Introduction to Neural Networks

Zachary Blanks

Introduction

For our first project we're going to use the classic Boston housing dataset to help us understand some key ideas for neural networks. In particular we are going to emphasize how to use the Keras API and discuss key hyper-parameters for neural architectures.

To start, let's get the data which is conveniently available from the Keras library.

```r
library(keras)
boston = dataset_boston_housing()

# Get the training and test sets
X_train = boston$train$x
y_train = boston$train$y
X_test = boston$test$x
y_test = boston$test$y
```

Exploratory Analysis/Warm-up Exercises

Before we dive into using neural networks, I want to give everyone a chance to get used to using R (and in particular the tidyverse) if you have not done so in a while. "Tidy" concepts will also come up more heavily with Phil's lecture so I want to give everyone a chance to refresh before then, as well.

To start, let's do two simple exploratory analysis exercises. The first will be exploring the relationship of the Charles River to home price.

```r
library(tidyverse)
col_names = c('CRIM', 'ZN', 'INDUS', 'CHAS', 'NOX', 'RM', 'AGE',
              'DIS', 'RAD', 'TAX', 'PTRATIO', 'B', 'LSTAT')

train_df = as_tibble(X_train)
colnames(train_df) = col_names
train_df$PRICE = y_train
```

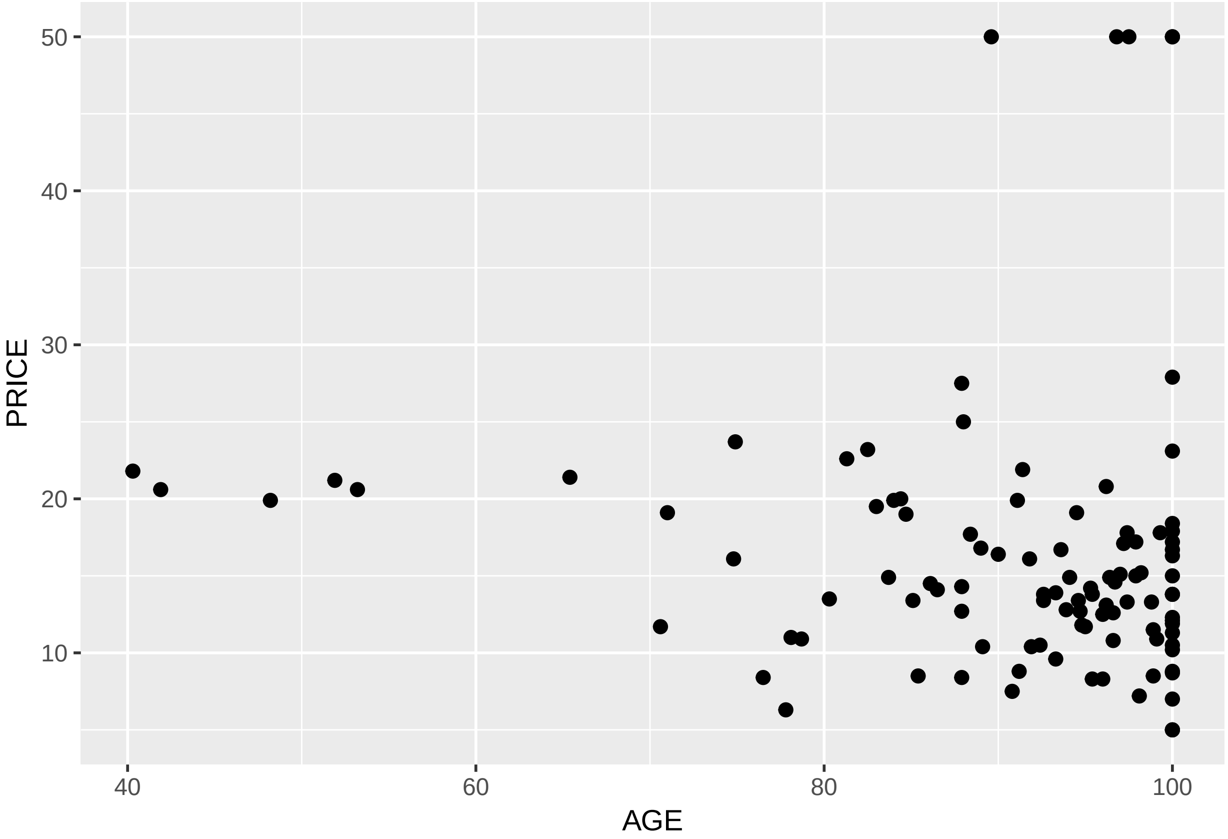
In particular, I want you to tell me how a home being bounded by the Charles River affects the average home price.

```r
train_df %>%
  group_by(CHAS) %>%
  summarize(mean(PRICE))
```

```
## # A tibble: 2 x 2
##    CHAS `mean(PRICE)`
##   <dbl>         <dbl>
## 1     0          22.0
## 2     1          28.4
```

