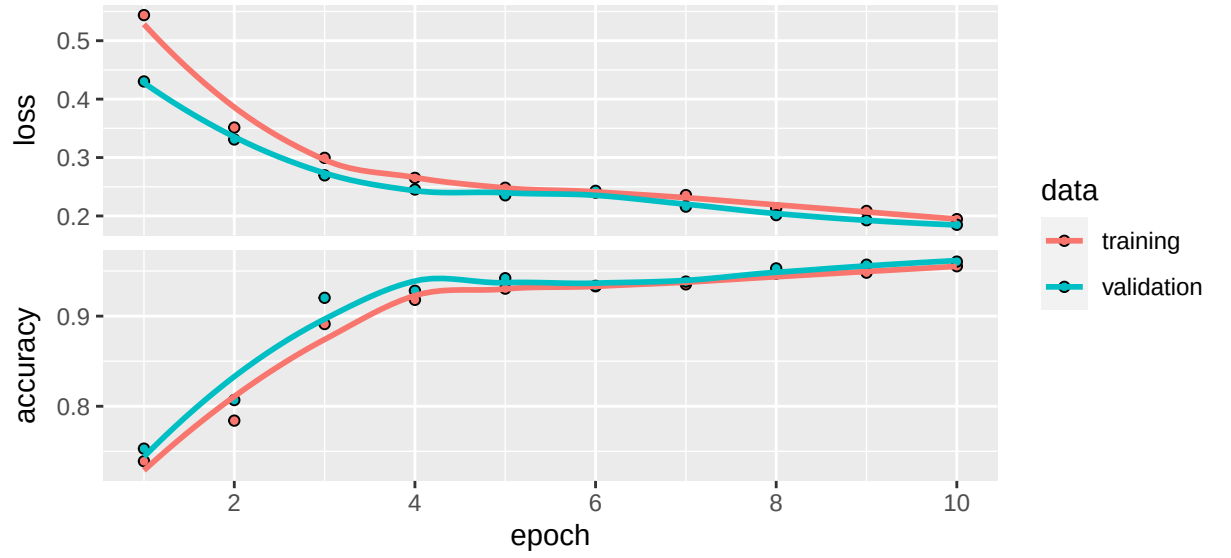
Question 5.1) In this question, we will develop a CNN to detect whether a plane is present in an image or not. The data set consists of 32000 images of which 8000 have planes (of different shapes and sizes) in the image. Images with planes have a label of 1, whereas images without a plane have a label of 0. An 80/20 split should be used for training and testing your developed CNN model. Alternatively, download the data set called planes.json from SunLearn and remember to shuffle the data set before generating the train and test data sets. I researched how to efficiently develop a base CNN model, and found the following **article**. I used this site to help me to set up some form of guideline to developing a baseline model. I drew my initial inspiration for designing the base model from the architecture of the LeNet CNN, such that the layer orders are as follows: convolutional - pooling - convolutional - pooling - fully connected - output layer. The amount of filters that should be employed at the convolutional layers seem to be problem specific, but the general consensus remains that the dimensionality of the output of convolutional layers should stacked in an increasing manner, such as to capture low-level information in the early layer, and more high-level information in the latter layers. The amount of filters are problem specific, but 32-64-128 seems to be a popular choice. Since I can always add layers, I initially opted for 32 and 128 filter sizes at the two convolutional layers. I elected to use 3x3 convolution windows, and 2x2 max pooling filters with a stride size of 2, which are considered standard selections sizes for small-moderately sized images. Same padding will be applied to the two convolutional layers. This will ensure that the input and output size remains the same. All intermediate layers will employ ReLu activation functions, except the final output layer. The classification section of the network will consist of a dense layer connected to the output layer, which received flattened arrays from the convolutional part of the network. The output layer needs to distinguish between two distinct labels, and therefore a sigmoid activation function was chosen at this layer since this is a binary classification problem.
Prior to developing the model, I read in, shuffled and normalized the dataset that was provided. The training and testing arrays were also reshaped to 20 x 20 images with 3 channels. Furthermore the epoch and batch size was selected as 10 and 512 respectively.

```
# Question 1.5) ------------------------------------------------------------
#Read in, shuffle and normalize data
setwd("/Users/jd-vz/Desktop/MEng/Deep Learning/Assignments/PBA3/")
rm(list = ls())
library(keras)
planes <- rjson::fromJSON(file = "planes.json")
epochs <- 10
batch_size <- 512
json_length <- 32000 #Total entries
data <- planes$data #[0, 1]
labels <- planes$labels #{0, 1}
train_ind <- sample(x = json_length, size = 0.8*json_length, replace = F)
x_train <- as.matrix(unlist(data[train_ind]))/255 #Shuffle and normalize
x_test <- as.matrix(unlist(data[-train_ind]))/255
y_train <- as.matrix(unlist(labels[train_ind]))
y_test <- as.matrix(unlist(labels[-train_ind]))
#Reshape data
inp_size <- c(20, 20, 3)
```

```
dim_train <- c(25600, inp_size)
dim_test <- c(6400, inp_size)
x_train <- array_reshape(x = x_train, dim = dim_train)
x_test <- array_reshape(x = x_test, dim = dim_test)
```

After preprocessing the data, the base model was constructed as follows. The model contains an input layer of shape 20 x 20 x 3, a convolutional 2D layer, with 32 filters and a kernel size of 3 x 3 and relu activation function, and same padding, a max pooling 2D layer, with a pooling size of 2 x 2 and a stride length of 2, a convolutional 2D layer, with 128 filters and a kernel size of 3 x 3 and relu activation function, and same padding. The output of the feature extraction side of the network is then flattened, and fed to a densely connected layer containing half the previous convolutional layers filters [chosen arbitrarily], that feeds into a final output layer employing a sigmoid activation function that will perform the binary classification. he model was compiled using binary crossentropy as a loss function, the adam optimizer, and accuracy as a metric. These are typical selections for a binary classification problem, and are not the only options that are available. The model was fit upon the training data and tested upon the validation data with 10 epochs and a batch size of 512. The performance metrics are displayed afterwards.

```
#Construct model
model.cnn <- keras_model_sequential() %>%
  layer_conv_2d(filters = 32, kernel_size = c(3,3), activation = 'relu', input_shape = inp_size,
                padding = "same") %>%
  layer_max_pooling_2d(pool_size = c(2, 2), strides = c(2,2)) %>%
  layer_conv_2d(filters = 128, kernel_size = c(3,3), activation = 'relu', padding = "same") %>%
  layer_max_pooling_2d(pool_size = c(2, 2), strides = c(2,2)) %>%
  layer_flatten() %>% #Flatten arrays for full connected
  layer_dense(units = 64, activation = 'relu') %>%
  layer_dense(units = 1, activation = 'sigmoid')
#Compile model
model.cnn %>% compile(
  loss="binary_crossentropy",
  optimizer = optimizer_adam(),
  metrics = c('accuracy')
)
#Train model
history.cnn <- model.cnn %>% fit(
  x_train, y_train,
  batch_size = batch_size,
  epochs = epochs,
  validation_data = list(x_test, y_test),
  verbose = 0
)
#Plot performance metrics
plot(
  history.cnn,
  method = "ggplot2",
  smooth = getOption("keras.plot.history.smooth", T)
)
```




```
val.acc1 <- history.cnn$metrics$val_accuracy[epochs]
val.loss1 <- history.cnn$metrics$val_loss[epochs]
```

