Project1

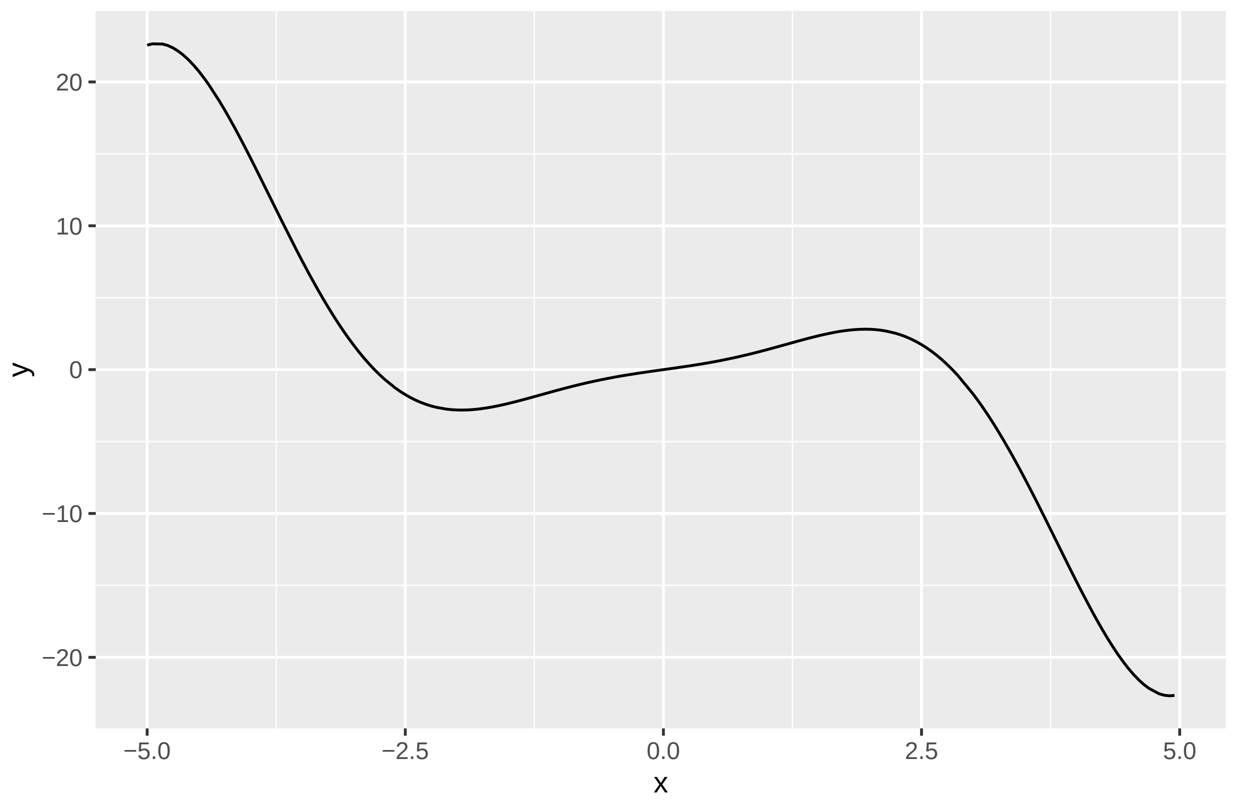
Anton Holm

2022-01-21

```r
suppressWarnings(suppressMessages(library(tidyverse)))
suppressWarnings(suppressMessages(library(keras)))
suppressWarnings(suppressMessages(library(tensorflow)))
```

```r
## Read data, divide into training/test and plot the training data
set.seed(931031)
df <- read.table("data.txt", sep = ",")


train <- df %>%
  slice_sample(prop = 0.9)

test <- anti_join(df, train, by = c("V1", "V2"))


ggplot(train, aes(x = V1, y = V2)) +
  geom_line() +
  labs(title = "Figure 1: Visualization of our training data", x = "x", y = "y")
```

