Code Appendix: A Dive into Autoencoders

Elvira Fleury and Rachel Slimovitch

5/8/2023

```
knitr::opts_chunk$set(echo = TRUE)
library(tensorflow)
library(keras)
library(ggplot2)
library(caret)
library(imager)
library(abind)
library(knitr)
```

Fitting a traditional autoencoder

```
knitr::include_graphics("intro_image.png")
```

Reading in data and pre-processing

```
knitr::include_graphics("data_processing_final.png")


load("~/Desktop/statistical learning and big data/final project data/A.RData")
load("~/Desktop/statistical learning and big data/final project data/B.RData")
load("~/Desktop/statistical learning and big data/final project data/C.RData")
load("~/Desktop/statistical learning and big data/final project data/D.RData")
load("~/Desktop/statistical learning and big data/final project data/E.RData")
load("~/Desktop/statistical learning and big data/final project data/F.RData")
load("~/Desktop/statistical learning and big data/final project data/G.RData")
load("~/Desktop/statistical learning and big data/final project data/H.RData")
load("~/Desktop/statistical learning and big data/final project data/I.RData")
load("~/Desktop/statistical learning and big data/final project data/J.RData")
```

```
#function that converts lists to array format
make_array<- function(letter_list){
  res_array<- array( 0, dim=c(length(letter_list), 28, 28)) #initialize an array
#loop through all images in list
for (i in 1:length(letter_list)){
  res_array[i,,] <- letter_list[[i]]
}
return(res_array)
}
```

```
#make arrays
A_array<- make_array(A)
B_array<- make_array(B)
C_array<- make_array(C)
D_array<- make_array(D)
E_array<- make_array(E)
F_array<- make_array(F)
G_array<- make_array(G)
H_array<- make_array(H)
I_array<- make_array(I)
J_array<- make_array(J)

rm(A, B, C, D, E, F, G, H, I, J) #clear workspace
```

```
#create label list
A_val<- rep("A",1873) #1873 images
B_val<- rep("B",1873) #1873 images
C_val<- rep("C",1873) #1873 images
D_val<- rep("D",1873) #1873 images
E_val<- rep("E",1873) #1873 images
F_val<- rep("F",416) #416 images
G_val<- rep("G",1872) #1872 images
H_val<- rep("H",1872) #1872 images
I_val<- rep("I",1872) #1872 images
J_val<- rep("J",1872) #1872 images
labels<- as.vector(c(A_val, B_val, C_val, D_val, E_val, F_val, G_val, H_val, I_val, J_val))
rm(A_val, B_val, C_val, D_val, E_val, F_val, G_val, H_val, I_val, J_val)
```

```
#bind all data together into one big array
all_data <- abind(A_array, B_array,
C_array, D_array, E_array, F_array,
G_array, H_array, I_array, J_array, along = 1)
rm(A_array, B_array, C_array, D_array, E_array, F_array,G_array, H_array, I_array, J_array )
```

```
#test train split
set.seed(123)
# Get the number of samples
n_samples <- dim(all_data)[1]

# Sample indices for the training set
train_index <- sample(n_samples, 0.75 * n_samples, replace = FALSE)

# Split the data into training and testing sets
xtrain <- all_data[train_index, , ]
xtest <- all_data[-train_index, , ]
xtrain_labels<- labels[train_index]
xtest_labels<- labels[-train_index]

rm(train_index, all_data) #clear workspace
```

```
input_size<- dim(xtrain)[2] * dim(xtrain)[3] #total number of pixels in image

# Reshape the training and testing data:
```

```
# Reshape the training data (xtrain) by keeping the number of images as the first dimension
# and setting the second dimension to be equal to the total number of pixels
x_train = array_reshape(xtrain, dim=c(dim(xtrain)[1], input_size))

# Print the dimensions of the reshaped training data (x_train) for verification
#dim(x_train)

# Reshape the testing data (xtest) in the same manner as the training data
x_test = array_reshape(xtest, dim=c(dim(xtest)[1], input_size))

# Print the dimensions of the reshaped testing data (x_test) for verification
#dim(x_test)
rm(xtrain)
```