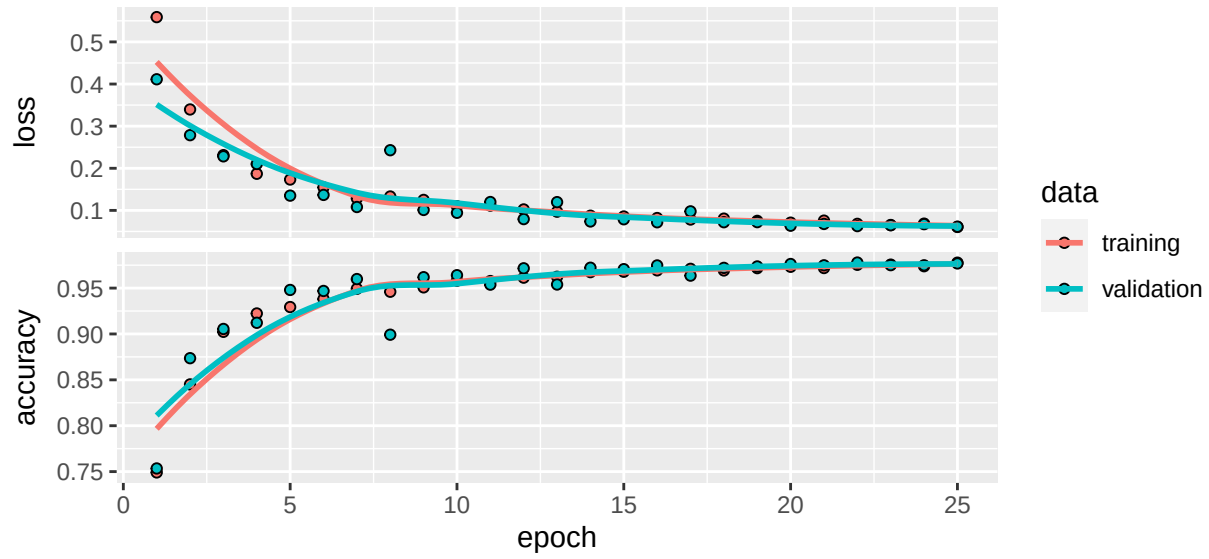
As seen above, the base model performs quite nicely. It receives a validation accuracy of around 96% after 10 epochs and I accepted it as my baseline model.

Question 5.1) ii. Improve the model developed in (i) by implementing at least 3 techniques which are commonly used to improve network performance in CNNs. Discuss and illustrate the effect of the techniques used on the performance of the model. Consider other classification performance metrics (such as F1 score, recall) in addition to training and testing accuracy (and loss) when discussing the performance of the various candidate models. [15] The baseline model design process seemed to work well, and I concluded that if I could alter the model such that it reaches a validation accuracy of 98% I will accept the model as the new baseline. I noted that my baseline model never did overfit on the training data, indicating that it can be trained further to improve it's performance. The following steps were applied to improve the model.

1) Increase training time by changing the epochs to 25
2) Increase classification hidden units from 64 to 512 to increase feature extraction capacity
3) Apply dropout with p = 0.1 [chosen by trial] to avoid overfitting and increase efficiency
4) Apply L2 regularization with a factor of 1e-6 to introduce weighting penalties at convolutional layers
5) Increase model complexity by adding in another convolutional layer with 64 filters and similar parameters to the final convolutional layer.
6) Apply early stopping with a patience of 5 to stop the model if it overfits.

```
# Increase epochs
epochs <- 25
# Construct model
model.better <- keras_model_sequential() %>%
  layer_conv_2d(filters = 32, kernel_size = c(3,3), activation = 'relu', input_shape = inp_size,
                padding = "same", kernel_regularizer = regularizer_l2(l = 1e-6)) %>%
  layer_max_pooling_2d(pool_size = c(2, 2), strides = c(2,2)) %>%
  layer_dropout(0.1) %>%  #Add dropout
  layer_conv_2d(filters = 64, kernel_size = c(3,3), activation = 'relu', padding = "same",
```

```
                kernel_regularizer = regularizer_l2(l = 1e-6)) %>% #64 filters
  layer_max_pooling_2d(pool_size = c(2, 2), strides = c(2,2)) %>%
  layer_dropout(0.1) %>%
  layer_conv_2d(filters = 128, kernel_size = c(3,3), activation = 'relu', padding = "same",
                kernel_regularizer = regularizer_l2(l = 1e-6)) %>% #lambda = 1e-6
  layer_max_pooling_2d(pool_size = c(2, 2), strides = c(2,2)) %>%
  layer_dropout(0.1) %>%
  layer_flatten() %>% #512 hidden units
  layer_dense(units = 512, activation = 'relu') %>%
  layer_dense(units = 1, activation = 'sigmoid')
# Compile model
model.better %>% compile(
  loss="binary_crossentropy",
  optimizer = optimizer_adam(),
  metrics = c('accuracy')
)
#Fit model
history.better <- model.better %>% fit(
  x_train, y_train,
  batch_size = batch_size,
  epochs = epochs,
  validation_data = list(x_test, y_test),
  verbose = 0,
  callback_early_stopping(monitor = "accuracy", patience = 5)
)
#Investigate performance metrics
plot(
  history.better,
  method = "ggplot2",
  smooth = getOption("keras.plot.history.smooth", T)
)
```




```
val.acc2 <- history.better$metrics$val_accuracy[epochs]
val.loss2 <- history.better$metrics$val_loss[epochs]
```

```r
df <- data.frame(Baseline=c(val.acc1, val.loss1), Improved = c(val.acc2, val.loss2))
rownames(df) <- c("Validation accuracy", "Validation loss")
knitr::kable(df, row.names = T) #Validation
```