For our final warm-up, I also want to remind us how to use gglot as well as combining this with dplyr commands. One of the features in the data is RAD this indicates the "index of accessibility" to radial highways. One of these indices is 24. Additionally there is a feature called AGE which defines the proportion of houses built before 1940. For this exercise, I want you to focus on instances where the RAD is 24 and then plot their relation of the home AGE to its price. Tell me what you see.

```
train_df %>%
  filter(RAD == 24) %>%
  ggplot(aes(x=AGE, y=PRICE)) +
  geom_point(size=2)
```



Now that we're warmed up on R, let's dive into neural networks.

Data Pre-Processing

Before we begin, it is a standard practice to normalize the features to (a) speed up the convergence rate of the model and (b) ensure that all of the features are scaled to the same units. Additionally, one should never use the test data to normalize features – only use the training data and apply the inferred mean and variance to the test set.

```
# Get the mean and sd for each column
mu_vect = apply(X_train, 2, mean)
sigma_vect = apply(X_train, 2, sd)

# Normalize the training and testing data
X_train = sweep(X_train, 2, mu_vect, "-", check.margin = F)
X_train = sweep(X_train, 2, sigma_vect, "/", check.margin = F)

X_test = sweep(X_test, 2, mu_vect, "-", check.margin = F)
X_test = sweep(X_test, 2, sigma_vect, "/", check.margin = F)
```

Intro to the Keras API

To implement a neural network in Keras, we need three primary components: a model which defines the neural network, a compiler which says how the model should be optimized, and a fitting mechanism which states how the model should be trained. Let's go through each of these components and talk about what that looks like programatically and then we will apply those ideas in some exercises.

Keras Model

The first component to define a neural network in Keras is telling the system what the "architecture" of the network looks like. To do this we must first define an empty model by typing

```
model <- keras_model_sequential()
```

By doing this we have told the R interpreter that we are defining an empty model. The phrase "sequential" in the above code means that we will add layers one after another. There are alternative ways to define model objects in Keras that are more complicated, but allow you greater control. If you are interested these are discussed at this link.

After we have define our empty model we need to add layers to the neural network. We can add as many or as few as would like and is done by typing

```
model %>%
  layer_dense(units = 32, activation = "relu",
              input_shape = dim(X_train)[2]) %>%
  layer_dense(units = 1, activation = "linear")
```

There is quite a bit going on a few lines of code, so let me break down in each part. First, the term "layer_dense" means that we are defining a new layer for the neural network. Within the *layer_dense* function, we have to provide a few key arguments. The most important one is *units*. This argument defines the number of nodes or perceptrons the model will have at that particular layer. Additionally, we have the argument *activation*; this tells the model which activation function to use for this layer (we will discuss and implement these soon). For the first layer of your model, you always have to provide the *input_shape*. This tells the network what sort of matrix it should expect. You do not have to specify this for future layers because the dimensionality will be inferred. Finally, the last layer of the network must have the number of units proportional to the number of outputs in your data. For our simple example, $y \in \mathbb{R}$ and so we only need one output node. However, if we are solving a classification problem with more than two labels, then we will need more output nodes.

