Assignment 2 (Machine Learning and AI)

- Shubhang Periwal (19201104)

Abstract

The file data_usps_digits.RData contains data recording handwritten digits from the United States Postal Service (USPS). More in details, the data consists of grayscale (16x16) grid representations of image scans of the digits “0” through “9” (10 digits). We need to use a multilayer neural network with 2 hidden layers to predict the type of digit. We then need to perform analysis of the output. This would be followed by an analysis for 3 hidden layers.

Methodology and Observations

1. Loading data
    - Set a seed so that output remains consistent.
    - Data has 2 main components. i.e. x_train, y_train (training data) and x_test, y_test (testing data).
    - We take 1000 images from training set for validation.

x_test	Large matrix (513792 elements, 3.9 Mb)
x_train	Large matrix (1866496 elements, 14.3 Mb)
y_test	num [1:2007, 1:10] 0 0 0 0 0 1 1 1 0 0 ..
y_train	num [1:7291, 1:10] 0 0 0 0 0 0 0 0 1 0 ..

    - Helper plot function to visualize this data.

2. Preprocessing
    - Conversion of y to categorical using one-hot encoding.
    - Normalization of data between 0 and 1.
    - Converting x into matrix format from a data frame.

3. Model Design

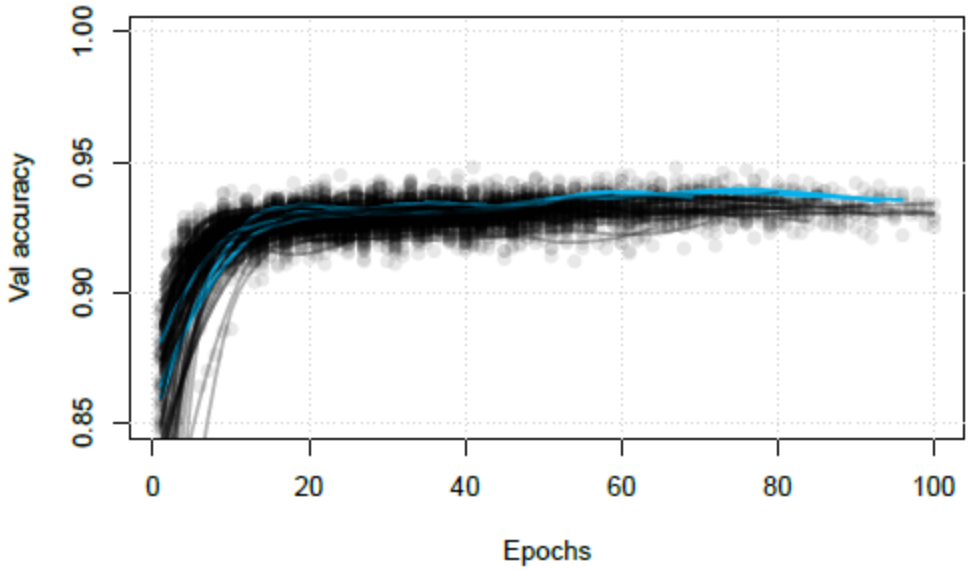
```
library(keras)

FLAGS <- flags(
  flag_integer("dense_units1", 256),
  flag_numeric("dropout1", 0.4),
  flag_integer("dense_units2", 128),
  flag_numeric("dropout2", 0.3)
)

# model configuration
model <- keras_model_sequential() %>%
  layer_dense(units =  FLAGS$dense_units1, input_shape = ncol(x_train), activation = "relu", name = "layer_1",
              kernel_regularizer = regularizer_l2(l = 0.001)) %>%
  layer_dropout(rate = FLAGS$dropout1) %>%
  layer_dense(units = FLAGS$dense_units2, activation = "relu", name = "layer_2",
              kernel_regularizer = regularizer_l2(l = 0.001)) %>%
  layer_dropout(rate = FLAGS$dropout2) %>%
  layer_dense(units = ncol(y_train), activation = "softmax", name = "layer_out") %>%
  compile(loss = "categorical_crossentropy", metrics = "accuracy",
          optimizer = optimizer_adam(lr = 0.001),
  )
```

The model consists of 3 hidden layers each with different number of nodes. We create a flag which consists of some other possible architectures that we may want to try. Due to computation limitation, I tried 86 out of 432 possible architectures.

