Post Block Assessment 2

Francois van Zyl : 18620426

13/07/2020

Stochastic Gradient Descent, Mini-Batch Training [52]

Question 1) What is the order of the main steps for using a gradient descent algorithm? [2]

```
D. 4, 3, 1, 5, 2
```

Question 2) In this question, you will implement stochastic gradient descent and mini-batch training to derive the coefficients of a linear regression model for the provided data set. In addition, you will develop a simple MLP model to compare its performance to that of the linear regression model.

Question 2.i) Use the data sets trainX and trainy to fit the linear regression model. Create all the required functions in order to fit the model based on the training data for both stochastic gradient descent and the mini-batch training algorithm. Clearly show all code and equations required to obtain the coefficients of the linear regression model. Use the mean squared error (mse) as the cost function. You do not need to tune the hyper-parameters (such as learning rate or batch size) to get the perfect model. A few (e.g. 5) iterations of tuning these parameters are sufficient. Plot the cost function versus the number of iterations for the various hyper-parameter settings that are implemented. [30]

```
rm(list = ls())
set.seed(0)
library(keras)
#tensorflow::tf_gpu_configured()
xtrain <- as.matrix(read.csv("/Users/jd-vz/Desktop/MEng/Deep Learning/Assignments/PBA2/Data-20200708/tra
ytrain <- as.matrix(read.csv("/Users/jd-vz/Desktop/MEng/Deep Learning/Assignments/PBA2/Data-20200708/tra
lr <- 0.1 #learning rate
epochs <- 25 #epochs
batch_size <- 256 #Batch size for Mini-Batch Gradient Descent
n <- nrow(xtrain) #Size of training data
graph_y <- 200 #Graph limit for convenient visualization
par(mfrow=c(1,2))

# Functions ----------------------------------------------------------------
cost.function <- function(yhat,x,y) 1/n*sum((y - yhat))^2
deriv.m <- function(yhat,x,y) (-2/n)*sum(x*(y-yhat))
deriv.c <- function(yhat,x,y) (-2/n)*sum((y-yhat))
```

```r
# Stochaistic Gradient Descent ----------------------------------------------
SGDesc <- function(x, y, epochs, lr) {
  cost.history <- yhat <-  c()
  m <- c <- 0
  for(j in 1:epochs)
  {

    for(i in 1:n)
    {
      idx <- sample.int(nrow(xtrain),nrow(xtrain), replace = F)
      x <- x[idx]
      y <- y[idx]
      yhat[i] <- m*x[i]+c
      m <- m - lr*deriv.m(yhat[i],x[i],y[i])
      c <- c - lr*deriv.c(yhat[i],x[i],y[i])
    }
    cost.history[j] <- cost.function(yhat, x, y)
  }
  plot(
    cost.history,
    pch = 19,
    col = "blue",
    xlab = "# Iterations",
    ylab = "Value of cost",
    main = "SGD",
    ylim = c(0,graph_y),
    type = "l"
  )
  data.frame(m, c)
}
```

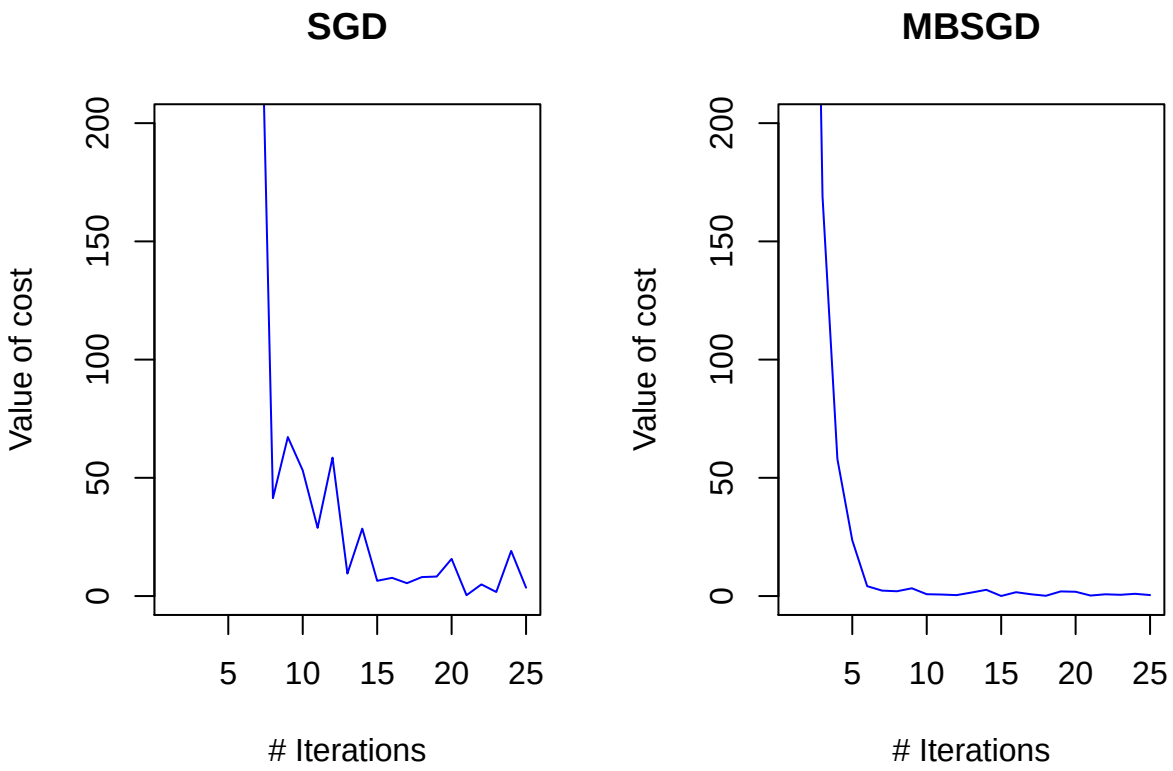
The Stochaistic Gradient Descent algorithm was implemented as seen above. The function loops through a single entry within the dataset and simultaneously updates the change in gradient and bias. After looping through all the entries and epochs, the function returns the final gradient and bias values for the linear regression model.

```r
# Mini-Batch Gradient Descent -----------------------------------------------
MBSGDesc <- function(x, y, epochs, lr, batch_size) {
  yhat <-  cost.history <- c()
  m <- c <- 0
  iter <- ceiling(nrow(y)/batch_size)
  for(j in 1:epochs)
  {
    for(i in 1:iter)
    {
      idx <- sample.int(nrow(x), batch_size, replace = F) #Gather 32 random samples
      x.batch <- x[idx] #Create batches
      y.batch <- y[idx]
      yhat <- m*x.batch+c
      m <- m - lr*deriv.m(yhat,x.batch,y.batch)
      c <- c - lr*deriv.c(yhat,x.batch,y.batch)
    }
    cost.history[j] <- cost.function(yhat, x.batch, y.batch)
  }
```

```
  plot(
    cost.history,
    pch = 19,
    col = "blue",
    xlab = "# Iterations",
    ylab = "Value of cost",
    main = "MBSGD",
    ylim = c(0,graph_y),
    type = "l"
  )
  data.frame(m, c)
}
```

The Mini-Batch Gradient Descent algorithm was implemented as seen above. The function loops through a batch of entries within the dataset and updates the change in gradient and bias per batch. Note that if a batch size of 1 is specified, this algorithm is identical to the stochaistic gradient descent algorith. After looping through all the batches and epochs, the function returns the final gradient and bias values for the linear regression model. The cost function (MSE) is plotted against the specified parameters from the first chunk of code.