Now that we have fully defined our model, we can look at all its components and see useful summary information by typing

```
summary(model)
```

```
## ___________________________________________________________________________
## Layer (type)                     Output Shape                  Param #
## ===========================================================================
## dense_1 (Dense)                  (None, 32)                    448
## ___________________________________________________________________________
## dense_2 (Dense)                  (None, 1)                     33
## ===========================================================================
## Total params: 481
## Trainable params: 481
## Non-trainable params: 0
## ---------------------------------------------------------------------------
```

The summary above tells us how many and the type of layers in our model as well as the number of parameters that have to be inferred during the training process. Looking at this summary can be a useful indicator to see if you have correctly defined your model as well as determining if it can be feasibly trained on your system.

Keras Compiler

The second component to defining a neural network in Keras is to tell the system how you want the model to be optimized. I will provide a brief example of how this can be done

```r
model %>% compile(
  loss = 'mse',
  optimizer = optimizer_sgd(),
  metrics = c("accuracy")
)
```

Once again, a bunch of information has been given in a small number of lines. In general when compiling Keras models, the system needs two pieces of information: the loss function and the optimizing algorithm. For regression problems, the most common loss function is the mean squared error which is defined as

$$\mathcal{L}(\mathbf{y}, \hat{\mathbf{y}}) = \sum_{i=1}^{n} (y_i - \hat{y}_i)^2$$

where $\hat{y}_i$ is defined as the predicted value for the i^{th} sample in the data. For classification problems, the most common loss function is either binary cross-entropy for binary classification or categorical cross-entropy for multi-class situations. The latter loss function is defined as

$$\mathcal{L}(\mathbf{Y}, \hat{\mathbf{Y}}) = -\sum_{i=1}^{n} \sum_{j=1}^{C} y_{ij} \log(\hat{y}_{ij})$$

where the original, $\mathbf{y}$ vector has been one-hot encoded (explained more later) and $\hat{\mathbf{Y}}$ is the matrix of predictions where each row defines a valid probability distribution over all possible labels.

Additionally, the *optimizer* argument defines which optimization algorithm to call when fitting the neural network. There are a large number of options and we will have a chance to play around with them later.

Keras Model Fitting

The final component needed to train your neural network is to use the model and compiler that were just defined to fit the data. An example of this code is shown below

```r
model %>% fit(
  x = X_train, y = y_train,
  epochs = 10, verbose = 1,
  validation_split = 0.25,
  batch_size = 128
)
```

Let's step through each key argument. First, the x and y arguments define the training data that will be used to fit the model. Next, *epochs* defines how many times the training process should go over the full training set. Next, *validation_split* defines the proportion of data to use for the validation set. You can either provide a validation set with the *validation_data* argument or have the system automatically split your training data. This argument is important because it allows us to evaluate how a set of hyper-parameters is doing for the model. Finally, the *batch_size* defines how many samples we will use to compute a gradient for the model. It is posible to put this value anywhere between $[1, n]$, but assuming you have a large amount of data, then it's typical to put this value in the range of $32, 64, 128, 256, 512$ because these are small enough sample sizes where gradients can be computed quickly. We also have them as powers of two because sometimes these helps with memory storage.

With those three parts, we have defined our model's architecture, specified the compiler, and fit the neural network to training data. This pattern can be generalized to any situation where we want to utilize Keras and neural networks for a supervised training situation. Now that we know how to define neural networks in Keras, let's do some exercises to gain some intution about the key parts of the model we discussed earlier.