Figure 1: Visualization of our training data

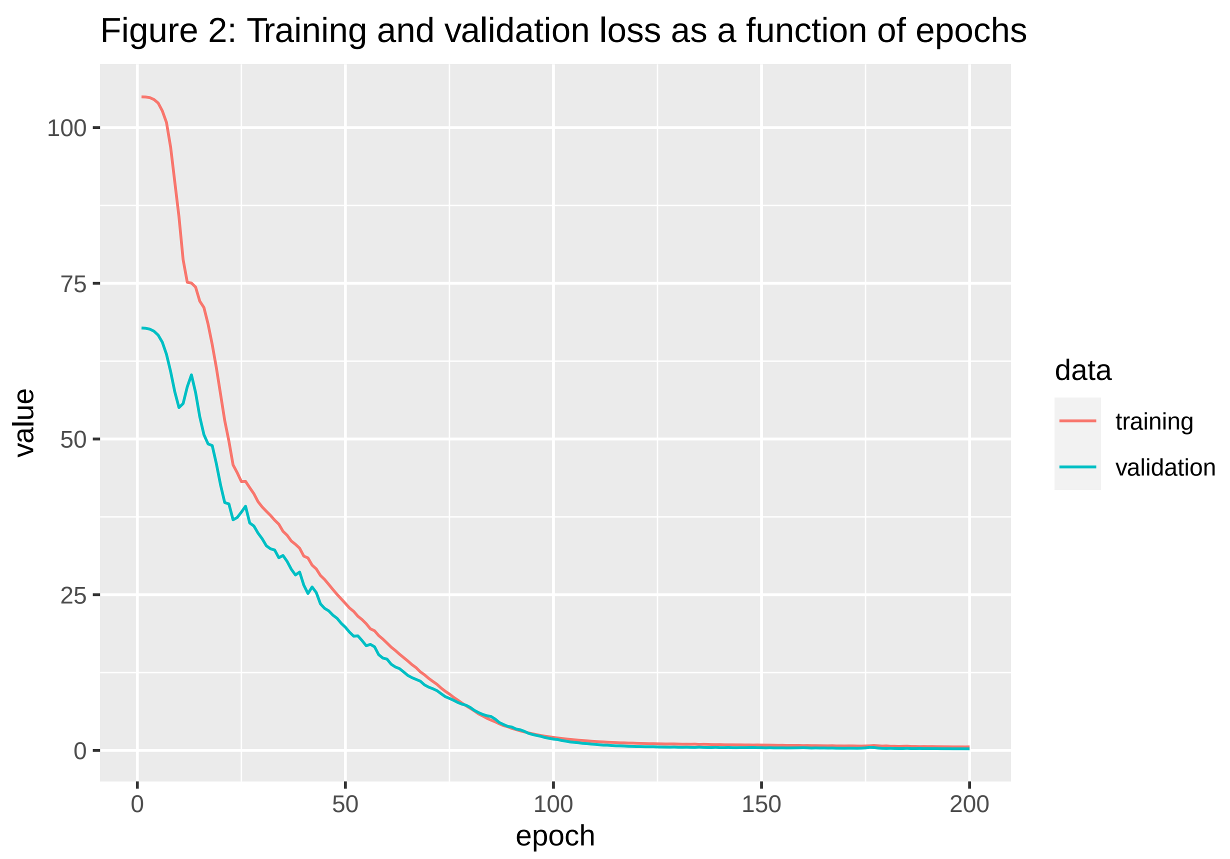


In Figure 1 we can see a visualization of our training data. Since any assumption regarding what model best fits our data should be independent of our test data, I only plot the training data leaving the test data out. We can clearly see that a linear model would not make a good fit for this data. At the same time we should not need a very complex function to fit to the data. From this I experimented with different combination of layers and found a Neural Network with good predictions that use three hidden layers with 8, 6, and 4 hidden units in them respectively. Since our input variable is one-dimensional we will have one unit in the input layer. Since our output variable is one-dimensional and we are doing regression, our output layer will have one unit and the output function is a linear function. For every hidden layer I will use the relu function as our activation function based once again on the fact that we are doing regression. The cost function I use is the mean squared error considering we are doing regression. Here we could also use the mean absolute error. One of the differences is that the MSE cost function punish big misspredictions more than if we use the MAE. Which cost function that should be used would be decided by what data we have and if we want to punish large misspredictions more or not. Since we do not have any information about what this data is, I will default to MSE. For optimizing the model I will use Adam instead of stochastic gradient descent. Adam is a version of sgd where we have an adaptive learning rate which depends on the first and second moments of the gradient. In order to speed up the training, but most importantly introduce some randomness to our training in order to escape local minima, I will use batches. This is implied when using Adam, or any type of SGD. I experimented with the batch size and found that a size of 32 worked the best. If I used larger batch sizes the training and validation error plateaued at quite large values and could imply that we were stuck in a local minima. Changing the batch size to a smaller value solved this problem. Furthermore, 20% of the batch is used for validation in every epoch in order to see if the validation loss has converged aswell as to look for overfitting. To follow the reasoning in class