Check range of pixel values:

```
hist(x_train[1,], breaks=50, main="Histogram of pixel values", xlab="pixel value")
```

```
# Print the dimensions of the reshaped training data (x_train) for verification
dim(x_train)
# Print the dimensions of the reshaped testing data (x_test) for verification
dim(x_test)
```

```
knitr::include_graphics("all_layers.png")
```

Hyperparameters:

```
knitr::include_graphics("loss_ten_latent.png")
knitr::include_graphics("loss_two_latent.png")
```

##E# Create architecture

Encoder:

```
set.seed(0417)
latent_size<- 10 #this is the size of the bottleneck
#ENCODER
#define input layer:
enc_input<-layer_input(shape=c(input_size)) #our input is the 28x28 pixels

#define other layers in encoder:
enc_output = enc_input %>% #first layer with pixels
  layer_dense(units=256, activation = "relu") %>% #hidden layer of the encoder
  layer_activation_leaky_relu() %>% #leaky relu prevents "dying nodes"
  layer_dense(units=128, activation = "relu") %>% #hidden layer of the encoder
  layer_activation_leaky_relu() %>%
  layer_dense(units=latent_size) %>% # "bottleneck" layer.
  layer_activation_leaky_relu()

encoder = keras_model(enc_input, enc_output)
summary(encoder)
```

Decoder:

```
set.seed(0417)
dec_input = layer_input(shape = latent_size) #our input is that latent/bottle neck size

dec_output = dec_input %>%
  layer_dense(units=128, activation = "relu") %>%
  layer_activation_leaky_relu() %>%
  layer_dense(units=256, activation = "relu") %>%
  layer_activation_leaky_relu() %>%
  layer_dense(units = input_size, activation = "sigmoid") %>%  #our output is
  #the original input, and sigmoid b/c forces output to be between 0 and 1
  layer_activation_leaky_relu()

decoder = keras_model(dec_input, dec_output)

summary(decoder)
```

Final autoencoder:

```
aen_input = layer_input(shape = input_size) #original input: 28x28

aen_output = aen_input %>% #original input
  encoder() %>%  #encoder
  decoder() #decoder

aen = keras_model(aen_input, aen_output)
summary(aen)
```

Fit network

```
#Fitting the model
aen %>% compile(
  optimizer = 'adam',  #note that the website used rmsprop instead, could choose either
  loss = 'binary_crossentropy',
)

#Train the model:
aen %>% fit(x= x_train,
            y= x_train,  #note that y is same as x b/c goal is to reconstruct input
            epochs=50, #number of times we feedforward
            batch_size=256,
            validation_data= list(x_test, x_test))
```

Encode images in test data:

```
encoded_imgs <- encoder %>%
                predict(x_test)
dim(encoded_imgs)
```

Pass encoded images through the decoder.

```
#decoder
decoded_imgs<- decoder %>%
               predict(encoded_imgs) #predicting on the output of the encoder
```

Convert data back to image dimensions:

```
#change shape of the decoded data: back to 10,000 x 28 x 28, currently 10,000 x 784
pred_images = array_reshape(decoded_imgs, dim=c(dim(decoded_imgs)[1], 28, 28))
```

Plot:

```
op <- par(mfrow=c(10,2), mar=c(1,0,0,0)) #setup: 12 images in each of 2 columns
letters_to_test<- c(4, 789, 977, 2345,3487 , 4256)
#loop through 1-10:
for (i in letters_to_test){
  plot(as.raster(xtest[i,,]))
    plot(as.raster(pred_images[i,,]))
}
```

Anomaly Detection

Anomaly Detection in not-MNIST

Calculate reconstruction error:

```
reconstruction_errors <- apply(X = x_test - decoded_imgs, MARGIN = 1,
                               FUN = function(x) sqrt(sum(x^2)))
```

```
threshold <- quantile(reconstruction_errors, probs = 0.999)
# Step 3: Identify the indices of the images with reconstruction errors
  #greater than the threshold value
anomaly_indices <- which(reconstruction_errors > threshold)
```

Plot:

```
op <- par(mfrow=c(10,2), mar=c(1,0,0,0)) #setup: 12 images in each of 2 columns
#loop through 1-10:
for (i in anomaly_indices){
  plot(as.raster(xtest[i,,]))
    plot(as.raster(pred_images[i,,]))
}
```