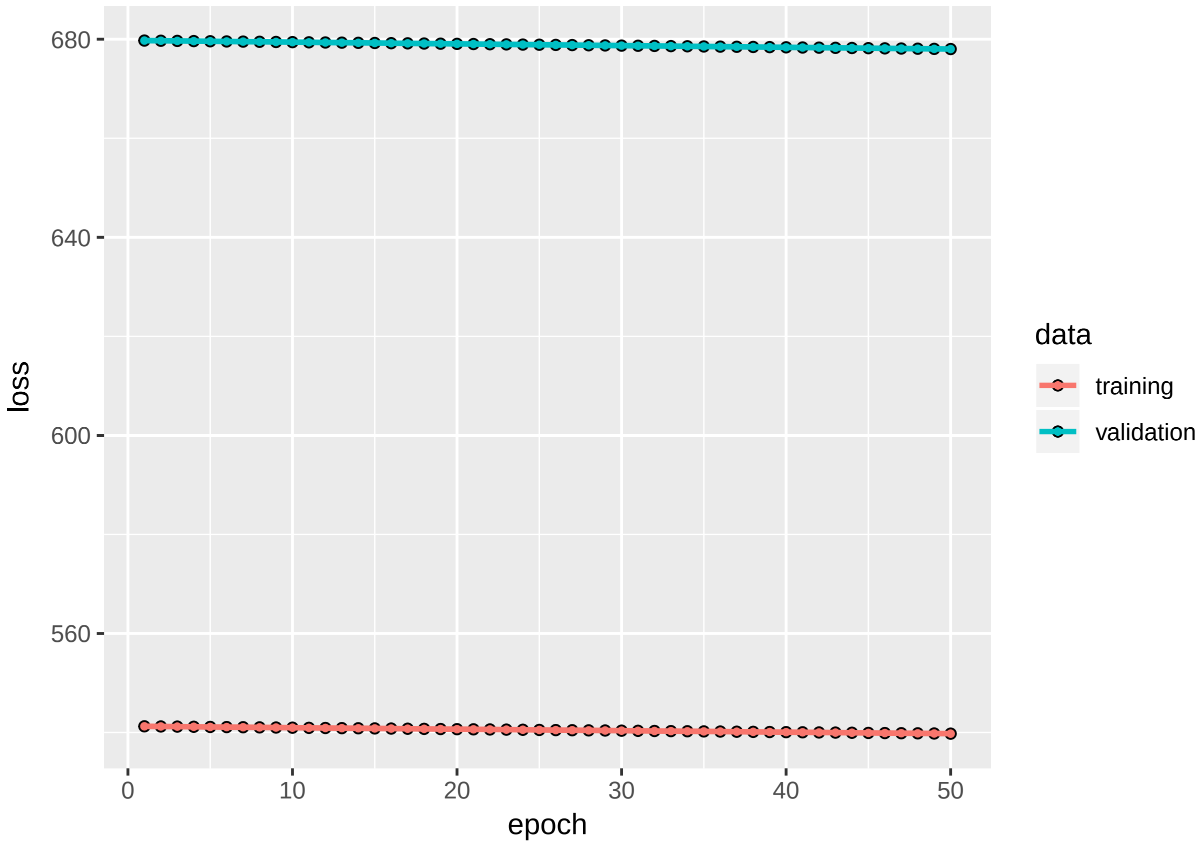
Learning Rate

One of the most important hyper-parameters for neural networks is the learning rate. This value defines how much we should update the weights in our model based on the value of the computed gradient at a particular step. To see how this value affects the model, while keeping everything else the same (i.e. architecture, number of epochs, etc.), try the following three learning rates: $\alpha = 1 \times 10^{-6}, 1 \times 10^{-3}, 1$ and we will discuss the results. To adjust the learning rate, take a look at the *optimizer_sgd* function.