	Baseline	Improved
Validation accuracy	0.9604688	0.9768750
Validation loss	0.1848910	0.0612684

The validation and losses are displayed above, and the new model can be seen to outperform the baseline model in every aspect. The training accuracy and losses for these models are also displayed below. The improved model also outperforms the initial model here.

```r
train.acc1 <- history.cnn$metrics$accuracy[10]
train.loss1 <- history.cnn$metrics$loss[10]
train.acc2 <- history.better$metrics$accuracy[epochs]
train.loss2 <- history.better$metrics$loss[epochs]
df <- data.frame(Baseline = c(train.acc1, train.loss1), Improved = c(train.acc2, train.loss2))
rownames(df) <- c("Training accuracy", "Training loss")
knitr::kable(df, row.names = T) #Training
```

	Baseline	Improved
Training accuracy	0.9553125	0.9776953
Training loss	0.1946050	0.0606214

More detailed testing performance measures are displayed below in the classication report.

```r
# Baseline model ----------------------------------------------------------------
caret::confusionMatrix(as.factor(y_test), as.factor(model.cnn %>% predict_classes(x = x_test)))
```

```
## Confusion Matrix and Statistics
##
##           Reference
## Prediction    0    1
##          0 4743   76
##          1  177 1404
##
##                Accuracy : 0.9605
##                  95% CI : (0.9554, 0.9651)
##     No Information Rate : 0.7688
##     P-Value [Acc > NIR] : < 2.2e-16
##
##                   Kappa : 0.8914
##
##  Mcnemar's Test P-Value : 3.238e-10
##
##             Sensitivity : 0.9640
##             Specificity : 0.9486
##          Pos Pred Value : 0.9842
```

```
##          Neg Pred Value : 0.8880
##              Prevalence : 0.7688
##          Detection Rate : 0.7411
##    Detection Prevalence : 0.7530
##       Balanced Accuracy : 0.9563
##
##        'Positive' Class : 0
##
```

```
# Improved model ------------------------------------------------------------
caret::confusionMatrix(as.factor(y_test), as.factor(model.better %>% predict_classes(x = x_test)))
```

```
## Confusion Matrix and Statistics
##
##           Reference
## Prediction    0    1
##          0 4712  107
##          1   41 1540
##
##                Accuracy : 0.9769
##                  95% CI : (0.9729, 0.9804)
##     No Information Rate : 0.7427
##     P-Value [Acc > NIR] : < 2.2e-16
##
##                   Kappa : 0.9387
##
##  Mcnemar's Test P-Value : 9.144e-08
##
##             Sensitivity : 0.9914
##             Specificity : 0.9350
##          Pos Pred Value : 0.9778
##          Neg Pred Value : 0.9741
##              Prevalence : 0.7427
##          Detection Rate : 0.7362
##    Detection Prevalence : 0.7530
##       Balanced Accuracy : 0.9632
##
##        'Positive' Class : 0
##
```

The classification report reveals the following.

1) **Accuracy**: The improved model has a lower total misclassification rate for both classes.
2) 95% CI: The improved model has a higher confidence interval, implying it's accuracy is more likely to fall in a higher range than the base model.
3) NIR: The NIR refers to the largest class percentage within the dataset.
4) P-value: This is a one-tailed test that is performed to check that the accuracy is significantly larger than the NIR. A null hypothesis is set up, and is defined such that the model can be used to predict the data. The p-value is far below the LoS and therefore the null hypothesis can be accepted for both models.
5) Kappa: Cohen's kappa statistic proves further insight into the misclassification can be seen to be higher in the initial model, simply implying that the ratio of observed and expected accuracy $p_o - p_e / 1 - p_e$ is higher for the initial model.

6) Sensitivity/**Recall**/TPR: The sensitivity can be seen to be higher for the initial model, and it reflects the ability of the model to predict the classes as the positive class if it truly is the positive class.
7) **Specificity**/TNR: The specificity can be seen to be higher for the initial model, and it reflects the ability of the model to predict the classes as not being part of the positive class if it truly is not part of the positive class.
8) PPV/Precision: The precision measures the probability that the positive and negative classes are assigned correctly. The PPV can be seen to have risen for the improved model.
9) NPV: The NPV measures the probability that the positive and negative classes are correctly not assigned to a class. The NPV can be seen to have decreased for the improved model.
10) Prevalence: The prevalence indicates what proportion of the validation set belongs to the positive class.
11) Detection rate: The detection rate is a holistic class-specific accuracy of the classes that were correctly classified as belonging to the postiive class. The detection rate can be seen to have increased in the improved model.
12) Detection prevalence: The detection prevalence is a measure that provides indication into how often a certain class occurred within the predictions.
13) Balanced Accuracy: The balanced accuracy is simply a weighted accuracy that disregards skew class distributions. The balanced accuracy can be seen to have increased with the improved model.

These measures all indicate that the improved model is better. The final performance metric that I investigated was the **F-measure**, which is displayed below. This measure is a harmonic mean between precision and recall, and can be seen to have increased for the improved model.

```
knitr::kable(data.frame(Baseline = caret::F_meas(as.factor(y_test), as.factor(model.cnn %>%
                                                                                 predict_classes(
                                                                                   x = x_test))),
                        Improved = caret::F_meas(as.factor(y_test), as.factor(model.better %>%
                                                                                 predict_classes(
                                                                                   x = x_test)))))
```

Baseline	Improved
0.974022	0.9845382

It is therefore clear that the improved model is the better model, and upon inspecting the confusion matrix that is presented above we can see that the improved model makes less mistakes in incorrectly classifying images with class label 1 as class label 0's. The opposite is also true where images containing planes are less frequently classified incorrectly in the improved model. After considering all these factors, I accepted the improved model as my new baseline model. Since it contains approximately the 98% testing accuracy and F-measure that I aimed for I accepted it as my new baseline and stopped the development process.

Long Short Term Memory (LSTM) networks [45]

Question 1) Why do LSTMs (and RNNs) work better with text data than other neural networks? [2] Normal feedforward neural networks have no method of relating the input to output such that relationships within sequential data can be fully grasped, and therefore typically operate on word frequency's and do not consider dependencies. Since speech is highly dependent on previous words to understand intent and potential words that can be placed within a sentence, feedforward neural networks have limited capacity to operate on text. In contrast, LSTMs and RNNs both have unique measures implemented through their designed architecture that enable them to model sequential and time-dependent data. Through their specific

connection methods, LSTM and RNN's have the capacity to model dependencies within text which allows for more insight into text than stock feedforward neural networks can provide.