Anomaly Detection with MNIST

```
#load mnist
mnist <- dataset_mnist()
X_train <- mnist$train$x
X_test <- mnist$test$x
y_train <- mnist$train$y
y_test <- mnist$test$y
```

```
#doesn't matter which we use (train vs. test), and we don't need the y's since we
#aren't actually doing classification so I'm just selecting 10 images from X_train

x_mnist<- X_train[1:10, , ] #select images

x_mnist = array_reshape(x_mnist, dim=c(dim(x_mnist)[1], 784)) #reshape the 10 images

rm(mnist, X_train, X_test, y_train, y_test) #remove the rest of the data
```

```
with_mnist<- abind(x_test,x_mnist, along=1) #add mnist to not-mnist
```

```
encoded_with_mnist <- encoder %>%
                predict(with_mnist)
decoded_with_mnist<- decoder %>%
               predict(encoded_with_mnist)

mnist_reconstruction_errors <- apply(X = with_mnist - decoded_with_mnist,
                                     MARGIN = 1, FUN = function(x) sqrt(sum(x^2)))
mnist_threshold <- quantile(mnist_reconstruction_errors, probs = 0.999)
# Step 3: Identify the indices of the images with reconstruction errors
  #greater than the threshold value
mnist_anomaly_indices <- which(mnist_reconstruction_errors > mnist_threshold)

mnist_pred_images = array_reshape(decoded_with_mnist,
                                  dim=c(dim(decoded_with_mnist)[1], 28, 28))
with_mnist_2 = array_reshape(with_mnist, dim=c(dim(with_mnist)[1], 28, 28))

normalized_with_mnist_2 <- with_mnist_2 / max(with_mnist_2)
normalized_mnist_pred_images <- mnist_pred_images / max(mnist_pred_images)

op <- par(mfrow=c(10,2), mar=c(1,0,0,0)) #setup: 12 images in each of 2 columns
#loop through 1-10:
for (i in mnist_anomaly_indices){
  plot(as.raster(normalized_with_mnist_2[i,,]))
  plot(as.raster(normalized_mnist_pred_images[i,,]))
}
```

Denoising

Visual representation:

```
knitr::include_graphics("denoising_diagram.png")
```

Corrupt images:

```
set.seed(0417)
corrupt_image<- function (input){
  res_mat<- matrix(NA, nrow=nrow(input), ncol=784) #store corrupted image
  for (i in 1:nrow(input)){
    #randomly select 40% to corrupt
    pixels_to_corrupt<- sample(1:784, .4*(784), replace = FALSE)
```

```
    #randomly select 50% of those to set to 1
    pixel_to_corrupt_1<- sample(pixels_to_corrupt, .5*0.4*784, replace = FALSE)
    #set remaining to 0
    pixel_to_corrupt_0<- pixels_to_corrupt[-pixel_to_corrupt_1]

  for(j in 1:784){
    if(j %in% pixel_to_corrupt_0){res_mat[i,j]<- 0} #if in 0 vector set to 0
    else if (j %in% pixel_to_corrupt_1){res_mat[i,j]<- 1}
    else{res_mat[i,j]<- input[i,j]}
    }
  }
  return(res_mat)
}

x_train_corrupt<- corrupt_image(x_train)
x_test_corrupt<- corrupt_image(x_test)
```

Pass corrupted images through the autoencoder:

```
#Fitting the model
aen %>% compile(
  optimizer = 'adam',  #note that the website used rmsprop instead, could choose either
  loss = 'binary_crossentropy', #the website used binary_crossentropy
)

#Train the model:
aen %>% fit(x= x_train_corrupt,
            y= x_train,  #b/c in autoencoders, goal is to reconstruct input
            epochs=100, #number of times we feedforward
            batch_size=256)
```

Compare output to corrupted image input:

```
encoded_corrupted_imgs <- encoder %>%
                predict(x_test_corrupt)
decoded_corrupted_imgs<- decoder %>%
               predict(encoded_corrupted_imgs)
pred_corrupted_images = array_reshape(decoded_corrupted_imgs,
                                      dim=c(dim(decoded_corrupted_imgs)[1], 28, 28))
corrupted_images = array_reshape(x_test_corrupt, dim=c(dim(x_test_corrupt)[1], 28, 28))

op <- par(mfrow=c(10,2), mar=c(1,0,0,0)) #setup: 12 images in each of 2 columns
#loop through 1-10:
for (i in letters_to_test){
  plot(as.raster(corrupted_images[i,,]))
    plot(as.raster(pred_corrupted_images[i,,]))
}
```