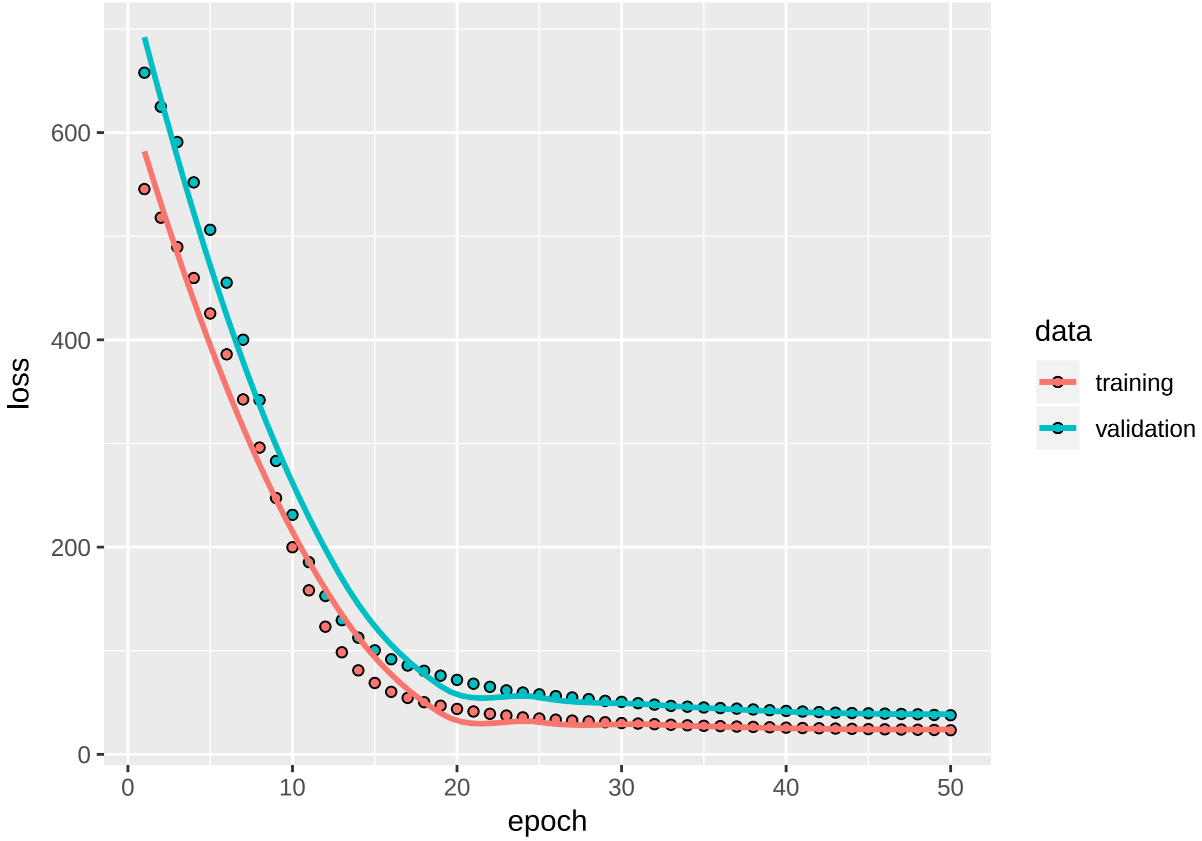
```r
# Define the model with lr=1e-6r
lr1 = 1e-6

model1 = keras_model_sequential() %>%
  layer_dense(units = 32, activation = "relu",
              input_shape = dim(X_train)[2]) %>%
  layer_dense(units = 1)

model1 %>% compile(
  loss = 'mse',
  optimizer = optimizer_sgd(lr = lr1)
)

res1 = model1 %>% fit(
  x = X_train, y = y_train, epochs = 50, verbose = 0, validation_split = 0.25,
  batch_size = 128
)

# Model 2 with lr = 1e-3
lr2 = 1e-3

model2 = keras_model_sequential() %>%
  layer_dense(units = 32, activation = "relu",
              input_shape = dim(X_train)[2]) %>%
  layer_dense(units = 1)

model2 %>% compile(
  loss = 'mse',
  optimizer = optimizer_sgd(lr = lr2)
)

res2 = model2 %>% fit(
  x = X_train, y = y_train, epochs = 50, verbose = 0, validation_split = 0.25,
  batch_size = 128
)

# Model 3 with lr = 1
lr3 = 1

model3 = keras_model_sequential() %>%
  layer_dense(units = 32, activation = "relu",
              input_shape = dim(X_train)[2]) %>%
  layer_dense(units = 1)

model3 %>% compile(
  loss = 'mse',
  optimizer = optimizer_sgd(lr = lr3)
)

res3 = model3 %>% fit(
  x = X_train, y = y_train, epochs = 50, verbose = 0, validation_split = 0.25,
  batch_size = 128
)
```

Let's take a look at each of the resulting loss profiles.