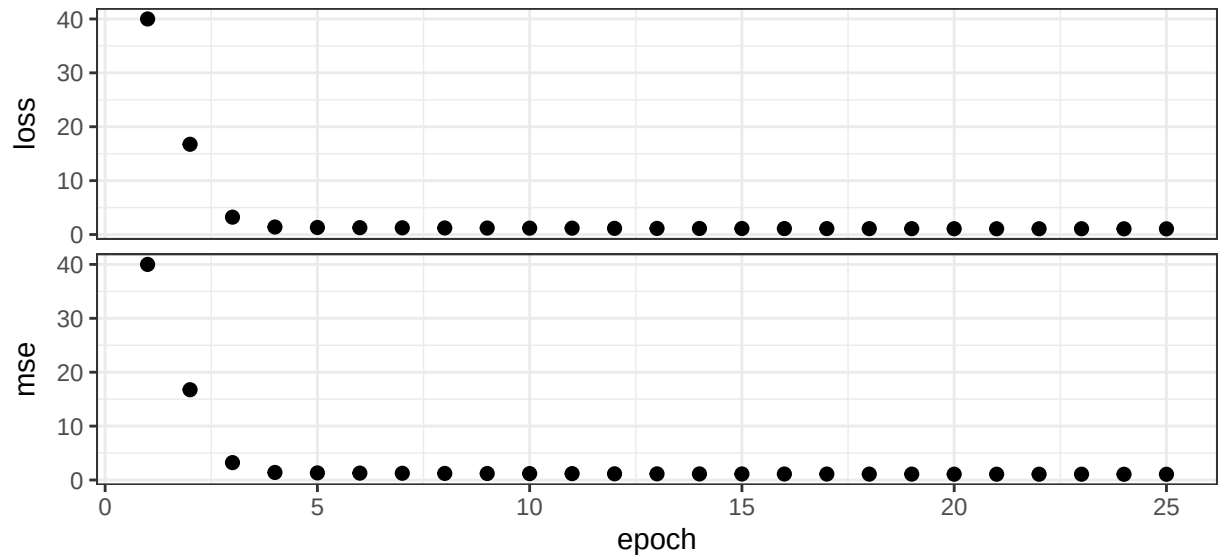
```
par(mfrow=c(1,2))
SGD.const <- SGDesc(xtrain, ytrain, epochs, lr)
MBSGD.const <- MBSGDesc(xtrain, ytrain, epochs, lr, batch_size )
```



As seen in the figure above, the SGD algorithm has a much more noisy approach towards convergence than the MBSGD algorithm. The SGD algorithm also takes longer to converge, where it can be seen that the MBSGD algorithm approaches convergence faster. I expect that the model derived from the MBSGD algorithm will perform better.

Question 2.ii) Using the same training data sets, develop and train a multi-layer perceptron (MLP) model to approximate the input data. Any architecture may be implemented, but reasons must be provided why the parameters were selected. Plot the mean squared error of the training set during training of the network. [10]

```
library(keras)
mlp.model <- keras_model_sequential() %>%
  layer_dense(units = 5, activation = "relu", input_shape = 1) %>%
  layer_dense(units = 1)

mlp.model %>% compile(
  optimizer = "sgd",
  loss = "mse",
  metrics = c("mse")
)

history <- mlp.model %>%
  fit(xtrain, ytrain, epochs=epochs, batch_size = batch_size, verbose = 0)

plot(
  history,
  method = "ggplot2",
  smooth = getOption("keras.plot.history.smooth", F),
  theme_bw = getOption("keras.plot.history.theme_bw", T)
)
```

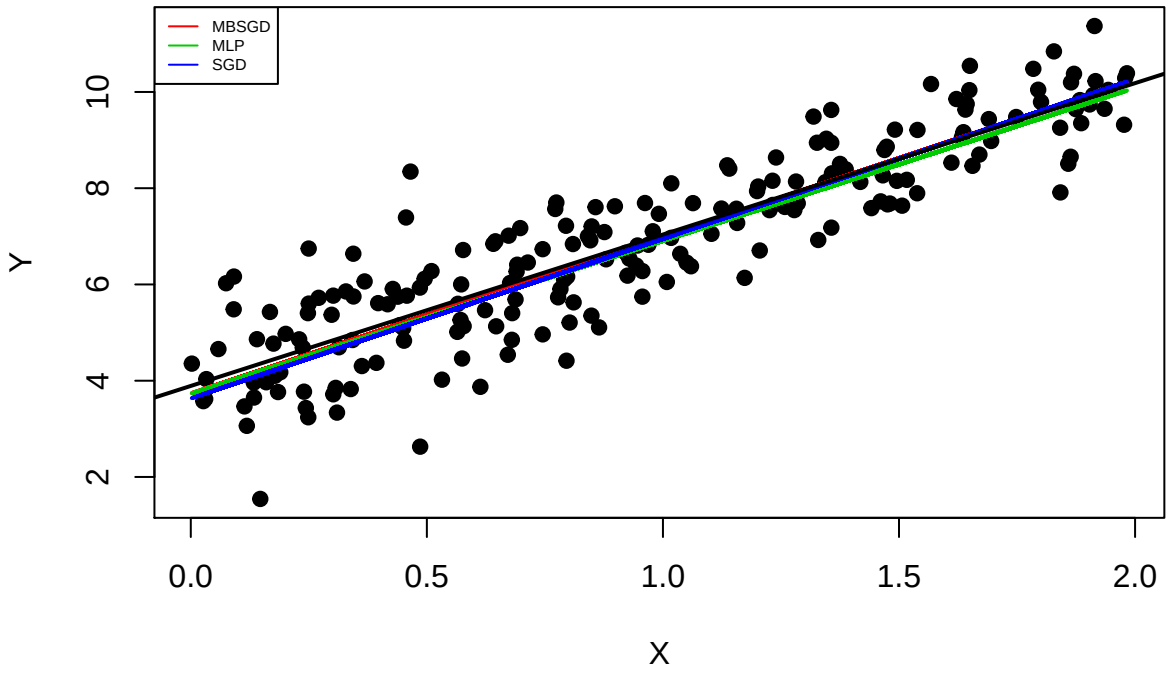


The MLP was implemented as seen above. A rule of thumb for selecting intial architectures was found at this **hyperlink**. I selected a unit input shape with a standard choice of Relu, as well as a single output layer with no activation function since this is a regression problem. The amount of hidden units was chosen

according to a simple rule of thumb: (#inputs + #outputs)$*\frac{2}{3}$ where #inputs = 1 + bias and #outputs = 1. Therefore I chose 5 as the amount of hidden units.