Setting this kind of grid helps us try multiple architectures with random sampling. This can be used to reduce to the architectures and values that are actually useful and give better results.

```
#setting a grid of values for the flags/hyperparameters of interest:
hdlayer1 <- c(128,64,256)
dropout1 <- c(0,0.1,0.3)
hdlayer2 <- c(64,32)
dropout2 <- c(0,0.2)
hdlayer3 <- c(64,32,16,8)
dropout3 <- c(0,0.1,0.3)
# total combinations 3 x 3 x 2 x 2 x 4 x 3 = 432
```

- I have tried to give different kind of shapes to my neural network, with the third layer having as low as 8 nodes.
- I have also given different combinations of dropouts with 0 being one of them, to check whether no dropout improves the model or not.
- Most of the models have an accuracy between 91% and 95% and they tend to reach their optimum after 30-40 epochs and accuracy almost remains constant after that.



Here are the top 10 models

```
## Data frame: 10 x 6
##    metric_val_accuracy eval_accuracy flag_hdlayer_1 flag_dropout_1 flag_hdlayer_2 flag_dropout_2
## 1                0.942         0.9533           128            0.0           32.0            0.2
## 2                0.940         0.9543           256            0.3           32.0            0.0
## 3                0.940         0.9444           256            0.0           32.0            0.0
## 4                0.940         0.9474           256            0.0           32.0            0.2
## 5                0.940         0.9464            64            0.1           64.0            0.2
## 6                0.938         0.9494           256            0.3           64.0            0.0
## 7                0.938         0.9563           256            0.3           32.0            0.0
## 8                0.937         0.9464            64            0.1           32.0            0.0
## 9                0.937         0.9454           128            0.1           64.0            0.2
## 10               0.937         0.9513            64            0.1           32.0            0.2
```

Conclusion:

The advantages of tuning and sampling our model among multiple available models is that some particular neural network architectures show more promising result than other architectures, as well as some drop out rate perform better on different kinds of dataset. It is very difficult to know the best architecture for a given problem and hence it is advisable to use multiple architectures and see which is performing better in general, in order to get the best possible result.

Appendix

R-Code

---
title: "ML AI Assignment"
author: "Shubhang Periwal 19201104"
date: "4/19/2020"

output:
  pdf_document:
    latex_engine: xelatex

```{r}
library(keras)
tensorflow::tf$random$set_seed(0)
library(tfruns)
library(reticulate)
library(jsonlite)
 load("data_usps_digits.RData") #loading the data into R
```

```{r}
# helper function
# to plot the digit
plot_digit <- function(index, data) {
tmp <- (-data + 1) / 2 # to convert back to original
z <- matrix( data = as.numeric(data[index, 256:1]), 16, 16 )
image(z[16:1,1:16], col = gray((1:100)/100),
xaxt = "n", yaxt = "n")
}

# plot few example digits
par(mfrow = c(1,2), mar = rep(1.5, 4))
plot_digit(14, x_train)
plot_digit(900, x_train)

```

```{r}
#preprocessing
# Convert y to categorical using one-hot encoding
y_train <- to_categorical(y_train, num_classes = 10)
y_test <- to_categorical(y_test , num_classes = 10)
# convert x_train and x_test from data frame to matrix for valid network input
```

```
x_train <- as.matrix(x_train)
x_test <- as.matrix(x_test)

# normalizen x(input) to 0-1
range_norm <- function(x, a = 0, b = 1) {
( (x - min(x)) / (max(x) - min(x)) )*(b - a) + a }

x_train <- apply(x_train, 2, range_norm)
x_test <- apply(x_test, 2, range_norm)
range(x_train)
range(x_test)
```

```{r}
#coverting the x datasets to matrices:
x_train<-as.matrix(x_train)
x_test<-as.matrix(x_test)