I start out with initial bias terms equal to zero and initial weights drawn from a gaussian distribution with mean equal to zero and small variance. I chose a small variance as to not get too large weights which would from the start make the model more complex. For the learning rate I tried 3 different values, 0.1, 0.01, and 0.001. For the smallest value the training was too slow. For the larger learning rate the training and validation loss plateaued which could be for many reasons, e.g. that we miss the minima because of too large jumps back and forth. Since the training was quick, I could try a large value of epochs. I noticed that it quite quickly started to converge and as such I lowered the number of epochs to 200 in order to make the results easier and faster to reproduce.

```r
set_random_seed(931031)
```

```
## Loaded Tensorflow version 2.7.0
```

```r
#Decide on the architecture of the neural network
model <-
  keras_model_sequential() %>%
  layer_flatten(input_shape = c(1,1)) %>%  #Input layer
  layer_dense(units = 8, activation = "relu",
              kernel_initializer = initializer_random_normal(stddev = 0.01),
              bias_initializer = initializer_constant(0)) %>%  #3 hidden layers with relu
  layer_dense(units = 4, activation = "relu",
              kernel_initializer = initializer_random_normal(stddev = 0.01),
              bias_initializer = initializer_constant(0)) %>%
  layer_dense(units = 2, activation = "relu",
              kernel_initializer = initializer_random_normal(stddev = 0.01),
              bias_initializer = initializer_constant(0)) %>%
  layer_dense(units = 1, activation = "linear") #output layer


#I use adam for optimization and MSE as a loss function since we are doing regression
model %>% compile(
  optimizer = optimizer_adam(learning_rate = 0.01),
  loss = 'mse'
)


history <- model %>% fit(
  x = as.matrix(train$V1),
  y = as.matrix(train$V2),
  epochs = 200,
  batch_size = 32, #Size of sample in SGD
  validation_split = 0.2 #20% of training data used to check validation loss
)

mse <- function(actual, predicted) {
  mean((actual - predicted)^2)
}

predicted <- model %>%
  predict(as.matrix(test$V1))

mse1 <- mse(test$V2, predicted)

history %>%
  as_tibble() %>%
  ggplot(aes(x = epoch, y = value, col = data)) +
  geom_line() +
  labs(title = "Figure 2: Training and validation loss as a function of epochs")
```