Question 2.iii) Evaluate the linear regression models resulting from the SGD and mini-batch algorithms as well as the MLP model on the test data sets (testX and testy) and compare their performance. Provide insights into the performance of the various models. [10]

```
# Compare the performance of the three models ----------------------------
library(MLmetrics)
testx <- as.matrix(read.csv("/Users/jd-vz/Desktop/MEng/Deep Learning/Assignments/PBA2/Data-20200708/tes
testy <- as.matrix(read.csv("/Users/jd-vz/Desktop/MEng/Deep Learning/Assignments/PBA2/Data-20200708/tes
# SGD model --------------------------------------------------------------
predict.SGD <- SGD.const$m * testx + SGD.const$c
# MBSGD model ------------------------------------------------------------
predict.MBSGD <- MBSGD.const$m * testx + MBSGD.const$c
# MLP model --------------------------------------------------------------
predict.MLP <- mlp.model %>% predict(testx)
# Plot predictions -------------------------------------------------------
par(mfrow=c(1,1))
plot(testy~testx, pch = 19, col = "black", xlab = "X", ylab = "Y")
lines(predict.MBSGD ~ testx, col = 2, lwd = 2, lty = 1)
lines(predict.MLP ~ testx, col = 3, lwd = 2, lty = 1)
lines(predict.SGD ~ testx, col = 4, lwd = 2, lty = 1)
legend("topleft", legend = c("MBSGD", "MLP", "SGD"), lty = 1, col = 2:4, cex = 0.5)
fit.lm <- lm(testy~testx) #Packaged Linear Regression Model
abline(lm(testy~testx), lwd = 2)
```



```
# Evaluate ------------------------------------------------------------------
lm.pred <- predict(fit.lm, data.frame(testx))
knitr::kable(data.frame(MBSGD = MSE(testy, predict.MBSGD), MLP = MSE(testy, predict.MLP),
                        SGD = MSE(testy, predict.SGD),
                        TrueLM = MSE(testy, lm.pred)), format = "markdown")
```

MBSGD	MLP	SGD	TrueLM
0.7967282	0.807744	0.8091908	0.7902405

The testing data plot with the regressor lines for the SGD, MBSGD, and MLP models are displayed above. I included a regression model that is allowed more computational cost to have some sort of *true* output that I can compare the models with. As expected, this packaged linear regression model, *fit.lm*, can be seen to have the lowest MSE in the table above. This packaged model is displayed in the plot as the black line through the middle.

The MBSGD and MLP models typically perform very similar to fit.lm's regressor line, and there is no clear winner between these two since the results of the MSE can and do change marginally upon execution of the code. The MLP model is displayed in green in the figure above, and the MBSGD model is displayed in red. Both models regression lines are very similar to the packaged linear regression model's regression line. The SGD model consistently performs worse than the other models, which I believe is to be expected since MBSGD has faster convergence and a less noisy trajectory towards convergence. This is due to the MBSGD algorithm smoothing out the noise due to having the specifed batch size which results in a sort of averaging of points. From class, we can recall that the converse is true for the SGD model as it is more harshly affected by noise and outliers within the dataset due to a singular batch size. Note that even if the MLP model was specified as having a SGD optimizer, I specified a batch size, which effectively made the model use MBSGD. This batch size was chosen arbitrarily and applied to both the MBSGD and MLP models.

Autoencoders and CNN [88]

Question 1) In this question, we will be using the popular MNIST Dataset. The MNIST database (Modified National Institute of Standards and Technology database) is a large database of handwritten digits that is commonly used for training various image processing systems. the black and white images from NIST were normalized to fit into a 28x28 pixel bounding box and anti-aliased, which introduced grayscale levels. Various autoencoders models will be developed and fitted to this data set to compare their performance and characteristics.
Question 1.i) Import the MNIST dataset into your environment (packages for all main programming examples exist to download and import the data set). Use 60000 images for training and 10000 images for testing.

```
# Question 2: Autoencoders and CNN ------------------------------------
rm(list = ls())
set.seed(0)
library(keras)
# Import Mnist --------------------------------------------------------
c(c(train_data, train_labels), c(test_data, test_labels)) %<-% dataset_mnist()
# Normalize data ------------------------------------------------------
train_data <- train_data / 255
test_data <- test_data / 255 # [0, 1]
# Input shape ---------------------------------------------------------
inp_size <- dim(train_data)[2]*dim(train_data)[3] #Image dimension
dim_train <- c(dim(train_data)[1], inp_size) #60000, (28*28)
```

```
dim_test <- c(dim(test_data)[1], inp_size) #10000, (28*28)
train_data <- array_reshape(x = train_data, dim = dim_train)
test_data <- array_reshape(x = test_data, dim = dim_test)#Data reshaped to [entries, img size]
hidden_size <- 32 #code_size
epochs <- 5 #5 epochs applied to all autoencoders
# Function to plot ----------------------------------------------------------
plot.images <- function(decoded.images, test.images, index)
{
  decoded.images <- scales::squish(decoded.images, c(0,1))
  test.images <- scales::squish(test.images, c(0,1))
  predicted.images <- array_reshape(decoded.images, dim = c(dim(decoded.images)[1], 28, 28))
  actual <- array_reshape(test.images, dim = c(dim(test.images)[1], 28, 28))

  par(mfrow = c(5,2), mar = c(1,0,0,0))
  for (i in 8:12)
  {
    plot(as.raster(predicted.images[i,,]))
    plot(as.raster(actual[i,,]))
  }
}
knitr::kable(data.frame(train = dim(train_data), test = dim(test_data)), format = "markdown")
```

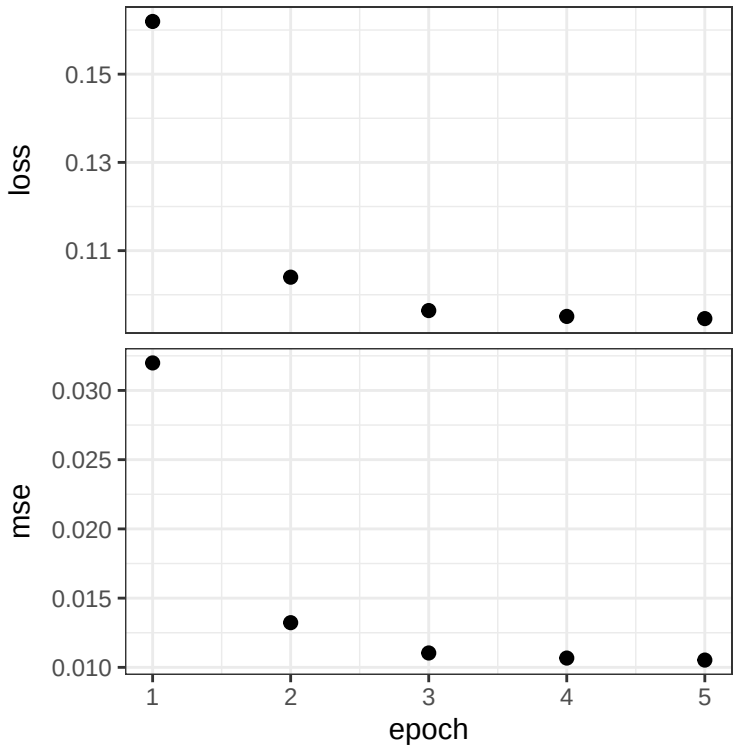
train	test
60000	10000
784	784

Question 1.ii) Develop the following autoencoders: For each of these models, following the instructions in iii)
Undercomplete autoencoder

