Homework13

Jun Rao

11/10/2020

The goal of this homework is to use an auto-encoder to learn a low dimensional nonlinear mapping of a high dimensional data set and compare to the PCA linear mapping.

```
library(keras)

library(digest)

library(ggplot2)
library(data.table)
library(tidyverse)
```

1.first select 100 rows of the zip.train data from the ESL book (10 from each class).

```
# use data.table::fread to read the compressed CSV data file into R as a data
 table.
dt<- fread("zip.train.gz")

sub.dt <- data.table()

index_seq <- sort(unique(dt$V1))

for (index in index_seq) {

  sub <- subset(dt, dt$V1 == index)
  row.dt <- sub[sample(nrow(sub), 10),]

  sub.dt <- rbind(sub.dt,row.dt)
}

sub.dt$class <- rep(0:9, each=10)
```

For these data use the keras R package to define an auto-encoder with only one hidden layer with two units, using keras::keras_model_sequential and keras::layer_dense. How many parameters are there to learn in this nonlinear model? How many parameters are there to learn in the corresponding PCA linear model with rank=2? Is the number of parameters in the auto-encoder larger as expected?

```
# use data.table::fread to read the compressed CSV data file into R as a data
 table.
n.input <- ncol(sub.dt)
n.code.units <- 2
```

```
model <- keras::keras_model_sequential() %>%
  keras::layer_dense(
    input_shape = n.input, units = n.code.units, name="code") %>%
  keras::layer_dense(units = n.input)

summary(model)

## _________________________________________________________________________
_______
## Layer (type)                         Output Shape                      Param
#
## =========================================================================
======
## code (Dense)                         (None, 2)                         518

## _________________________________________________________________________
_______
## dense_1 (Dense)                      (None, 258)                       774

## =========================================================================
======
## Total params: 1,292
## Trainable params: 1,292
## Non-trainable params: 0
## _________________________________________________________________________
_______
```

There are 516 parameters to nonlinear model.

And there 771 parameters to learn in the corresponding PCA linear model with rank=2 Yes, the auto-encoder has more parameter, because the auto-enccoder is more complicated than nonlinear.

Now learn the auto-encoder parameters using keras::compile (with keras::loss_mean_squared_error) and keras::fit. Use the predict function to compute the predicted values. Also compute a PCA with rank=2 and compute its predicted values. What is the reconstruction error (mean squared error between data and predicted values) for the two methods? Is the auto-encoder more accurate as expected?

```
#compute the predit value for  keras::compile  function
compiled.model <- keras::compile(
  model,
  optimizer=keras::optimizer_sgd(),
  loss=keras::loss_mean_squared_error)


i.mat <- as.matrix(sub.dt)
compile.fit.model <- keras::fit(compiled.model, x=i.mat, y=i.mat)

compile.fit.model[["metrics"]][["loss"]]
```

```
##  [1] 1.072825 1.069979 1.067164 1.065277 1.063297 1.061531 1.060275 1.0589
62
##  [9] 1.057791 1.056649
```

```
compile.pred.mat <- predict(compiled.model, i.mat)#last layer.
compile.pred.dt <- data.table(compile.pred.mat)
names(compile.pred.dt) <- names(sub.dt)
```

```
#compute a PCA with rank=2 and compute its predicted values
pc.fit <- prcomp(sub.dt, rank = 2)
class <- rep(0:9, each=10)
PC1 <- pc.fit[["rotation"]][,1]
PC1.mat <- matrix(PC1, nrow=nrow(sub.dt),  ncol=ncol(sub.dt), byrow=TRUE)
```

```
mean.vec <- colMeans(sub.dt)
mean.mat <- matrix(mean.vec, nrow=nrow(sub.dt),  ncol=ncol(sub.dt), byrow=TRU
E)
pc.pred.mat <- mean.mat + PC1.mat * pc.fit[["x"]][, 1]
colnames(pc.pred.mat) <- colnames(sub.dt)
pc.pred.dt <- data.table(pc.pred.mat)
```

```
## sum of squared error.
compile.mse <- sum((compile.pred.mat - sub.dt)^2)
pca.mse <- sum((pc.pred.mat - sub.dt)^2)
```

```
compile.mse
```

```
## [1] 27244.18
```

```
pca.mse
```

```
## [1] 11108.47
```