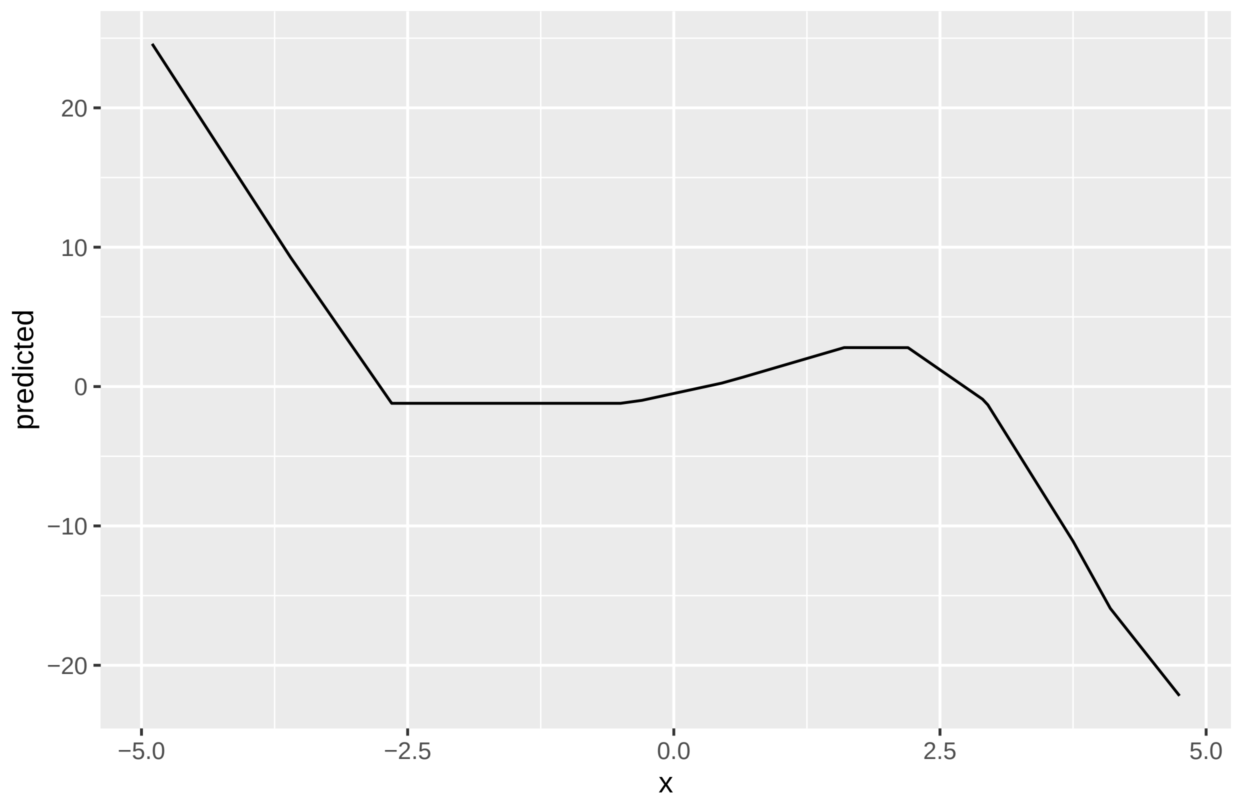


```r
save_model_hdf5(model, "model.h5")
```

We can see in Figure 2 that both the training and validation loss is starting to flatten out at around 100 epochs. However, the validation error is still not at zero and has yet to start increasing so we could increase the number of epochs further to get an even better model. By using the test data we constructed at the beginning we can calculate the MSE. We get a MSE of 0.447 which is quite low and as such the model can already predict well and as such we can stop training at around 100 epochs. We could ofcourse fine tune the hyperparameters even further, trying different initial weight and bias values, more epochs and even more different learningrates if we want. We can most likely also find different architectures and hyper-parameters in order to speed up the convergence to decrease the number of epochs but that is not needed in this case. Would our validation loss start to increase it would be an indication of overfitting. In this case we could use some stopping criteria to make sure we stop training the model when the validation error starts to increase. We could do this by e.g. deciding that if we for 50 consecutive epochs lie above the smallest validation error we have had, we stop training the model. Lastly by looking at Figure 3 we can see that our fitted function looks quite similar to the true function which corroborates the low MSE.

```r
fit_func <- as_tibble(x = test$V1, y = predicted)
ggplot(test, aes(x = V1, y = predicted)) +
  geom_line() +
  labs(title = "Figure 3: Fitted function", x = "x")
```

Figure 3: Fitted function


predicted
x