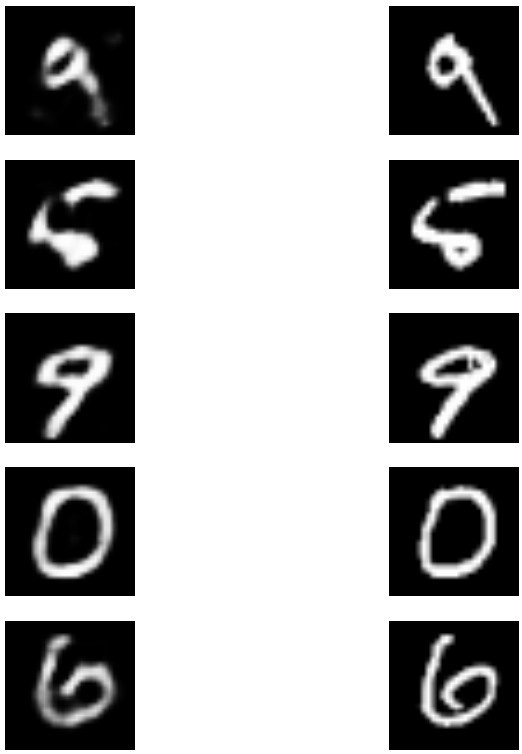
```
# Develop model -------------------------------------------------------------
auto.under <- keras_model_sequential() %>%
  layer_dense(units = hidden_size, activation = "relu", input_shape = inp_size) %>%
  layer_dense(units = inp_size, activation = "sigmoid")
auto.under %>% compile(optimizer="adam", loss="binary_crossentropy", metric = "mse")
# Train/Test model ----------------------------------------------------------
history.under <- auto.under %>% fit(train_data,train_data, epochs=epochs, verbose = 0)
decoded.images <- auto.under %>% predict(test_data)
MSE.under <- MLmetrics::MSE(decoded.images, test_data)
MSE.under
```

```
## [1] 0.01013796
```

```
plot(
  history.under,
  method = "ggplot2",
  smooth = getOption("keras.plot.history.smooth", F),
  theme_bw = getOption("keras.plot.history.theme_bw", T)
)
```




```
# Plot 5 Images ------------------------------------------------------------
plot.images(decoded.images, test_data)
```

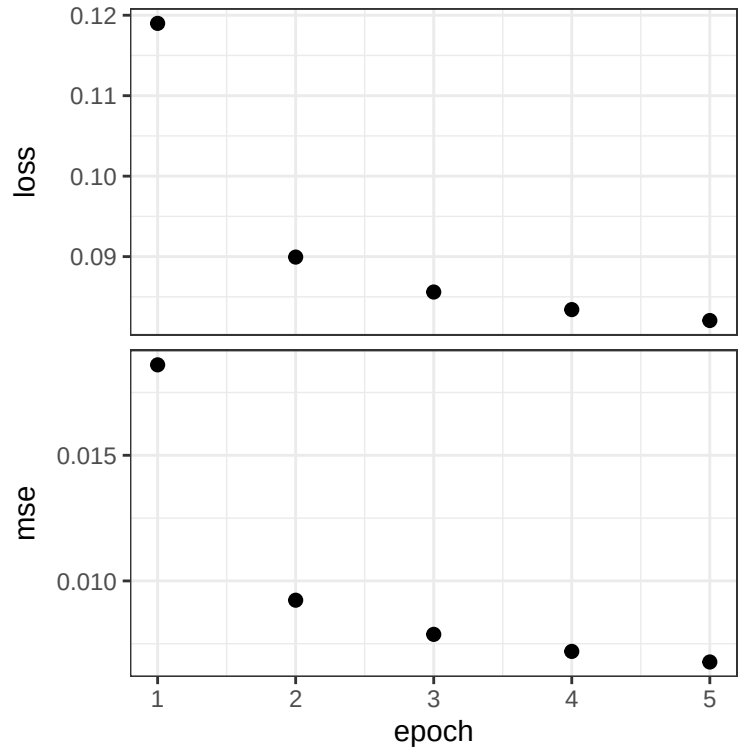


Deep/Stacked Autoencoder

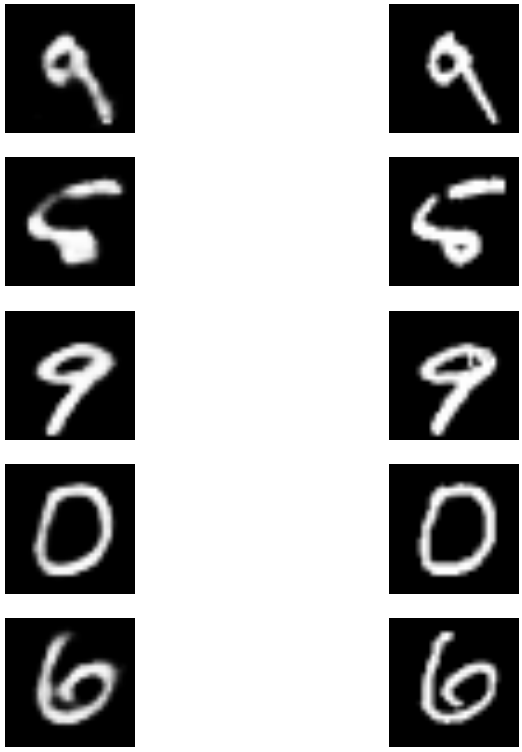
```
# Develop model ----------------------------------------------------------
auto.stack <- keras_model_sequential() %>%
  layer_dense(units = inp_size/2, activation = "relu", input_shape = inp_size) %>%
  layer_dense(units = hidden_size, activation = "relu") %>%
  layer_dense(units = inp_size/2, activation = "relu") %>%
  layer_dense(units = inp_size, activation = "sigmoid")

auto.stack %>% compile(optimizer="adam", loss="binary_crossentropy", metric = "mse")

# Train/Test model -------------------------------------------------------
history.stack <- auto.stack %>% fit(train_data,train_data, epochs=epochs, verbose = 0)
decoded.images <- auto.stack %>% predict(test_data)
MSE.stack <- MLmetrics::MSE(decoded.images, test_data)
MSE.stack
```

```
## [1] 0.006642735
```

```
plot(
  history.stack,
  method = "ggplot2",
  smooth = getOption("keras.plot.history.smooth", F),
  theme_bw = getOption("keras.plot.history.theme_bw", T)
)
```



```
# Plot 5 Images ------------------------------------------------------------
plot.images(decoded.images, test_data)
```