```

```{r}
# split the test data in two halves: one for validation
# and the other for actual testing
val <- sample(1:nrow(x_test), 1000) # there are 10000 images in x_test
test <- setdiff(1:nrow(x_test), val)
x_val <- x_test[val,]
y_val <- y_test[val,]
x_test <- x_test[test,]
y_test <- y_test[test,]
# need these later
N <- nrow(x_train)
V <- ncol(x_train)
```

```{r}
```

```
#setting a grid of values for the flags/hyperparameters of interest:
hdlayer1 <- c(128,64,256)
dropout1 <- c(0,0.1,0.3)
hdlayer2 <- c(64,32)
dropout2 <- c(0,0.2)
hdlayer3 <- c(64,32,16,8)
dropout3 <- c(0,0.1,0.3)
# total combinations 3 x 3 x 2 x 2 x 4 x 3 = 432
```

```{r}
# run ----------------------------------------------------------------
runs <- tuning_run("assignment3config.R", #creating runs to simulate output
                   runs_dir = "runs_assignment",
                   flags = list(
                  hdlayer_1 = hdlayer1,
                  dropout_1 = dropout1,
                  hdlayer_2 = hdlayer2,
                  dropout_2 = dropout2,
                  hdlayer_3 = hdlayer3,
                  dropout_3 = dropout3
                    ),
                 sample = 0.2)
#sampling 86 models

```

```{r}
#Determing the optimal configuration for the data
#Extracting values from the stored runs

read_metrics <- function(path, files =NULL)
{
path <- paste0(path, "/")
if(is.null(files)) files <- list.files(path)
n <- length(files)
out <- vector("list", n)
for(i in 1:n) {
dir <- paste0(path, files[i], "/tfruns.d/")
out[[i]] <- jsonlite::fromJSON(paste0(dir, "metrics.json"))
out[[i]]$flags <- jsonlite::fromJSON(paste0(dir, "flags.json"))
```

```
out[[i]]$evaluation <- jsonlite::fromJSON(paste0(dir,"evaluation.json"))
}
return(out)
}
#Plotting the corresponding validation learning curves
plot_learning_curve <- function(x, ylab = NULL, cols = NULL, top = 3,
span = 0.4, ...)
{
smooth_line <- function(y) {
x <- 1:length(y)
out <- predict(loess(y~x, span = span))
return(out)
}
matplot(x, ylab = ylab, xlab = "Epochs", type = "n", ...)
grid()
matplot(x, pch = 19, col = adjustcolor(cols, 0.3), add = TRUE)
tmp <- apply(x, 2, smooth_line)
tmp <- sapply(tmp, "length<-", max(lengths(tmp)))
set <- order(apply(tmp, 2, max, na.rm = TRUE), decreasing = TRUE)[1:top]
cl <- rep(cols, ncol(tmp))
cl[set] <- "deepskyblue2"
matlines(tmp, lty = 1, col = cl, lwd = 2)
}
```

```{r}
# extract results
out <- read_metrics("runs_assignment")
# extract validation accuracy and plot learning curve
acc <- sapply(out, "[[", "val_accuracy")
plot_learning_curve(acc, col = adjustcolor("black", 0.3), ylim = c(0.85, 1),ylab = "Val accuracy", top =
3)
```

```{r}
res1<- ls_runs(metric_val_accuracy > 0.87, runs_dir = "runs_assignment", order =
metric_val_accuracy)
res1
```

```{r}
```

```
res1 <- res1[,c(2,4,8:13)]
res1[1:10,]
```

```
#======== DATA USPS DIGITS - Model and settings configuration

# model instantiation ---------------------------------------------

library(keras)

FLAGS <- flags(
  flag_integer("dense_units1", 256),
  flag_numeric("dropout1", 0.4),
  flag_integer("dense_units2", 128),
  flag_numeric("dropout2", 0.3)
)

# model configuration
model <- keras_model_sequential() %>%
  layer_dense(units =  FLAGS$dense_units1, input_shape = ncol(x_train), activation = "relu", name =
"layer_1",
              kernel_regularizer = regularizer_l2(l = 0.001)) %>%
  layer_dropout(rate = FLAGS$dropout1) %>%
  layer_dense(units = FLAGS$dense_units2, activation = "relu", name = "layer_2",
              kernel_regularizer = regularizer_l2(l = 0.001)) %>%
  layer_dropout(rate = FLAGS$dropout2) %>%
  layer_dense(units = ncol(y_train), activation = "softmax", name = "layer_out") %>%
  compile(loss = "categorical_crossentropy", metrics = "accuracy",
          optimizer = optimizer_adam(lr = 0.001),
  )

fit <- model %>% fit(
  x = x_train, y = y_train,
  validation_data = list(x_val, y_val),
  epochs = 100,
  batch_size = 32,
  verbose = 1,
  callbacks = callback_early_stopping(monitor = "val_accuracy", patience = 20)
)

# store accuracy on test set for each run
score <- model %>% evaluate(
```

```
  x_test, y_test,
  verbose = 0
)
```