Dimension Reduction

Visualization:

```
knitr::include_graphics("linear_vs_nonlinear.png")
```

Autoencoder:

```
#for 2d viz encoder
set.seed(0417)#this is the size of the bottleneck
#ENCODER
#define input layer:
enc_input<-layer_input(shape=c(input_size)) #our input is the 28x28 pixels

#define other layers in encoder:
enc_output = enc_input %>% #first layer with pixels
  layer_dense(units=256, activation = "relu") %>% #hidden layer of the encoder
  layer_activation_leaky_relu() %>% #leaky relu prevents "dying nodes"
  layer_dense(units=128, activation = "relu") %>% #hidden layer of the encoder
  layer_activation_leaky_relu() %>%
  layer_dense(units=2) %>% # "bottleneck" layer.
  layer_activation_leaky_relu()

encoder = keras_model(enc_input, enc_output)
```

```
#for 2d viz decoder
set.seed(0417)
dec_input = layer_input(shape = 2) #our input is that latent/bottle neck size

dec_output = dec_input %>%
  layer_dense(units=128, activation = "relu") %>%
  layer_activation_leaky_relu() %>%
  layer_dense(units=256, activation = "relu") %>%
  layer_activation_leaky_relu() %>%
  layer_dense(units = input_size, activation = "sigmoid") %>%
  layer_activation_leaky_relu()

decoder = keras_model(dec_input, dec_output)
```

```
#combine encoder and decoder for 2d visualization
aen_input = layer_input(shape = input_size) #original input: 28x28

aen_output = aen_input %>% #original input
  encoder() %>%  #encoder
  decoder() #decoder

aen = keras_model(aen_input, aen_output)
```

```
#fit the 2d visualization model
#Fitting the model
aen %>% compile(
  optimizer = 'adam',
  loss = 'binary_crossentropy',
)

#Train the model:
```

```
aen %>% fit(x= x_train,
            y= x_train,
            epochs=50,
            batch_size=256,
            validation_data= list(x_test, x_test))
```

```
#force down to 2d
encoded_imgs <- encoder %>%
                predict(x_test)
```

```
#create plot of 2 node representation of the data
encoded_df<- as.data.frame(encoded_imgs)
encoded_df$labs<- xtest_labels

ggplot(data=encoded_df) +
  geom_point(aes(x=V1, y=V2, color=labs)) +
  scale_color_manual(values=c("blue", "red", "green", "purple", "orange",
                              "darkblue", "brown", "cyan", "magenta", "darkgreen")) +
  theme_bw() +
  labs(title="Latent Node 1 vs. Latent Node 2", x="Latent Node 1 ", y="Latent Node 2 ")
```

```
knitr::include_graphics("dim_red_aen.png")
```

PCA:

```
#doing pca
pca_result<-prcomp(x_test, center=TRUE, scale=TRUE)
new_pts<-pca_result$x #17269 x 784

#add to data frame: new pts col1 and col 2 (first 2 PC's) and actual values
PCA_pts<-data.frame(new_pts[,1], new_pts[,2])
PCA_pts_labels<-cbind(PCA_pts, xtest_labels)
```

```
#pca plot
ggplot(data=PCA_pts_labels, aes(x=new_pts...1., y=new_pts...2., color=xtest_labels)) +
  geom_point() +
  scale_color_manual(values=c("blue", "red", "green", "purple", "orange", "darkblue", "brown", "cyan",
  labs(
    title = "Figure 12: Linear PCA, 2 Principal Components",
    x= "Principal Component 1",
    colour="Letter",
    y= "Principal Component 2") +
  theme(plot.title = element_text(hjust = 0.5))+
  theme_bw()
```

```
knitr::include_graphics("dim_red_pca.png")
```

Autencoder Extensions

Graphic for paper 2:

```
knitr::include_graphics("aen_impute_scheme.png")
```