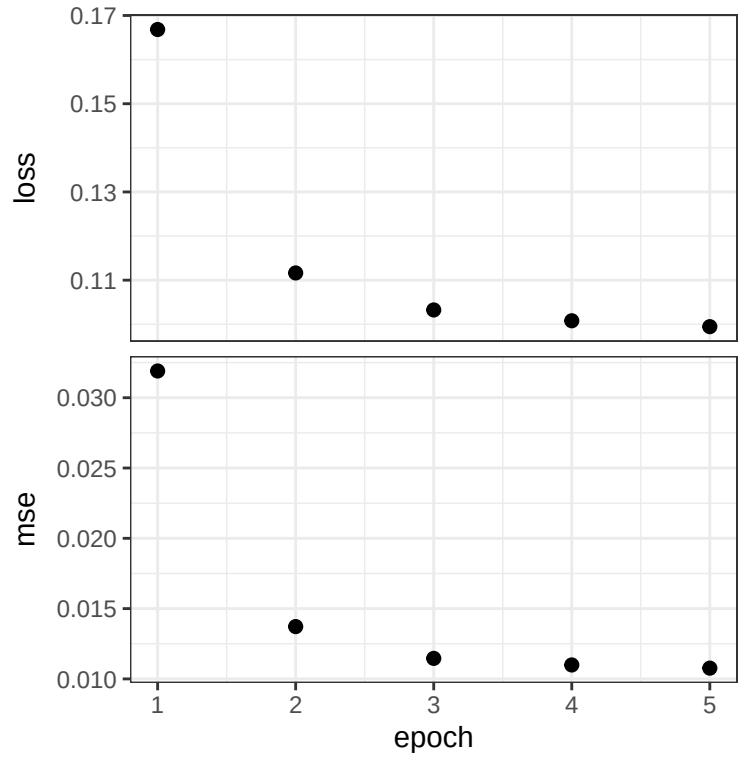


Sparse Autoencoder

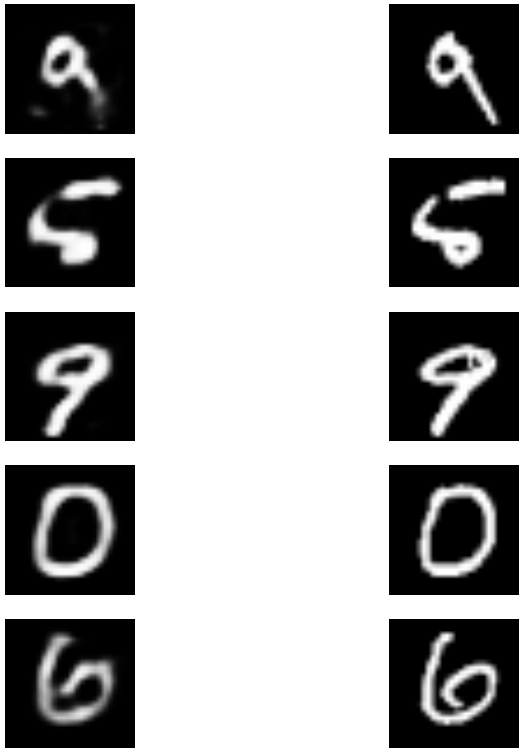
```
# Develop model ----------------------------------------------------------
auto.sparse <- keras_model_sequential() %>%
  layer_dense(units = hidden_size, activation = "relu", input_shape = inp_size,
              activity_regularizer= regularizer_l2(l = 0.00001)) %>%
  layer_dense(units = inp_size, activation = "sigmoid")

auto.sparse %>% compile(optimizer="adam", loss="binary_crossentropy", metric = "mse")
# Train/Test model -------------------------------------------------------
history.sparse <- auto.sparse %>% fit(train_data,train_data, epochs=epochs, verbose = 0)
decoded.images <- auto.sparse %>% predict(test_data)
MSE.sparse <- MLmetrics::MSE(decoded.images, test_data)
MSE.sparse
```

```
## [1] 0.01028185
```

```
plot(
  history.sparse,
  method = "ggplot2",
  smooth = getOption("keras.plot.history.smooth", F),
  theme_bw = getOption("keras.plot.history.theme_bw", T)
)
```




```
# Plot 5 Images ----------------------------------------------------------
plot.images(decoded.images, test_data)
```

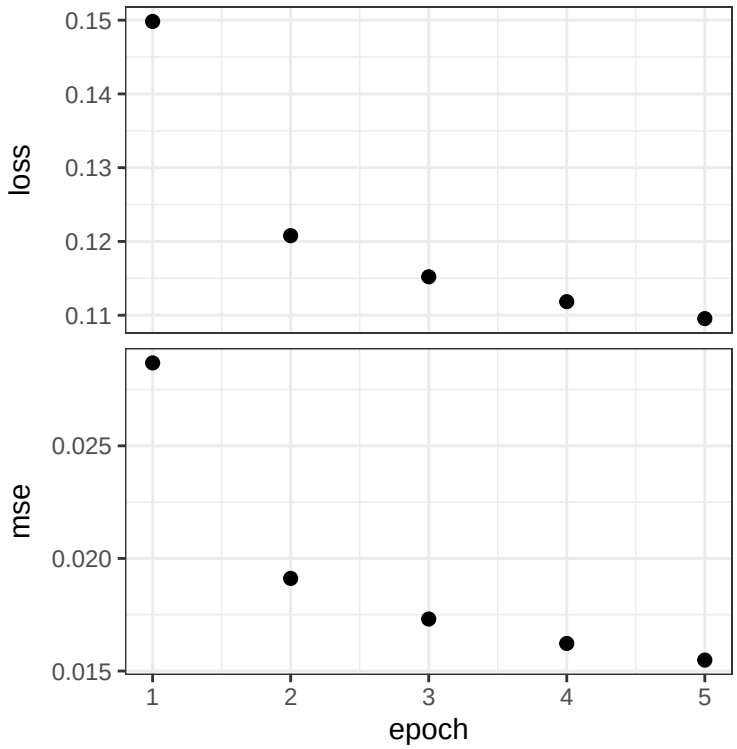


Denoising Autoencoder

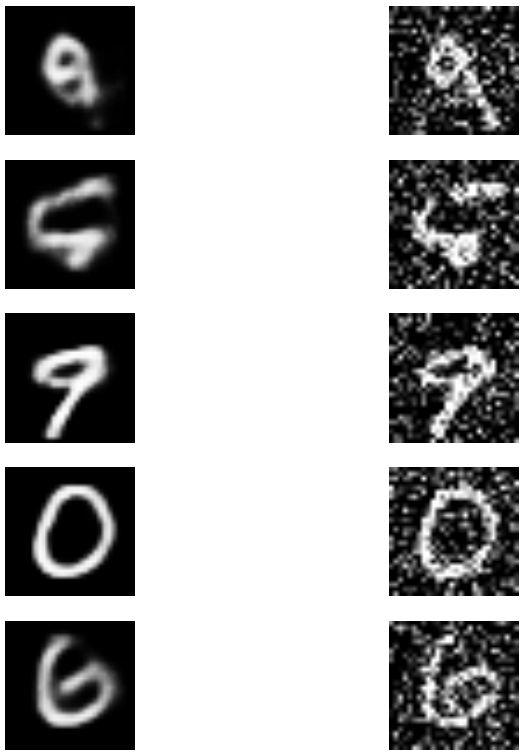
```
# Create some noisy data ------------------------------------------------
noise_factor <- 0.4
train_noise <- noise_factor*matrix(rnorm(n = dim(train_data)[1]*dim(train_data)[2]), ncol = dim(train_da
test_noise<- noise_factor*matrix(rnorm(n = dim(test_data)[1]*dim(test_data)[2]), ncol = dim(test_data)[1
knitr::kable(data.frame(dim(train_data), dim(train_noise), dim(test_data), dim(test_noise)))
```

dim.train_data.	dim.train_noise.	dim.test_data.	dim.test_noise.
60000	60000	10000	10000
784	784	784	784

```
train_data_noisy <- train_data + train_noise
test_data_noisy <- test_data + test_noise
train_data_noisy <- scales::squish(train_data_noisy, c(0,1))
test_data_noisy <- scales::squish(test_data_noisy, c(0,1))
# Develop model ---------------------------------------------------------
auto.noise <- keras_model_sequential() %>%
  layer_dense(units = inp_size/2, activation = "relu", input_shape = inp_size) %>%
  layer_dense(units = hidden_size, activation = "relu") %>%
  layer_dense(units = inp_size/2, activation = "relu") %>%
  layer_dense(units = inp_size, activation = "sigmoid")

auto.noise %>% compile(optimizer="adam", loss="binary_crossentropy", metric = "mse")
# Train/Test model ------------------------------------------------------
history <- auto.noise %>% fit(train_data_noisy, train_data, epochs=epochs, verbose = 0)
decoded.images <- auto.noise %>% predict(test_data_noisy)
MSE.noise <- MLmetrics::MSE(decoded.images, test_data)

plot(
  history,
  method = "ggplot2",
  smooth = getOption("keras.plot.history.smooth", F),
  theme_bw = getOption("keras.plot.history.theme_bw", T)
)
```




```
# Plot 5 Images ------------------------------------------------------------
plot.images(decoded.images, test_data_noisy)
```



