Deep Learning Lecture 1 - Boston Housing Price: Densily connected NN

MA8701 General Statistical Methods

Thiago G. Martins, Department of Mathematical Sciences, NTNU

Spring 2019

The Boston housing price dataset

The objective here is to predict the median price of homes in a given Boston suburb in the mid-1970s, given data points about the suburb at the time, such as the crime rate, the local property tax rate, and so on.

Preparing the data

- Few data points: only 506, split between 404 training samples and 102 test samples.
- Each feature in the input data (for example, the crime rate) has a different scale.
- For instance, some values are proportions, which take values between 0 and 1; others take values between 1 and 12, others between 0 and 100, and so on.

```
dataset <- dataset_boston_housing()
c(c(train_data, train_targets), c(test_data, test_targets)) %<-% dataset
```

```
str(train_data)
```

```
##  num [1:404, 1:13] 1.2325 0.0218 4.8982 0.0396 3.6931 ...
```

```
str(test_data)
```

```
##  num [1:102, 1:13] 18.0846 0.1233 0.055 1.2735 0.0715 ...
```

The targets are the median values of owner-occupied homes, in thousands of dollars:

```
str(train_targets)
```

```
##  num [1:404(1d)] 15.2 42.3 50 21.1 17.7 18.5 11.3 15.6 15.6 14.4 ...
```

A widespread best practice to deal with such data is to do feature-wise normalization.

```
mean <- apply(train_data, 2, mean)
std <- apply(train_data, 2, sd)
train_data <- scale(train_data, center = mean, scale = std)
test_data <- scale(test_data, center = mean, scale = std)
```

Note that the quantities used for normalizing the test data are computed using the training data. You should never use in your workflow any quantity computed on the test data, even for something as simple as data normalization.

Model building

Because so few samples are available, you'll use a very small network with two hidden layers, each with 64 units.

Because you'll need to instantiate the same model multiple times, you use a function to construct it.

```
build_model <- function() {
  model <- keras_model_sequential() %>%
    layer_dense(units = 64, activation = "relu",
                input_shape = dim(train_data)[[2]]) %>%
    layer_dense(units = 64, activation = "relu") %>%
    layer_dense(units = 1)
  model %>% compile(
    optimizer = "rmsprop",
    loss = "mse",
    metrics = c("mae")
  )
}
```

The network ends with a single unit and no activation (it will be a linear layer).

You're also monitoring a new metric during training: mean absolute error (MAE). It's the absolute value of the difference between the predictions and the targets. For instance, an MAE of 0.5 on this problem would mean your predictions are off by $500 on average.

Model validation

K-fold CV: Small dataset

```
# randomly allocate each sample to a particular fold
k <- 4
```

```
indices <- sample(1:nrow(train_data))
folds <- cut(indices, breaks = k, labels = FALSE)
```

```
# Run CV and keep the MAE scores
num_epochs <- 100
all_scores <- c()
for (i in 1:k) {
  cat("processing fold #", i, "\n")

  val_indices <- which(folds == i, arr.ind = TRUE)
  val_data <- train_data[val_indices,]
  val_targets <- train_targets[val_indices]
  partial_train_data <- train_data[-val_indices,]
  partial_train_targets <- train_targets[-val_indices]

  model <- build_model()

  model %>% fit(partial_train_data, partial_train_targets,
                epochs = num_epochs, batch_size = 1, verbose = 0)

  results <- model %>% evaluate(val_data, val_targets, verbose = 0)
  all_scores <- c(all_scores, results$mean_absolute_error)
}
```

```
## processing fold # 1
## processing fold # 2
## processing fold # 3
## processing fold # 4
```

Running this with num_epochs = 100 yields the following results:

```
all_scores
```

```
## [1] 2.923744 2.444761 2.376462 2.208323
```

```
mean(all_scores)
```

```
## [1] 2.488323
```