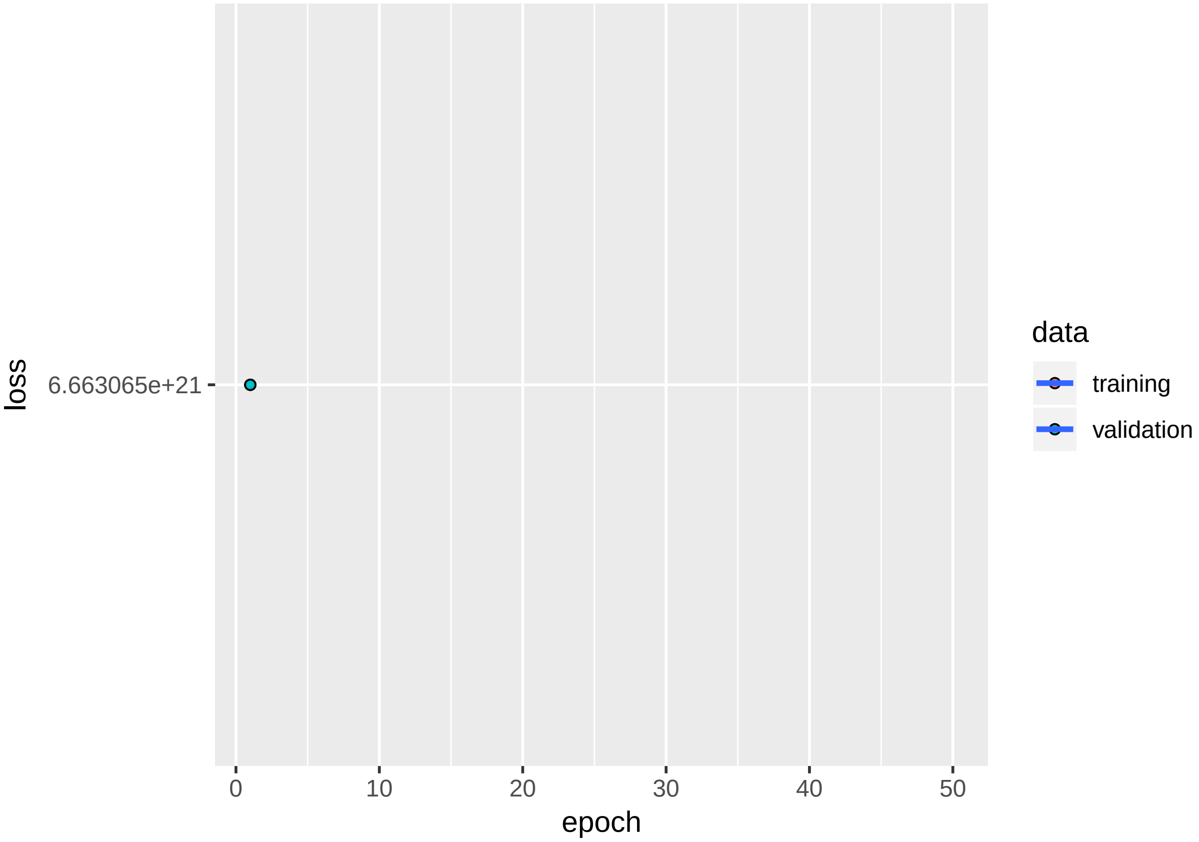
```
plot(res1)
```



```
plot(res2)
```



```
plot(res3)
```



Let's briefly discuss why we're seeing the particular results for each of the learning rates. For the model with $\alpha = 1 \times 10^{-6}$, the history shows that the loss function is hardly changing at all. You are not likely to diverge from the loss surface, but it will take a very long time to converge. For the model with $\alpha = 1$, our model's validation loss goes to infinity. This is because our learning rate is too high and thus makes it very likely to diverge. You might also see behavior where the loss is not infinite, but rapidly jumps from one value to another. This is also undesirable behavior from the neural network. Finally, when we set $\alpha = 1 \times 10^{-3}$, we seem to get a very smooth loss profile. The validation loss goes down and then seems to almost converge to the final model. This is the behavior we would like to see. Ultimately what this exercise suggests is that having an appropriate learning rate can have a big impact on the overall performance of the model.