Convolutional Autoencoder

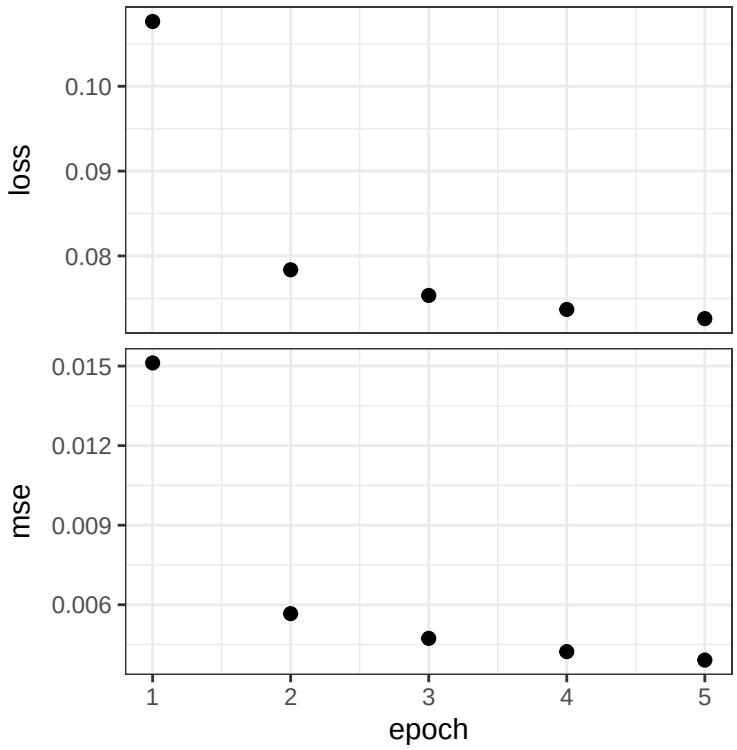
```
# Need 4 Dimensions ------------------------------------------------------
# Import Mnist -----------------------------------------------------------
c(c(train_data, train_labels), c(test_data, test_labels)) %<-% dataset_mnist()
# Normalize data ---------------------------------------------------------
train_data <- train_data / 255
test_data <- test_data / 255 # [0, 1]
# Input shape ------------------------------------------------------------
dim(train_data)
```

```
## [1] 60000    28    28
```

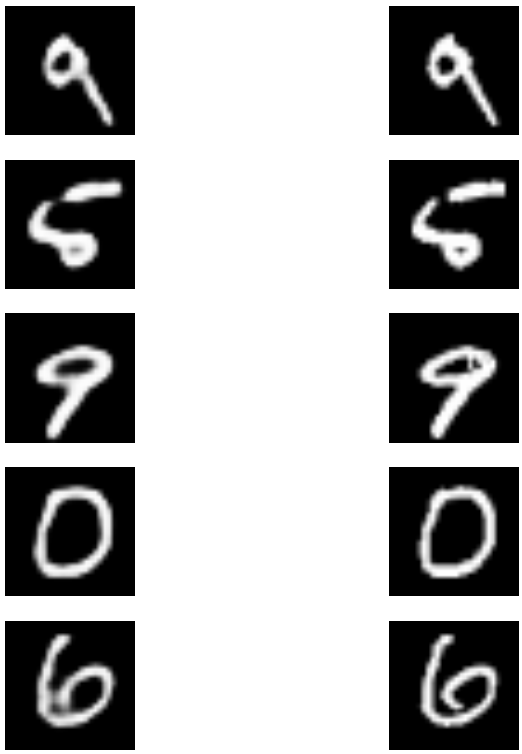
```
inp_size <- c(28, 28, 1)
dim_train <- c(dim(train_data)[1], inp_size) #60000, 28.28.1
dim_test <- c(dim(test_data)[1], inp_size) #10000, 28.28.1
train_data <- array_reshape(x = train_data, dim = dim_train)
test_data <- array_reshape(x = test_data, dim = dim_test)#Data reshaped to [entries, img size]
# Develop model ----------------------------------------------------------
encoder_input = layer_input(shape = inp_size)

decoder_output = encoder_input %>%
  layer_conv_2d(16, kernel_size=c(3,3), activation="relu", padding="same") %>%
  layer_max_pooling_2d(c(2,2))  %>%
  layer_conv_2d(8, kernel_size=c(3,3), activation="relu", padding="same") %>%
  layer_max_pooling_2d(c(2,2), padding="same")  %>%
  layer_conv_2d(8, kernel_size= c(3,3), activation="relu", padding="same") %>%
  layer_upsampling_2d(c(2,2)) %>%
  layer_conv_2d(16, kernel_size = c(3,3), activation="relu", padding="same") %>%
  layer_upsampling_2d(c(2,2)) %>%
  layer_conv_2d(1, kernel_size=c(3,3), activation="sigmoid", padding="same")

auto.conv = keras_model(encoder_input, decoder_output)
auto.conv %>% compile(optimizer="adam", loss="binary_crossentropy", metric = "mse")
# Train/Test model -------------------------------------------------------
history.conv <- auto.conv %>% fit(train_data, train_data, epochs=epochs, verbose = 0)
decoded.images <- auto.conv %>% predict(test_data)
MSE.conv <- MLmetrics::MSE(decoded.images, test_data)
MSE.conv
```

```
## [1] 0.003666358
```

```
plot(
  history.conv,
  method = "ggplot2",
  smooth = getOption("keras.plot.history.smooth", F),
  theme_bw = getOption("keras.plot.history.theme_bw", T)
)
```




```
# Plot 5 Images ------------------------------------------------------------
plot.images(decoded.images, test_data)
```



Training Set up

All the models were trained using the adam optimizer, binary cross-entropy as the loss function, MSE as an additional evaluation metric, for a total of 20 epochs. The sigmoid activation function was selected as the output activation function to keep the output values within [0, 1]. The relevant loss and evaluation training plots are displayed, and at the end of training, the reconstructed images are displayed on the left hand side, and the original image on the right hand side.

Undercomplete autoencoder
The undercomplete autoencoder was set up with a hidden unit size of 32 and input and output size of 28 * 28 = 784. No fine tuning was performed, the only explicit requirement that was enforced was the fact that the hidden layer is required to be smaller than the input layer. This model therefore contained 3 layers with 784,32, and 784 units within the respective layers. This model implements relu activation functions in the intermediate layers as well as a sigmoid activation function at the output layer.

Stacked/deep autoencoder
The deep autoencoder was set up with 5 layers. The input and output size was the same as the previous autoencoder, and the first and third hidden layers had half the input layer's size; I chose this value arbitrarily. The middle hidden layer was kept the same size as before. Once again, this model implements relu activation functions in the intermediate layers and a sigmoid activation function at the output layer.

Sparse autoencoder
The sparse autoencoder was set up identically to the undercomplete autoencoder, with the exception that the activation function of the hidden layer now has a L2 regularizer with a regularization factor of 0.00001. I believe this model could have performed better if more layers are used, but I did not implement this.

Denoising autoencoder
This denoising autoencoder was set up by adding gaussian noise to the training data. Other than this change in training data, I implemented the model with similar architecture to the deep autoencoder.

Convolutional autoencoder
The implementation of the convolutional autoencoder differs the most from the other autoencoders. The data was reshaped to be 4 dimensional for the input dimension, and was applied with output filters in the order 16, 8, 8, 16, and 1. The convolution window's width and height were both kept constant at 3, the layers were reduced by a pooling function, and eventually reconstructed by applying upsampling at the latter layers. Relu functions were applied at the intermediate layers, and a sigmoid function was applied at the final layer.