Question 2) Briefly discuss how LSTM addresses the vanishing gradient problem. [3] For an MLP the weight update that a specific layer recieves exists as a multiple of the selected learning rate, as well as the error signal returned from the previous layer, and the input to that layer. This error signal is calculated as some product of the derivative of the activation function, and since multiples of many small values are even smaller values, the error signal diminishes and training ceases. Although this error signal is still used during the backpropagation of LSTM networks, it never achieves these malicious small values due to the structure of the LSTM cell. The LSTM cell is comprised of smaller parts, namely the internal memory state, input gate, output gate, and forget gate. The error gradient returned contains the adddition of the forget gate, which implies that the product of multiple error gradients is dictated by the product of multiple sums. This allows for easy balancing of the additive equation, which is balanced by the forget gate such that the value of the error gradient return does not decrease so low as to ensure that the VGP does not occur. It is due to this reason that forget gates are typically initialized to a vector of ones, implying everything should be remembered and ensuring that the VGP will not be an issue at the starting of training.

Question 3) In this question, we consider a multi-class text classification problem. The well-known Consumer Complaints data set is used. A smaller version (consisting of about 39000 complaints) has been created for the purpose of this exercise. The data consists of 11 categories of complaints. Prior to developing the model, I first preprocessed the training text. I accomplished this by fitting a tokenizer to the data such that a tokenized dictionary could be set up of all the words within the text. Thereafter, I used this dictionary and applied it to the text such that the sequences of integers representing the words could be set up. Note that there are 43125 unique tokens/words within this dataset, *but only the largest 1000 are being used.*

Since not all the documents are of the same length, I applied post padding and truncation to ensure that no documents are shorter nor longer than others. Also as this is a multi-class classification problem, the target variable was one-hot encoded into an 11-dimensional matrix. During my research in how to develop an acceptable baseline, I found a **site** which gave me some pointers on how to develop a suitable LSTM baseline model. The model was developed as follows, where the layers are displayed sequentially. Note that this design will be used for the RNN model too, with the exception of the LSTM layer being changed to a recurrent layer.

1) Embedding layer: This layer is responsible for embedding the sequences of words into embedding vectors and acts as the input layer. This layer was set up with the input dimensions as the size of the vocabulary being used, which was chosen as the 1000 most frequent words. The output dimensions were chosen to be the same as the LSTM layer. The input length was chosen to be the maximum length of the input sequences received.
2) LSTM layer: Chosen to be the same size as that of the embedding output dimensions, which was chosen as 32 based upon similar applications. During the design process, I opted for a bidirectional LSTM architecture. This bidirectional architecture improved the performance of my model and essentially fed the sequences into two identical LSTM layers: one with normal and another with reversed sequences. Referring to an **article**, bidirectional LSTMS are quite established in the field of NLP and have exceeded unidirectional LSTMs in word sequence prediction problems. In some sense, this bears similarities with data augmentation as it presents altered input data in conjunction with normal data, albeit this operates in a different manner. Recurrent dropout and normal dropout were chosen with a probability of 0.1 such that both input transforming and recurrent state transforming units would be dropped.
3) Dense layer: The last two layers are responsible for the classification, and a ssomewhat standard selection of 512 was chosen at this layer with a relu activation function.
4) Output layer: This layer was setup with units corresponding to the amount of output classes with an activation of softmax to output class probabilities.

The model was compiled with an adam optimizer and categorical crossentropy as the loss function. The preprocessing and setup of the data sets is displayed below, with snippets from the dictionary, the total amount of words and documents within the dataset, as well as the length of sequences after padding.

```
# Question 2.3 ------------------------------------------------------------
setwd("/Users/jd-vz/Desktop/MEng/Deep Learning/Assignments/PBA3/")
rm(list = ls())
library(keras)
set.seed(0)
# Read in and set parameters ----------------------------------------------
vocab <-  1000
embedding_dim <-  32
max_length = 500
epochs <- 5
internal_layer <- 100
complaints <- read.csv("Complaints.csv")
# Create tokenizing dictionary from text ----------------------------------
tokenizer <-  text_tokenizer(num_words = vocab)
tokenizer %>% fit_text_tokenizer(complaints$consumer_complaint_narrative)
knitr::kable(data.frame(head(tokenizer$word_index, 10)))
```

xxxx	the	i	to	and	a	my	of	that	was
1	2	3	4	5	6	7	8	9	10

```
knitr::kable(data.frame(TotalWords = length(tokenizer$index_word), NumWords = tokenizer$num_words,
                        NumDocs = tokenizer$document_count))
```

TotalWords	NumWords	NumDocs
43125	1000	38909

```
# Use dictionary to set up sequences of words-----------------------------
train_seq <- tokenizer %>% texts_to_sequences(complaints$consumer_complaint_narrative)
# Pad sequences -----------------------------------------------------------
train_pad <- pad_sequences(train_seq, maxlen = max_length, padding = "post", truncating = "post")
knitr::kable(data.frame(First = length(train_pad[1,]), Second = length(train_pad[2,]),
                        Third = length(train_pad[3,])))
```

First	Second	Third
500	500	500

```
# Set Up Train/Valid/Test -------------------------------------------------------
target <- to_categorical(complaints$category_id)
idx <- sample(nrow(train_pad), 0.8*nrow(train_pad), replace = F) # Train
for_eval <- as.numeric(complaints$category_id[-idx])
x_train <- train_pad[idx, ]
y_train <- target[idx,]
vidx <- sample(nrow(x_train), 0.1*nrow(train_pad), replace = F) # Valid
```