Multi-Layered Neural Networks

Thus far we have only worked with models that have a single layer. However, what happens when we add more layers? For our next exercise, add one more layer to the model and report the results. Namely, plot the loss profile, determine the final validation loss, and compare the results to the model with only one layer. When adding layers, keep the number of units the same and use a learning rate of 1×10^{-3}.

```
# Model with one additional layer
model = keras_model_sequential() %>%
  layer_dense(units = 32, activation = "relu",
              input_shape = dim(X_train)[2]) %>%
  layer_dense(units = 32, activation = "relu") %>%
  layer_dense(units = 1)

model %>% compile(
  loss = 'mse',
  optimizer = optimizer_sgd(lr = 1e-3)
)

res = model %>% fit(
  x = X_train, y = y_train, epochs = 50, verbose = 0, validation_split = 0.25,
  batch_size = 128
)
```

Let's compare the model with one layer versus two

```
paste("One layer loss:", res2$metrics$val_loss[50])
```

```
## [1] "One layer loss: 37.7241516113281"
```

```
paste("Two layer loss:", res$metrics$val_loss[50])
```

```
## [1] "Two layer loss: 18.0880699157715"
```

As you can see, adding additional layers can lead to performance improvements in your model, but it comes with the cost of additional computational time. For this dataset, that penalty is negligible, but for other datasets you might see, the difference can be non-trivial.

Number of Nodes

Another hyper-parameter that can be changed is the number of nodes or units to have for a particular layer of a neural network. For this exercise, using the single-layer architecture, fit a layer with 512 nodes. Evaluate this model both in and out-of-sample

```
# Model with 1024 units
model = keras_model_sequential() %>%
  layer_dense(units = 512, activation = "relu",
              input_shape = dim(X_train)[2]) %>%
  layer_dense(units = 1)

model %>% compile(
  loss = 'mse',
  optimizer = optimizer_sgd()
)

res = model %>% fit(
  x = X_train, y = y_train, epochs = 50, verbose = 0, validation_split = 0.25,
  batch_size = 128
)
```

```
model %>% evaluate(X_train, y_train)
```

```
##     loss
## 8.704157
```

```
model %>% evaluate(X_test, y_test)
```

```
##     loss
## 21.21895
```

That's a very large gap. This is a sign that we are *over-fitting* the data. How can we control this?

Dropout

Typically when working with models that have a large number of parameters, one way to ensure that the model does not over-fit to the data is by adding regularization. In the context of linear regression, this involves adding either an L_1 or L_2 penalty to the model to prevent the model from adding too many coefficients.

While this can be done with neural networks, a simpler approach is to use a technique known as drop-out. Basically what this does is randomly select units from the previous layer to remove. In the original paper they demonstrated that by using this technique, you can have significant improvements out-of-sample.

For our next exercise, let's try adding drop-out to the model we just created to see if we decrease the gap from the training to the test set.

```
model = keras_model_sequential() %>%
  layer_dense(units = 512, activation = "relu",
              input_shape = dim(X_train)[2]) %>%
  layer_dropout(rate = 0.7) %>%
  layer_dense(units = 1)

model %>% compile(
  loss = 'mse',
  optimizer = optimizer_sgd()
)

res = model %>% fit(
  x = X_train, y = y_train, epochs = 50, verbose = 0, validation_split = 0.25,
  batch_size = 128
)
```

```
model %>% evaluate(X_train, y_train)
```

```
##     loss
## 9.642704
```

```
model %>% evaluate(X_test, y_test)
```

```
##     loss
## 18.81391
```

In this portion of the lecture we have introduced the Keras API, introduced some key concepts for neural networks, and seen the effects of two key hyper-parameters: the learning rate and the number of layers. Now we're going to switch gears to another data set and also introduce some more advanced neural network concepts.