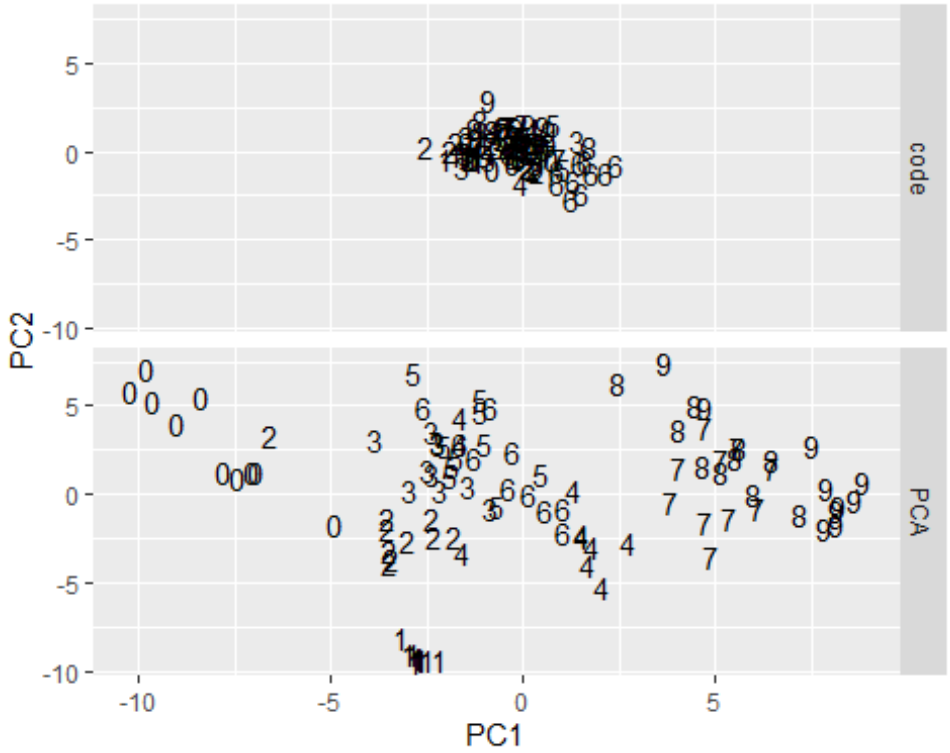
the reconstruction error for auto-encoder is 27244.18

the reconstruction error for PCA is 11108.47

Yes, ***the auto-encoder more accurate as expected***

Now use keras::keras_model, keras::get_layer, and predict functions to compute the low-dimensional embedding of the original train data. Make a ggplot with these auto-encoder embeddings in one panel, and the PCA in another panel. Use geom_text(label=digit) or geom_point(color=digit) to visualize the different digit classes. Which of the two methods results in better separation between digit classes?

```
# code layer predictions.
intermediate_layer_model <- keras::keras_model(
  inputs = compiled.model$input,
  outputs = keras::get_layer(compiled.model, "code")$output)

code.output <- data.table(
  predict(intermediate_layer_model,i.mat),
  data.type = "code",
  sub.dt
)




pc.output <- data.table(pc.fit$x,data.type = "PCA", sub.dt)




my.dt <- rbind(pc.output,code.output,use.names=FALSE)

# (my.dt <- my.dt[1:200,1:3])
ggplot() +
   facet_grid(data.type ~ .)+
  geom_text(aes(x=PC1, y=PC2,label=class),
            data = my.dt)
```




```
# intermediate_output <- predict(intermediate_layer_model, i.mat)
```

The PCA methods results in better separation between digit classes

Your job is to investigate a deep auto-encoder and see whether or not it is more prone to overfitting than the shallow auto-encoder described above.

First decide on a number of hidden units U to use in the intermediate layers. If your input data dimension is D then there should be five neural network layers with sizes (D, U, 2, U, D). Typically U should be chosen such that 2<U<D. Create a variable named model.list, which should be a list of two keras models (shallow=previous model with three layers described above, deep=new model with five layers). Make sure that in the five layer model there are non-linear activations for all layers but the last. Make a for loop over these two models, and use keras::fit(validation_split=0.5) to learn parameters for each model using a 50% subtrain, 50% validation split. Make a ggplot of y=square loss as a function of x=iterations, with different sets in different colors (e.g., subtrain=black, validation=red), and the two different models in two different panels, facet_grid(. ~ model). Does either model overfit? Finally make another ggplot which displays the low dimensional embeddings, as in problem 3 above. Which of the two methods results in better separation between digit classes?

```
#To exclude the label column (that is the first column)
df<-as.matrix(dt[, -1])
D <- 256
```

```
U <- 128
enc_input = layer_input(shape = D)
enc_output = enc_input %>%
  layer_dense(units=U, activation = "relu") %>%
  layer_activation_leaky_relu() %>%
  layer_dense(units=2)

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

## _________________________________________________________________________
_______
## Layer (type)                         Output Shape                      Param
#
## =========================================================================
======
## input_1 (InputLayer)                 (None, 256)                       0

## _________________________________________________________________________
_______
## dense_2 (Dense)                      (None, 128)                       32896

## _________________________________________________________________________
_______
## leaky_re_lu_1 (LeakyReLU)            (None, 128)                       0

## _________________________________________________________________________
_______
## dense_3 (Dense)                      (None, 2)                         258

## =========================================================================
======
## Total params: 33,154
## Trainable params: 33,154
## Non-trainable params: 0
## _________________________________________________________________________
_______
```