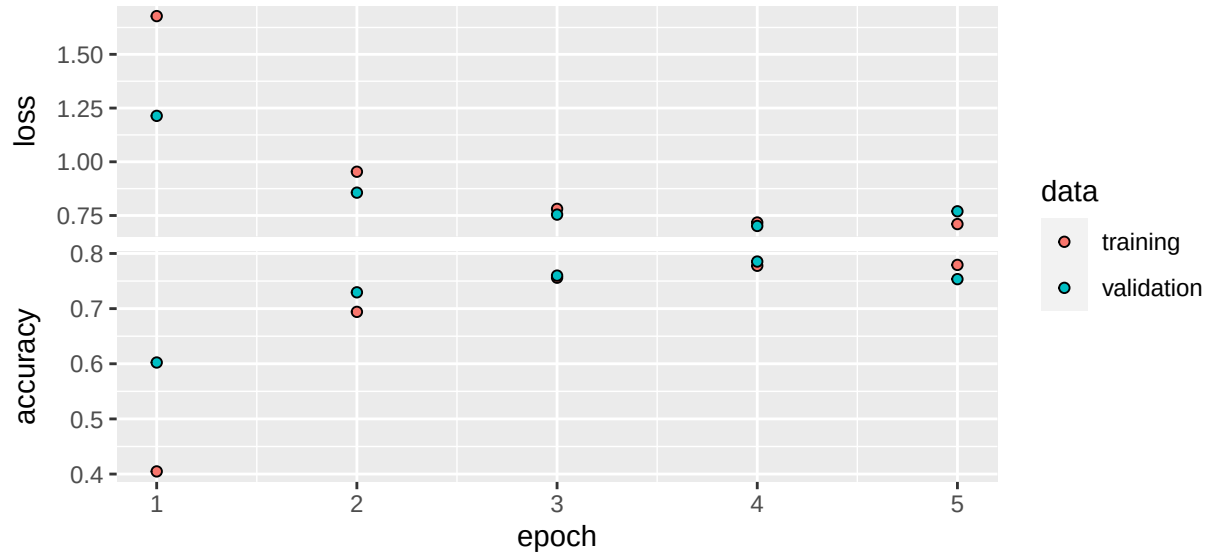
```
x_valid <- x_train[vidx,]
y_valid <- y_train[vidx,]
x_train <- x_train[-vidx,]
y_train <- y_train[-vidx,]
x_test <- train_pad[-idx,]
y_test <- target[-idx,]
```

After performing the preprocessing, I proceeded to develop the LSTM and RNN models according to the steps mentioned above. Their design was performed identically.

```
# LSTM model ------------------------------------------------------------
# Construct model -------------------------------------------------------
model.lstm <- keras_model_sequential() %>%
  layer_embedding(input_dim = vocab, output_dim = embedding_dim, input_length = max_length) %>%
  bidirectional(layer = layer_lstm(units = internal_layer, recurrent_dropout = 0.1,
                                   dropout = 0.1)) %>%
  layer_dense(512, activation = "relu") %>%
  layer_dense(11, activation = "softmax")
# Compile model ---------------------------------------------------------
model.lstm %>% compile(
  optimizer = "adam",
  loss = "categorical_crossentropy",
  metrics = c("accuracy")
)
# Fit model -------------------------------------------------------------
history.lstm <- model.lstm %>% fit(x_train, y_train, epochs = epochs, batch_size = 256,
                                   validation_data = list(x_valid, y_valid))
```

The performance metrics are displayed above. The model took quite a substantial time to train and I could not improve the performance further. I assumed that if a more sophisticated architecture and larger vocabulary was used it would achieve higher accuracies. However, it did achieve an acceptable accuracy and I proceeded to investigate the performance metrics and classification report as follows.

```
# Plot metrics ----------------------------------------------------------
plot(history.lstm)
```



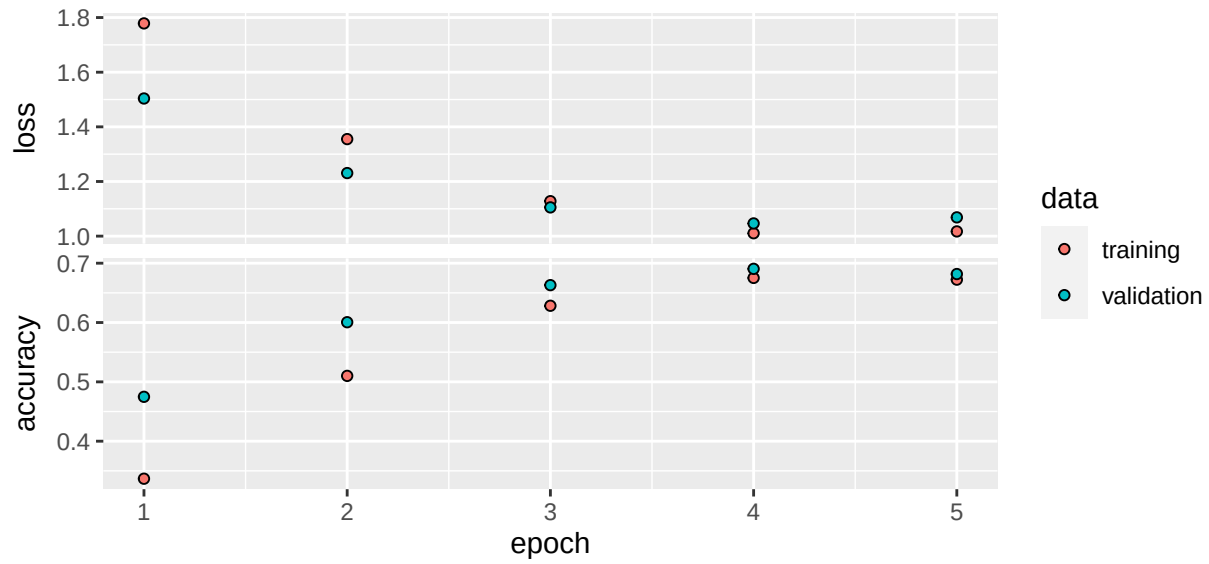
```r
# Generate classification report ----------------------------------------
actual <- for_eval %>% as.factor()
pred.lstm <- as.numeric(model.lstm %>% predict_classes(x = x_test)) %>% as.factor()
levels(actual) <- levels(pred.lstm) <- c(0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10)
caret::confusionMatrix(reference = actual, data = pred.lstm)
```

```
## Confusion Matrix and Statistics
##
##           Reference
## Prediction   0   1   2   3   4   5   6   7   8   9  10
##         0  916  64  34  29 111  50   7  22   1   3   1
##         1    5  45  12   5  16   8   5   9   1   0   0
##         2   15  12 876   3  14   2   7   0   1   0   0
##         3   47  29   3 322  48   0  72   5  17   1  24
##         4   51  28  11  24 535   3   6   0   1   0   0
##         5    2  15  16   2   2  66   1   7   0   0   0
##         6    5  14  17  24   0   1 188   4  22   1   3
##         7    0   0   0   0   0   0   0   0   0   0   0
##         8    0   0   0   0   0   0   0   0   0   0   0
##         9    0   0   0   0   0   0   0   0   0   0   0
##         10   0   0   0   0   0   0   0   0   0   0   0
##
## Overall Statistics
##
##                Accuracy : 0.7576
##                  95% CI : (0.7439, 0.771)
##     No Information Rate : 0.2675
##     P-Value [Acc > NIR] : < 2.2e-16
##
##                   Kappa : 0.6973
##
##  Mcnemar's Test P-Value : NA
##
## Statistics by Class:
##
##                      Class: 0 Class: 1 Class: 2 Class: 3 Class: 4 Class: 5
## Sensitivity            0.8799  0.21739   0.9040  0.78729   0.7369  0.50769
## Specificity            0.8870  0.98344   0.9815  0.92935   0.9608  0.98804
## Pos Pred Value         0.7399  0.42453   0.9419  0.56690   0.8118  0.59459
## Neg Pred Value         0.9529  0.95720   0.9686  0.97382   0.9409  0.98307
## Prevalence             0.2675  0.05320   0.2490  0.10511   0.1866  0.03341
## Detection Rate         0.2354  0.01157   0.2251  0.08276   0.1375  0.01696
## Detection Prevalence   0.3182  0.02724   0.2390  0.14598   0.1694  0.02853
## Balanced Accuracy      0.8835  0.60042   0.9428  0.85832   0.8489  0.74786
##                      Class: 6 Class: 7 Class: 8 Class: 9 Class: 10
## Sensitivity           0.65734  0.00000  0.00000 0.000000  0.000000
## Specificity           0.97476  1.00000  1.00000 1.000000  1.000000
## Pos Pred Value        0.67384      NaN      NaN      NaN       NaN
## Neg Pred Value        0.97287  0.98792  0.98895 0.998715  0.992804
## Prevalence            0.07350  0.01208  0.01105 0.001285  0.007196
## Detection Rate        0.04832  0.00000  0.00000 0.000000  0.000000
## Detection Prevalence  0.07170  0.00000  0.00000 0.000000  0.000000
## Balanced Accuracy     0.81605  0.50000  0.50000 0.500000  0.500000
```