Let's try training the network a bit longer: 500 epochs. To keep a record of how well the model does at each epoch, you'll modify the training loop to save the per-epoch validation score log.

```
num_epochs <- 500
all_mae_histories <- NULL
for (i in 1:k) {
  cat("processing fold #", i, "\n")

  val_indices <- which(folds == i, arr.ind = TRUE)
  val_data <- train_data[val_indices,]
  val_targets <- train_targets[val_indices]

  partial_train_data <- train_data[-val_indices,]
  partial_train_targets <- train_targets[-val_indices]

  model <- build_model()

  history <- model %>% fit(
    partial_train_data, partial_train_targets,
    validation_data = list(val_data, val_targets),
    epochs = num_epochs, batch_size = 1, verbose = 0
  )
  mae_history <- history$metrics$val_mean_absolute_error
  all_mae_histories <- rbind(all_mae_histories, mae_history)
}
```

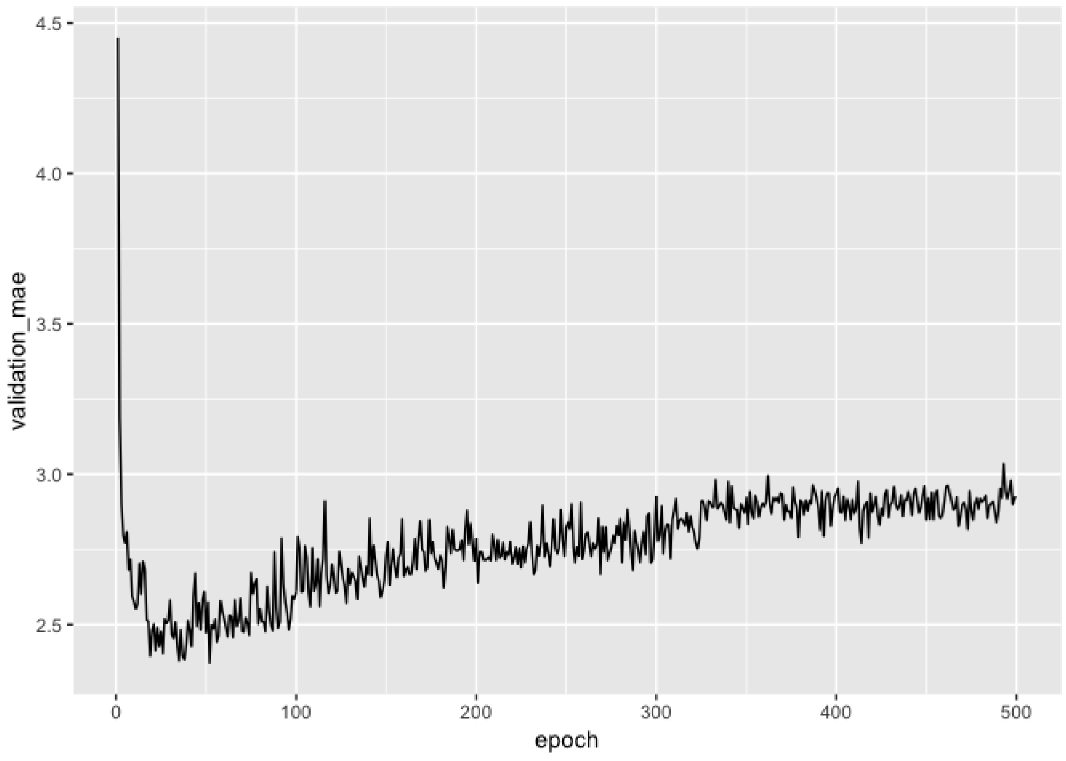
```
## processing fold # 1
## processing fold # 2
## processing fold # 3
## processing fold # 4
```

You can then compute the average of the per-epoch MAE scores for all folds.

```
average_mae_history <- data.frame(
  epoch = seq(1:ncol(all_mae_histories)),
  validation_mae = apply(all_mae_histories, 2, mean)
)
```