Testing Results

```
knitr::kable(data.frame(Undercomplete = MSE.under, Stacked = MSE.stack, Sparse = MSE.sparse,
                        Denoise = MSE.noise, Conv = MSE.conv))
```

Undercomplete	Stacked	Sparse	Denoise	Conv
0.010138	0.0066427	0.0102819	0.0154746	0.0036664

The MSE for the various models are displayed above. It can be noted that the undercomplete autoencoder performs the worst of all the models with the highest testing MSE. This is to be expected, since an undercomplete autoencoder uses a constrained amount of nodes within the hidden layer and is generally used to identify the most important or latent attributes within the dataset. The stacked model performs better than the undercomplete autoencoder, which is also expected due to it having a more expansive architecture.

The sparse autoencoder performed similarly to the undercomplete architecture, and this is due to the sparse model having a similar architecture to the undercomplete architecture. In fact the only difference stems from the fact that the sparse autoencoder's hidden layer activation function is regularized with L2 regularization with the specified regularization factor. In my research for this assignment, I discovered that there exists variants to this autoencoder that implement more hidden layers, but I decided to stick with the vanilla definition of a sparse autoencoder. This ensures that the sparse autoencoder uses less neurons than it's total capacity. The denoising model performed the worst of the models, but it still performed remarkably well for a model that was trained and tested on noise. The convolutional model performed the best out of all the models, which implements convolution to encode and decode the inputs. This model has a higher computational budget than the other models, learns differently than the other models, and it was expected to perform better than the other problems.

Question 2) In this question we will use the Stanford dogs dataset. You will develop a convolutional autoencoder to fit to the data set. The Stanford Dogs Dataset has over 20k images categorized into 120 breeds with uniform bounding boxes. I have decreased the number of images to use to minimize cpu resources required during training. I. Download the dogs folder which contains all the 64x64 pixel cropped images of the dogs from SunLearn. Load the images into your environment. (hint: you may want to use the ImageDataGenerator and flow_from_directory functionality from Keras). II. Develop a convolutional autoencoder to fit the data. Clearly illustrate your process of determining the best model (within reasonable time limits). Discuss why the chosen model was selected among those developed. [15]

```
# Question 3: Convolutional Autoencoder: Dogs -------------------------------
# Read in, set dir, and preprocess data ------------------------------------------
rm(list = ls())
set.seed(0)
library(keras)
setwd(dir = "/Users/jd-vz/Desktop/MEng/Deep Learning/Assignments/PBA2")
train.dir <- "C:/Users/jd-vz/Desktop/MEng/Deep Learning/Assignments/PBA2/Train/"
valid.dir <- "C:/Users/jd-vz/Desktop/MEng/Deep Learning/Assignments/PBA2/Validation/"
# Ensure [0, 1] ------------------------------------------------------------
train_data_gen <- image_data_generator(rescale = 1/255)
valid_data_gen <- image_data_generator(rescale = 1/255)
# Set up generator ---------------------------------------------------------------
train_size <- 300
valid_size <- 100
batch_size <- 10
train.gen <- flow_images_from_directory(directory = train.dir, generator = train_data_gen,
                                        target_size = c(64,64),batch_size = batch_size)
valid.gen <- flow_images_from_directory(directory = valid.dir, generator = valid_data_gen,
                                        target_size = c(64,64),batch_size = batch_size)
# Some global variables ----------------------------------------------------------
image.res <- c(64,64)
rgb.channel <- 3
inp_size <- c(image.res, rgb.channel) #64x64 pixels, with 3 for RGB
patience.factor <- 3 #Patience factor for all models
```

```
# Construct model 1-----------------------------------------------------------
auto.dogs <-  keras_model_sequential() %>%
  layer_conv_2d(filters = 32, kernel_size = c(3, 3), activation = "relu",
                input_shape = inp_size) %>%
  layer_batch_normalization() %>%
  layer_max_pooling_2d(pool_size = c(2, 2)) %>%
  layer_conv_2d(filters = 64, kernel_size = c(3, 3), activation = "relu") %>%
  layer_batch_normalization() %>%
  layer_max_pooling_2d(pool_size = c(2, 2)) %>%
  layer_conv_2d(filters = 128, kernel_size = c(3, 3), activation = "relu") %>%
  layer_batch_normalization() %>%
  layer_max_pooling_2d(pool_size = c(2, 2)) %>%
  layer_flatten() %>%
  layer_dense(units = 512, activation = "relu") %>%
  layer_batch_normalization() %>%
  layer_dense(units = 1, activation = "sigmoid")

auto.dogs %>% compile(optimizer=optimizer_rmsprop(lr = 1e-6), loss="binary_crossentropy",
                      metric = "mse")

# Train model 1---------------------------------------------------------------
history.init <- auto.dogs %>% fit_generator(
  generator = train.gen,
  epochs = 30,
  steps_per_epoch = train_size/batch_size,
  validation_data =valid.gen,
  validation_steps = valid_size/batch_size,
  callbacks = callback_early_stopping(patience = patience.factor)
)

evaluate.init <- auto.dogs %>% evaluate_generator( generator = valid.gen,
                                                   steps = valid_size/batch_size)
```

```
 #Construct model 2 ----------------------------------------------------------
# This model has dropout
auto.sec <-  keras_model_sequential() %>%
  layer_conv_2d(filters = 32, kernel_size = c(3, 3), activation = "relu",
                input_shape = inp_size) %>%
  layer_batch_normalization() %>%
  layer_max_pooling_2d(pool_size = c(2, 2)) %>%
  layer_dropout(rate = 0.3) %>%
  layer_conv_2d(filters = 64, kernel_size = c(3, 3), activation = "relu") %>%
  layer_batch_normalization() %>%
  layer_max_pooling_2d(pool_size = c(2, 2)) %>%
  layer_dropout(rate = 0.3) %>%
  layer_conv_2d(filters = 128, kernel_size = c(3, 3), activation = "relu") %>%
  layer_batch_normalization() %>%
  layer_max_pooling_2d(pool_size = c(2, 2)) %>%
  layer_dropout(rate = 0.3) %>%
  layer_flatten() %>%
  layer_dense(units = 512, activation = "relu") %>%
  layer_batch_normalization() %>%
  layer_dropout(rate = 0.3) %>%
```

```
  layer_dense(units = 1, activation = "sigmoid")

auto.sec %>% compile(optimizer=optimizer_rmsprop(lr = 1e-6), loss="binary_crossentropy",
                     metric = "mse")

# Train and evaluate model 2 ------------------------------------------------
history.sec <- auto.sec %>% fit_generator(
  generator = train.gen,
  epochs = 30,
  steps_per_epoch = train_size/batch_size,
  validation_data =valid.gen,
  validation_steps = valid_size/batch_size,
  callbacks = callback_early_stopping(patience = patience.factor)
)

evaluate.sec <- auto.sec %>% evaluate_generator(generator = valid.gen,
                                                steps = valid_size/batch_size)
```

```
# Construct model 3 ------------------------------------------------------
# Augment training data
train_data_gen <- image_data_generator(rescale = 1/255, rotation_range = 30,
                                       width_shift_range = 0.1,
                                       height_shift_range = 0.1,
                                       zoom_range = 0.3, horizontal_flip = T)
valid_data_gen <- image_data_generator(rescale = 1/255)
train.gen <- flow_images_from_directory(directory = train.dir, generator = train_data_gen,
                                        target_size = c(64,64),
                                        batch_size = batch_size)
valid.gen <- flow_images_from_directory(directory = valid.dir, generator = valid_data_gen,
                                        target_size = c(64,64),
                                        batch_size = batch_size)

auto.three <-  keras_model_sequential() %>%
  layer_conv_2d(filters = 32, kernel_size = c(3, 3), activation = "relu",
                input_shape = inp_size) %>%
  layer_batch_normalization() %>%
  layer_max_pooling_2d(pool_size = c(2, 2)) %>%
  layer_dropout(rate = 0.3) %>%
  layer_conv_2d(filters = 64, kernel_size = c(3, 3), activation = "relu") %>%
  layer_batch_normalization() %>%
  layer_max_pooling_2d(pool_size = c(2, 2)) %>%
  layer_dropout(rate = 0.3) %>%
  layer_conv_2d(filters = 128, kernel_size = c(3, 3), activation = "relu") %>%
  layer_batch_normalization() %>%
  layer_max_pooling_2d(pool_size = c(2, 2)) %>%
  layer_dropout(rate = 0.3) %>%
  layer_flatten() %>%
  layer_dense(units = 512, activation = "relu") %>%
  layer_batch_normalization() %>%
  layer_dropout(rate = 0.3) %>%
  layer_dense(units = 1, activation = "sigmoid")

auto.three %>% compile(optimizer=optimizer_rmsprop(lr = 1e-6), loss="binary_crossentropy",
```

```
                        metric = "mse")

# Train and evaluate model 3 ------------------------------------------------
history.three <- auto.three %>% fit_generator(
  generator = train.gen,
  epochs = 30,
  steps_per_epoch = train_size/batch_size,
  validation_data =valid.gen,
  validation_steps = valid_size/batch_size,
  callbacks = callback_early_stopping(patience = patience.factor)
)

evaluate.three <- auto.three %>% evaluate_generator(generator = valid.gen,
                                                    steps = valid_size/batch_size)
```

```
knitr::kable(data.frame(First = evaluate.init, Second = evaluate.sec, Final = evaluate.three))
```