Thereafter, I proceeded to develop the RNN model as follows. Note that the architecture is identical.

```
#RNN Model ------------------------------------------------------------------
# Construct model -----------------------------------------------------------
model.rnn <- keras_model_sequential() %>%
  layer_embedding(input_dim = vocab, output_dim = embedding_dim, input_length = max_length) %>%
  bidirectional(layer = layer_simple_rnn(units = internal_layer, recurrent_dropout = 0.1,
                                         dropout = 0.1)) %>%
  layer_dense(512, activation = "relu") %>%
  layer_dense(11, activation = "softmax")
# Compile model -------------------------------------------------------------
model.rnn %>% compile(
  optimizer = "adam",
  loss = "categorical_crossentropy",
  metrics = c("accuracy")
)
# Fit model -----------------------------------------------------------------
history.rnn <- model.rnn %>% fit(x_train, y_train, epochs = epochs, batch_size = 256,
                                 validation_data = list(x_valid, y_valid))
```

```
# Plot metrics --------------------------------------------------------------
plot(history.rnn)
```




```
# Generate classification report --------------------------------------------
pred.rnn <- model.rnn %>% predict_classes(x = x_test) %>% as.factor()
levels(pred.rnn) <- c(0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10)
```

```
caret::confusionMatrix(reference = actual, data = pred.rnn)
```

```
## Confusion Matrix and Statistics
##
##           Reference
## Prediction   0   1   2   3   4   5   6   7   8   9  10
##         0  736  68  18  13  58  48   2  16   2   2   0
##         1    3  12   6   1   4   5   0   0   1   0   0
##         2   26  44 896   8  10  34  32   5   3   0   0
##         3   17   9   7 208  15   1  89   5  14   0  22
##         4  240  55  22 144 635   2  22   6   7   2   2
##         5   11  15  11   1   3  40   1  12   1   0   0
##         6    8   4   9  34   1   0 140   3  15   1   4
##         7    0   0   0   0   0   0   0   0   0   0   0
##         8    0   0   0   0   0   0   0   0   0   0   0
##         9    0   0   0   0   0   0   0   0   0   0   0
##         10   0   0   0   0   0   0   0   0   0   0   0
##
## Overall Statistics
##
##                Accuracy : 0.6854
##                  95% CI : (0.6706, 0.7)
##     No Information Rate : 0.2675
##     P-Value [Acc > NIR] : < 2.2e-16
##
##                   Kappa : 0.6047
##
##  Mcnemar's Test P-Value : NA
##
## Statistics by Class:
##
##                      Class: 0 Class: 1 Class: 2 Class: 3 Class: 4 Class: 5
## Sensitivity            0.7070 0.057971   0.9247  0.50856   0.8747  0.30769
## Specificity            0.9204 0.994571   0.9446  0.94859   0.8414  0.98538
## Pos Pred Value         0.7643 0.375000   0.8469  0.53747   0.5585  0.42105
## Neg Pred Value         0.8958 0.949469   0.9742  0.94264   0.9670  0.97629
## Prevalence             0.2675 0.053200   0.2490  0.10511   0.1866  0.03341
## Detection Rate         0.1892 0.003084   0.2303  0.05346   0.1632  0.01028
## Detection Prevalence   0.2475 0.008224   0.2719  0.09946   0.2922  0.02442
## Balanced Accuracy      0.8137 0.526271   0.9346  0.72858   0.8580  0.64653
##                      Class: 6 Class: 7 Class: 8 Class: 9 Class: 10
## Sensitivity           0.48951  0.00000  0.00000 0.000000  0.000000
## Specificity           0.97809  1.00000  1.00000 1.000000  1.000000
## Pos Pred Value        0.63927      NaN      NaN      NaN       NaN
## Neg Pred Value        0.96024  0.98792  0.98895 0.998715  0.992804
## Prevalence            0.07350  0.01208  0.01105 0.001285  0.007196
## Detection Rate        0.03598  0.00000  0.00000 0.000000  0.000000
## Detection Prevalence  0.05628  0.00000  0.00000 0.000000  0.000000
## Balanced Accuracy     0.73380  0.50000  0.50000 0.500000  0.500000
```

The performance metrics and classification reports are displayed above, in order of LSTM first and RNN last. Both models failed to predict any class belonging to 7, 8, 9, and 10. There seems to be a class imbalance at the final 4 classes. This is displayed in a table below where classes 7-10 can be seen to be thinly distributed.

```
# Comparison ------------------------------------------------------------
test_summary <- summary(as.factor(complaints$category_id[-idx]))
temp <- complaints$category_id[idx]
valid_summary <- summary(as.factor(temp[vidx]))
train_summary <- summary(as.factor(temp[-vidx]))
pred_lstm_summary <- summary(pred.lstm)
pred_rnn_summary <- summary(pred.rnn)
knitr::kable(data.frame(Training = train_summary, Validation = valid_summary,
                        Testing = test_summary, LSTM = pred_lstm_summary,
                        RNN = pred_rnn_summary))
```

	Training	Validation	Testing	LSTM	RNN
0	8536	1048	1041	1238	963
1	1645	211	207	106	32
2	7047	890	969	930	1058
3	3437	418	409	568	387
4	5995	743	726	659	1137
5	1108	149	130	111	95
6	2451	298	286	279	219
7	352	48	47	0	0
8	321	44	43	0	0
9	49	5	5	0	0
10	187	36	28	0	0

The classification reports reveals the following.