Next, compile the model with appropriate loss function, optimizer, and metrics:

```
dec_input = layer_input(shape = 2)
dec_output = dec_input %>%
  layer_dense(units=U, activation = "relu") %>%
  layer_activation_leaky_relu() %>%
  layer_dense(units = D, activation = "sigmoid")
   # inputsize dimensions for the output layer
decoder = keras_model(dec_input, dec_output)
summary(decoder)
```

```
## _________________________________________________________________
_____
## Layer (type)                 Output Shape              Param
#
## =================================================================
======
## input_2 (InputLayer)         (None, 2)                 0

## _________________________________________________________________
_____
## dense_4 (Dense)              (None, 128)               384

## _________________________________________________________________
_____
## leaky_re_lu_2 (LeakyReLU)    (None, 128)               0

## _________________________________________________________________
_____
## dense_5 (Dense)              (None, 256)               33024

## =================================================================
======
## Total params: 33,408
## Trainable params: 33,408
## Non-trainable params: 0
## _________________________________________________________________
_____
```

Use the fit() function to train the model for 30 epochs using batches of 100 images:

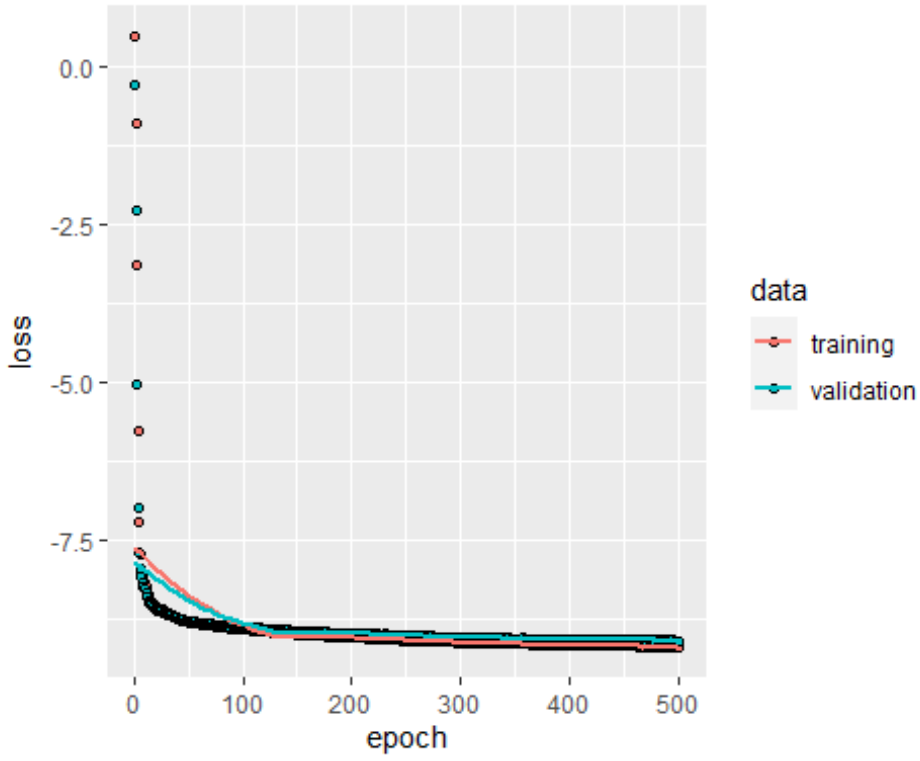
```
aen_input = layer_input(shape = D)

aen_output = aen_input %>%
  encoder() %>%
  decoder()

autoencoder_model = keras_model(aen_input, aen_output)

summary(autoencoder_model)
```

```
## _________________________________________________________________
_____
## Layer (type)                 Output Shape              Param
#
## =================================================================
======
## input_3 (InputLayer)         (None, 256)               0

## _________________________________________________________________
_____
```

```
## model_2 (Model)                     (None, 2)                         33154

## _________________________________________________________________
_____
## model_3 (Model)                     (None, 256)                       33408

## ================================================================
======
## Total params: 66,562
## Trainable params: 66,562
## Non-trainable params: 0
## _________________________________________________________________
_____
```

Use the fit() function to train the model for 500 epochs using batches of 1000 images:

```
compile.aen <- autoencoder_model %>%  compile(optimizer="rmsprop", loss="bina
ry_crossentropy")

history <-compile.aen %>% fit(df,df, epochs=500, batch_size=1000,validation_s
plit = 0.5 )

plot(history)
```



There is no method at here over fit.