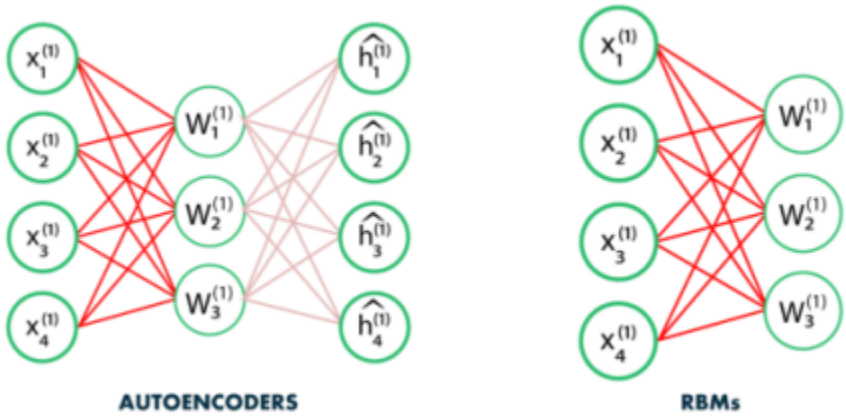
	First	Second	Final
loss	0.6583835	1.0526155	0.3150113
mse	0.2353789	0.4234309	0.0774532

I approached this problem by arbitrarily choosing a autoencoder that has sufficient capacity to learn the data. The initial model for the autoencoder included 32,64,128 and 512 filters respectively that led to a single unit output with a sigmoid activation function. All intermediate layers were chosen to have relu activation functions. Each model was trained for 30 epochs, and early stopping with a patience factor of 3 was applied to all the models. Batch normalization was also applied to each model. After constructing the first model, I added dropout with a rate of 0.3 to the model as a potential regularization method. The results seem to change with execution, and in most of the cases this seemed to aid the performance of the model. As a final method to improve the performance of this model, I added data augmentation to the training set. As expected, this consistently improved the performance of the model. The three different models loss and MSE are displayed above.

Restricted Boltzmann Module [20]

Question 1) Provide a detailed discussion of what Restricted Boltzmann Module/Machines (RBM) are. Specifically, refer to the following. a. Structure/architecture of an RBM. Illustrate this visually as part of your discussion. [5] b. Differences and similarities between autoencoders and RBMs. [5] c. A high-level overview of the contrastive divergence (CD) procedure. No detailed equations are required here. [5] d. Discuss 5 use cases for RBM and why it (RBMs) is better/worse than autoencoders for these cases. [5]

RBM's are a specific type of Boltzmann machines, which have the unique restriction that there is a limitation placed upon the connections between the visible and hidden layers. The RBM is structured as a symmetric bipartite graph, such that no two units within a layer are allowed to be connected to each other. [Note: This structure restriction enables for efficient training through the contrastive divergence procedure]. The first difference that can be noted between autoencoders and RBM's, is that RBM's are two-layer stochastic neural networks, and therefore the neurons will act stochastically, or there behaviour is randomly determined. Another difference between autoencoders and RBM's is the facts that autoencoders attempt to find the most optimal encoding-decoding process for a set of inputs, where RBM's attempt to learn the probability distribution and relationships of the inputs received from the visible layer. RBM's also have hidden and visible bias units, which differentiates it further from autoencoders. These hidden bias units produce the RBM's activation on the forward pass, and the backwards bias aids in reconstruction through the backward pass. The structure for an RBM and autoencoder is displayed below.



As mentioned previously, the RBM attempts to learn probability distributions. Recall that RBM networks are energy-based networks, and therefore the energy can be calculated for the combination of the visible and hidden units, which in turn allows for the calculation of the probability that the RBM assigns to a vector received on the visible layer. This probability of the vector can then be used to calculate and move in the direction of the maximum log-likelihood through gradient-ascent based approaches [maximize the probability of a vector belonging to a distribution]. The reconstruction error is therefore determined by the difference between the conditional probability of a hidden layer output occurring given a visible layer input, and the conditional probability of a visible layer input occurring given a hidden layer output, with respect to the same weights. This difference is known as the Kullback–Leibler divergence, and the goal of the RBM's learning process is to minimize this difference such that the hidden and visible probability distributions are as similar as possible. However, this sampling process is complicated and suffers from biased samples. To resolve this problem of biased samples and get a suitable approximation of the gradient, the k-step CD procedure is implemented. This procedure speeds up the sampling process by using Gibbs sampling and initializing the markov chain from a distribution that it is expected to be close to its' true distribution to speed up convergence. This allows for a faster approximation of the gradient of the energy function, and therefore faster and more efficient maximization of the probability of a vector belonging to a distribution. This method therefore leads to a fast and farely accurate estimation of the maximum gradient of the log-likelihood without a full Monte Carlo sample.

RBM's have been applied to movie recommender systems, collaborative filtering, dimensionality reduction,

classification, regression, feature learning, topic modelling. I found the following examples online.

As seen **here**, both RBM's and autoencoders have been applied to speech recognition and the auto-encoder outperformed the RBM. I believe this is due to the superior performance of autoencoders to compress data.

RBM's and autoencoders' performance have also been compared on image denoising, as found **here**, and it was found at high levels of noise the RBM outperformed the auto-encoder. I believe this is due to the ability of RBMs to learn probability distributions; and their superior generative capabilities to that of autoencoders.

RBMs and autoencoders can also be combined, as found **here**, stacked autoencoders with and without RBM's were applied to facial recognition and the model without the RBM had a higher reconstruction error than the models with the RBM. I believe this displays the disparity that not only one model might necessarily always be the best, but combinations of models should be investigated if the situation calls for it.

As seen **here**, RBMs and autoencoders have both been applied to efficient image search and data compression, and autoencoders have outperformed the RBM's due to their compression capabilities.

Finally, from the class material, RBMs have outperformed autoencoders and have received a large amount of public attention due to a Netflix recommender system that was implemented. This is due the RBM having a high learning capability for probability distributions, making it excellent for recommender systems.