1) Accuracy: The LSTM model achieved a higher overall accuracy than the RNN model with $75\% > 68\%$.
2) 95% CI: The LSTM model achieved a higher and more narrow CI than the RNN model implying it can be expected to consistently achieve higher accuracies.
3) NIR: The NIR refers to the largest class percentage and is the same for both models
4) P-value: Both p-values indicate that the models can be used to model the data.
5) Kappa: Cohen's kappa statistic is higher for the LSTM model which implies it has a higher expected and observed accuracy than the RNN model.
6) Sensitivity/Recall/TPR: The sensitivity displays the ability of the model to predict the classes as belonging to a class if it truly is of the class. Some interesting insights were gained here, where it can be seen that the RNN model slightly outperformed the LSTM model in classes 2 and 4, but severely underperformed in classes 0, 1, 5 and 6.
7) Specificity/TNR: The specificity gives insight into the ability of the model to predict the classes as not being part of a class if it truly is not part of a class. The models performed similarly here. The RNN performed worse than the LSTM in class 4. This lowered specificity as well as higher sensitivity, leads me to believe the RNN model classified a multitude of points as being in class 4. Other than this class the models perform similarly with slight variation between the classes.
8) PPV/Precision: The precision gives insight into the probability that the classes are assigned correctly. Class 2 seems to be the easiest class since both models performed good here, but both struggled substantially with class 1 as well as the latter skewly distributed classes.
9) NPV: The NPV measures the probability that the classes are correctly not assigned to a class. The models performed similarly, with the largest difference coming from class 0. In general the LSTM performed the best here,
10) Prevalence: The prevalence simply indicates what percentage of the testing data belongs to a certain class and is therefore the same across the models. Note the low prevalence metric for classes 7-10 which are all approximately 1% of the testing data.

11) Detection rate: The detection rate is a sort of class-specific accuracy. The LSTM had higher detection rates than the RNN in most of the classes.
12) Detection prevalence: The detection prevalence indicates how often a class was predicted. The combination of the detection prevalence and prevalence reveal that the LSTM over-predicted more class 0 and 3 than there actually were within the dataset, and the RNN predicted more class 4 than there actually were.
13) Balanced Accuracy: The balanced accuracy is a class specific weighted accuracy that disregards skew class distributions. At this point, it is clear to see that the RNN failed to model class 1. The LSTM also struggled but performed better in the majority of the classes.

Considering the aforementioned factors as well as the confusion matrices and, we can see that both models completely failed to classify classes 7, 8, 9 and 10. All four of these classes are skewly distributed, which complicate this problem since LSTM and RNN's are not robust to skew class distributions. The testing and validation instances both contain at the minimum 5 and at the maximum 50 instances of these classes. This amounts to 0.01% of the total dataset which is extremely low. Class 1 was also a hard class, but the LSTM managed to perform better than the RNN in this class. I believe if a larger vocabulary size was used, the results could have been more accurate, but due to the inclusion of the bidirectional layer the models had significant computational cost and I did not have time to experiment with a larger vocabulary value.

Asynchronous Actor-Critic Agent (A3C) and Generative Adversarial Network (GAN) [50]

Question 1) The Asynchronous Advantage Actor-Critic (A3C) algorithm has gained popularity in recent years and has become the go-to Deep RL algorithm for new challenging problems with complex state and action spaces. Provide a high-level overview of:

Question 1.1) What the A3C algorithm is by elaborating on the mechanics behind the algorithm with respect to its name. [5] The Asynchronous Advantage Actor Critic Agent (A3C) is a deep reinforcement learning algorithm which, unlike many deep learning algorithms, learns from a multitude of agents simultaneously. The A3C algorithm employs multiple local agents that each have their own local environments and parameters, and these agents interact with these environments **asynchronously** from other agents, meaning they act independently of other agents. These agents interact with a global network, which defines the global parameters which effects how the agents act. The agents work together to improve the global network's knowledge. Since these agents work so asynchronously, their combined inset provides a measure of diversity to the global network. In contrast to value iteration and policy gradient based approaches, the A3C algorithm predicts the value function and the optimal policy function by using a neural network to perform function approximation. It uses these policy functions and value functions to determine a measure of how **advantageous** a certain action is in a certain state, and this proves insight into how much better the rewards that the agent experiences are than the rewards than the agent expected to receive from taking this action in this state. The **actor-critic** part of the name can be related to the way that the value function and policy function are used within the algorithm. Critic refers to the value function, and the actor refers to the optimal policy network. The value function is updated by the learning agents, and this value function is used to update the policy and therefore the neural network. Due to this tweaking nature, this part of the algorithm resembles an interaction between a critic and actor, where the critic's input alters the actor's actions.

Question 1.2) How the A3C algorithm work. Elaborate on how the algorithm trains the A3C model. [5] First, global shared parameters and local thread-specific parameters are initialized. From a high level, the training process is then as follows.

1) Each agent fetches the global network parameters

2) Each agent interacts with their own environment according to their own local policy from their local thread-specific parameters.
3) Each agent calculates it's current value and policy losses
4) The gradients of the individual agents' value and policy losses can be found according to the current local and global parameters.
5) These individual gradients can be used to update the global network parameters, hence why the global network parameters are said to receive diverse inputs.
6) Repeat steps 1-5

Question 1.3) Contrast the A3C algorithm with the Deep Q-Network (DQN) algorithm. [5] The A3C algorithm employs multiple agents which interact asynchronously with their own respective copies of the environment that all serve to improve the global parameters. The DQN algorithm employs one single agent and environment and therefore only has one set of network parameters. As the A3C algorithm employs multiple agents, there is a linear speed up with the amount of agents that are employed. The DQN algorithm employs one agent and therefore does not experience any speed up. During my research, I found a published article, that stated due to the multiple agents employed the A3C algorithm also has a more stable learning process when compared to the DQN algorithm. Similarly, due to the speed-up with parallel agents, the A3C is considered a more data-efficient algorithm.