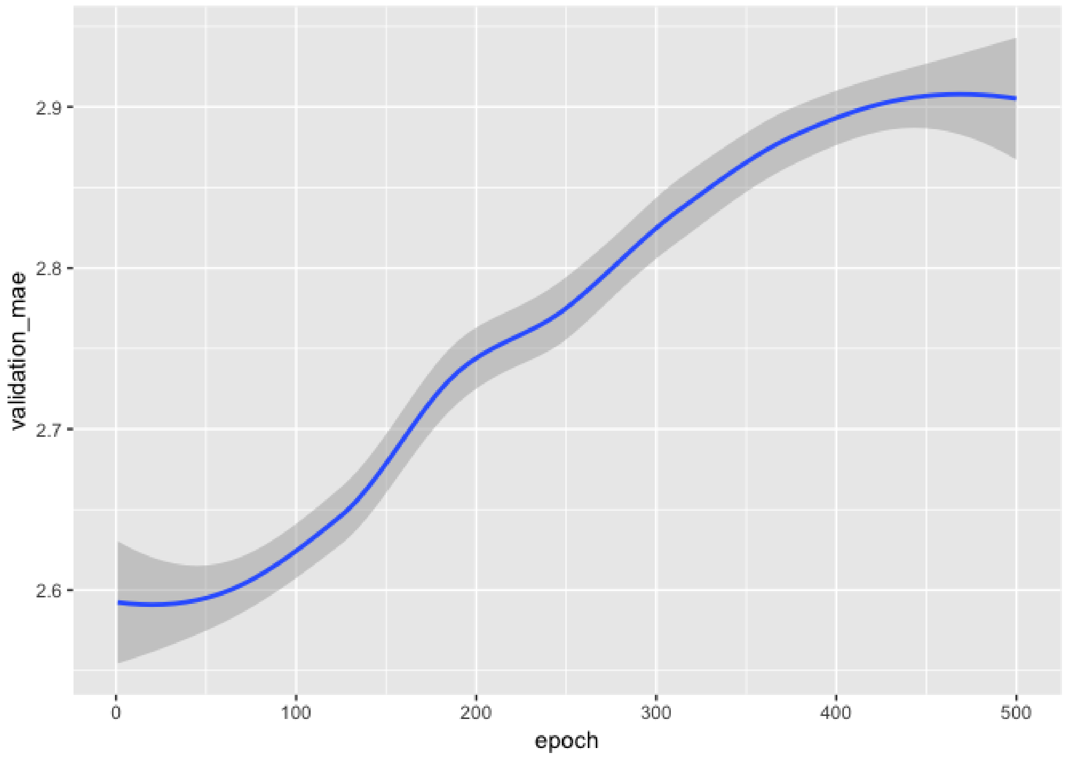
Let’s plot this:

```
library(ggplot2)
ggplot(average_mae_history, aes(x = epoch, y = validation_mae)) + geom_line()
```



Let's use `geom_smooth()` to try to get a clearer picture:

```
ggplot(average_mae_history, aes(x = epoch, y = validation_mae)) + geom_smooth()
```



According to this plot, validation MAE stops improving significantly after 125 epochs. Past that point, you start overfitting.

Once you're finished tuning other parameters of the model (in addition to the number of epochs, you could also adjust the size of the hidden layers), you can train a final production model on all of the training data, with the best parameters, and then look at its performance on the test data.

```
model <- build_model()
model %>% fit(train_data, train_targets,
          epochs = 80, batch_size = 16, verbose = 0)
result <- model %>% evaluate(test_data, test_targets)

result
```

```
## $loss
## [1] 18.14386
##
## $mean_absolute_error
## [1] 2.689312
```

You're still off by about $2,540.

Recommended exercise 3

Try to improve the MAE on the test data by changing model parameters such as number of layers and number of hidden units in each layer.