Question 2) Develop a GAN that has the ability to learn by itself how to synthesize new images. You are free to use any architecture and hyperparameters when developing the discriminator and adversarial models. Clearly describe your logic and add comments to your code. [25] Produce sample generative outputs of the images when you train the network and discuss your observations. [10]

The GAN development process is displayed below. Some preprocessing was performed, namely the training data was scaled to either -1 or 1, as an hyperbolic tan would be used in the output layer of the generator function, and after normalization the training data was reshaped to flatten the pixels.

MLP GANs typically employ increasing architecture for the generator (since it needs to generate an image) and decreasing architecture for the discriminator (since it needs to classify an image). Therefore the generator was designed with an increasing architecture leading up to an output layer with a hyperbolic tangent function and units corresponding to the dimensions of the flattened pixels. This intermediate parameters were chosen arbitrarily as being similar to that of the example that was done in class.

The discriminator was set up in a similar way, with a decreasing architecture to a single unit layer with a sigmoid activation function. This is done such that the discriminator can classify whether an input image is generated or real. The discriminator uses dropout for added noise with a probability of 0.3. Both these models used leaky ReLu with an alpha value of 0.2 in the intermediate layers to mitigate the effects of sparse gradients. Both models were also compiled with adam optimizers and used binary crossentropy as the loss function. These models were then stacked into the GAN model. The GAN was compiled with the same optimizer and losses as the generator and discriminator. The model setup is displayed below.

```
# Question 3.2) ------------------------------------------------------------
rm(list = ls())
library(keras)
setwd("/Users/jd-vz/Desktop/MEng/Deep Learning/Assignments/PBA3/")
c(c(x_train, y_train), c(x_test, y_test)) %<-% dataset_mnist()
x_train <- (x_train - 127.5) / 127.5 #Use tanh at generator [-1, 1]
x_train <- array_reshape(x_train, c(60000,784)) #Flatten pixels
# Generator ----------------------------------------------------------------
generator <- keras_model_sequential() %>%
  layer_dense(units = 256, input_shape = 100) %>%
  layer_activation_leaky_relu(alpha = 0.2) %>%
```

```
  layer_dense(units = 512) %>%
  layer_activation_leaky_relu(alpha = 0.2) %>%
  layer_dense(units = 1024) %>%
  layer_activation_leaky_relu(alpha = 0.2) %>%
  layer_dense(units = 784, activation = "tanh") #tanh for generation

generator %>% compile(optimizer = "adam", loss = "binary_crossentropy")

# Discriminator ------------------------------------------------------------
discriminator <- keras_model_sequential() %>%
  layer_dense(units = 1024, input_shape = 784) %>%
  layer_activation_leaky_relu(alpha = 0.2) %>%
  layer_dropout(0.3) %>%
  layer_dense(units = 512) %>%
  layer_activation_leaky_relu(alpha = 0.2) %>%
  layer_dropout(0.3) %>%
  layer_dense(units = 256) %>%
  layer_activation_leaky_relu(alpha = 0.2) %>%
  layer_dense(units = 1, activation = "sigmoid") #sigmoid for discrimination

discriminator %>% compile(optimizer = "adam",  loss = "binary_crossentropy")

# Compilation --------------------------------------------------------------
freeze_weights(discriminator)
gan_input <- layer_input(shape = 100)
gan_output <- discriminator(generator(gan_input))
gan <- keras_model(gan_input, gan_output)
gan %>% compile(optimizer = "adam",
                loss = "binary_crossentropy")
```

The training flow can be summarized as follows.

1) Generate noise to initialize the generator.
2) Get real and generated images, and combine the images.
3) Using these combined images, train the discriminator to classify these images as real or generated
4) Freeze the discriminator's weights, such that the training of the GAN does not alter it.
5) Lie to the generator, by training the GAN on noise and claiming that these are real images.

By repeating these steps throughout epochs, the generator and discriminator models are both being trained and improven. Therefore, the generator gets better at generating images that seem like real images, and the discriminator gets better at recognizing images that are generated. The training is displayed below.

```
# Training ----------------------------------------------------------------
epoch <- 20
batch_size <- 64

for(i in 1:epoch)
{
  # 1) Generate noise for generator ----------------------------------------
  noise <- rnorm(batch_size*100) %>% matrix(nrow = batch_size , ncol = 100)
  noise %>% dim()
  # Generate fake images from noise ----------------------------------------
  generated_images <- generator %>% predict(noise)
```

```
  generated_images %>% dim()
  # Get random real images -------------------------------------------------
  real_images <- x_train[sample.int(dim(x_train)[1], size = c(batch_size)),]
  real_images %>% dim()

  #2) Concatenate ---------------------------------------------------------
  comb = rbind(real_images, generated_images)
  comb %>% dim()
  # Generate Labels --------------------------------------------------------
  labels <- rbind(matrix(0, nrow = 2*batch_size, ncol = 1))
  labels[batch_size,1] <- 1 #These are the real images

  #3 + 4) Train discriminator ---------------------------------------------
  discriminator$trainable = T
  discriminator %>% train_on_batch(comb, labels) #Trained on generated and real data
  discriminator$trainable = F

  #5) Train generator -----------------------------------------
  noise <- rnorm(batch_size*100) %>% matrix(nrow = batch_size , ncol = 100)
  y_gen <- matrix(1, nrow = batch_size)
  gan %>% train_on_batch(noise, y_gen)
  # Generated --------------------------------------------------------------
  generated_images <- array_reshape(generated_images, dim =  c(batch_size,28,28))
  generated_images <-generated_images * 127.5 + 127.5
  generated_images <- generated_images/255
  # Real -------------------------------------------------------------------
  real_images <- array_reshape(real_images, dim =  c(batch_size,28,28))
  real_images <-real_images * 127.5 + 127.5
  real_images <- real_images/255
  }
```

The output of the model is displayed below. At this point I confess I made a mistake somewhere in my code, and the generative output did not resemble an image. I could not find the mistake in time. I believe my design and training process is correct, and something minor is disturbing the output.

```
# Plot  ------------------------------------------------------------------
par(mfrow=c(1,2))
plot(as.raster(generated_images[1,,]))
plot(as.raster(real_images[1,,]))
```