Lab 6

Emilio Dorigatti

2020-12-11

Welcome to the sixth lab.

In this lab, we will learn how to use Keras, a very popular and easy-to-use framework for deep learning. We will see what are the signs of overfitting, and how to avoid it using regularization. Then, we will analyze convergence of gradient descent on quadratic surfaces, and apply the intuition we gain on a practical example, comparing gradient descent with and without momentum.

Exercise 1

In this exercise, we will learn the basic usage of Keras, a popular deep learning library.

You can find help on the "R interface to Keras" webpage: https://keras.rstudio.com/, and through the integrated help system of RStudio. Use it to learn about the Keras functions we will be using!

The networks that we will use are small enough that they can run on your personal computer, but, if you need a GPU, you can try the R Kernel for Jupyter on Google Colab (click here). At the beginning of your session, get a GPU by clicking on "Runtime", then "Change runtime type", then choose "GPU" as hardware accelerator. Then, install Keras for R using `install.packages` before loading it with `library(keras)`.

Installation

Since Keras is a Python library, installing Keras in R is not as straightforward as installing normal R packages. Run the following commands:

```r
if(FALSE) {
install.packages("keras")
library(keras)
install_keras(
  method = "conda",
  tensorflow = "cpu"  # Change to "gpu" if you have a discrete GPU and want to use it
)
}
# If prompted to install conda, say yes.
```

Getting this to work can be a pain. Google is your friend, and do not hesitate to ask on Moodle and help your fellow coursemates there.

Loading and preparing the dataset

The dataset that we will be working with is the IMDB dataset, which is included in Keras. It contains 50,000 reviews that are highly polarized, that is, they are unambiguously either 'positive' or 'negative'. When the data is loaded, the training and test sets will contain 25,000 reviews each. In both sets, half of the reviews are positive and half are negative.

Each review is represented as a sequence of integers, where each integer corresponds to a specific word in a dictionary and the length of the sequence is the number of words in the review.

First, load the Keras package:

```
library(keras)
```

Then, load the dataset:

```
# The argument num_words = 10000 specifies that we only keep the top
# 10,000 most frequently occuring words in the (training) data (just
# for convenience)
imdb <- dataset_imdb(num_words = 10000)
c(c(train_data, train_labels), c(test_data, test_labels)) %<-% imdb
```

The variables `train_data` and `test_data` are lists of reviews, each review being a list of word indices (encoding a sequence of words). `train_labels` and `test_labels` are lists of 0s and 1s, where 0 stands for "negative" and 1 stands for "positive":

Here is how you can quickly decode one of these reviews back to English words:

```
# word_index is a dictionary mapping words to an integer index
word_index <- dataset_imdb_word_index()

# We reverse it, mapping integer indices to words
reverse_word_index <- names(word_index)
names(reverse_word_index) <- word_index

# We decode the review; note that our indices were offset by 3, because
# 0, 1 and 2 are reserved indices for "padding", "start of sequence", and "unknown".
decoded_review <- sapply(train_data[[1]], function(index) {
  word <- if (index >= 3) reverse_word_index[[as.character(index - 3)]]
  if (!is.null(word)) word else "?"
})

cat(decoded_review)
```

```
## ? this film was just brilliant casting location scenery story direction everyone's really suited
```

Before we feed the reviews to the network, we need to convert them from sequences of integers to "bag of words" vectors. For example, turning the sequence (3,5,9) into a 10 dimensional vector gives us (0,0,1,0,1,0,0,0,1,0) which has a 1 in the positions 3, 5 and 9 and zeros everywhere else.

```
vectorize_sequences <- function(sequences, dimension = 10000) {
  results <- matrix(0, nrow = length(sequences), ncol = dimension)
  for (i in 1:length(sequences))
    results[i, sequences[[i]]] <- 1
  results
}


# Transform the training and test data
x_train <- vectorize_sequences(train_data)
x_test <- vectorize_sequences(test_data)
```

Our data is now ready to be fed to a neural network.

Building the network

When deciding on an architecture for a neural network with fully connected layers, the two key things to consider are:

- The number of hidden layers in the network
- The number of neurons in each of the hidden layers

Increasing the depth of the network (that is, adding layers) or increasing the number of neurons in a given layer will increase the capacity of the network and allow it to learn more complex non-linear decision boundaries. However, making the network too large can lead to overfitting.

In practice, deciding on an architecture is an iterative process where many different networks are trained in order to find a good setting for the hyperparameters. For this exercise, however, we will use a simple feedforward neural network with two fully-connected hidden layers consisting of 16 neurons each, and a single output neuron which outputs the probability of the review being 'positive'.

Keras has two main APIs for building neural nets: Sequential and Functional, and we will mainly work with the Sequential API. The most important concepts in Keras are 'layers' and 'models':

- A layer represents a layer of the network, with all its neurons, their activations, regularization parameters, and so on. Although we have only seen fully connected layers, there are other types, too, such as convolutions and recurrent layers.
- A model contains all the layers of the network and their connections. It also contains information about the loss function and the optimizer.

Models are created with `keras_model_sequential`, and dense layers with `layer_dense`. Please read the documentation to know what parameters are accepted by these functions.

```
model <- keras_model_sequential() %>%
  layer_dense(
    units = 16, activation = "relu", input_shape = c(10000)
  ) %>%
  layer_dense(
    units = 16, activation = "relu"
  ) %>%
  layer_dense(
    units = 1, activation = "sigmoid"
  )
```

Compiling the model

So, we've built our model. Before we can train the network, however, we must specify:

1) The loss function to use (mean squared error, cross entropy, etc)
2) The optimizer (SGD, Adam, RMSProp, etc.)
3) Any metrics (such as accurary) used to measure the performance of the model

To do this, we call the `compile` function. The arguments passed to compile can be string, such as `"rmsprop"`, or the corresponding function instance, e.g. `optimizer_rmsprop`, which is included in the Keras package. The same works for losses and metrics.

```
model %>% compile(
  optimizer = "rmsprop",
  loss = "binary_crossentropy",
  metrics = c("accuracy")
)
```

Validation set

We are now ready to train the model. Before we do so, let's create a validation set to monitor the (generalization) performance of the model during training, by taking the first 10,000 samples of the training data and the corresponding labels. The new training set should contain only the remaining 15,000 samples.

```
val_indices <- 1:10000

x_val <- x_train[val_indices,]
partial_x_train <- x_train[-val_indices,]

y_val <- train_labels[val_indices]
partial_y_train <- train_labels[-val_indices]
```

Fit

We wish to train the network for 20 epochs with batches of size 512. Do not forget to specify a separate validation set to evaluate the network at the end of every epoch. Check the parameters on the documentation for `fit`.

```
history <- model %>% fit(
  partial_x_train,
  partial_y_train,
  epochs = 20,
  batch_size = 512,
  verbose = 0,
  validation_data = list(x_val, y_val)
)
```

Let's inspect the history object:

```
str(history)
```

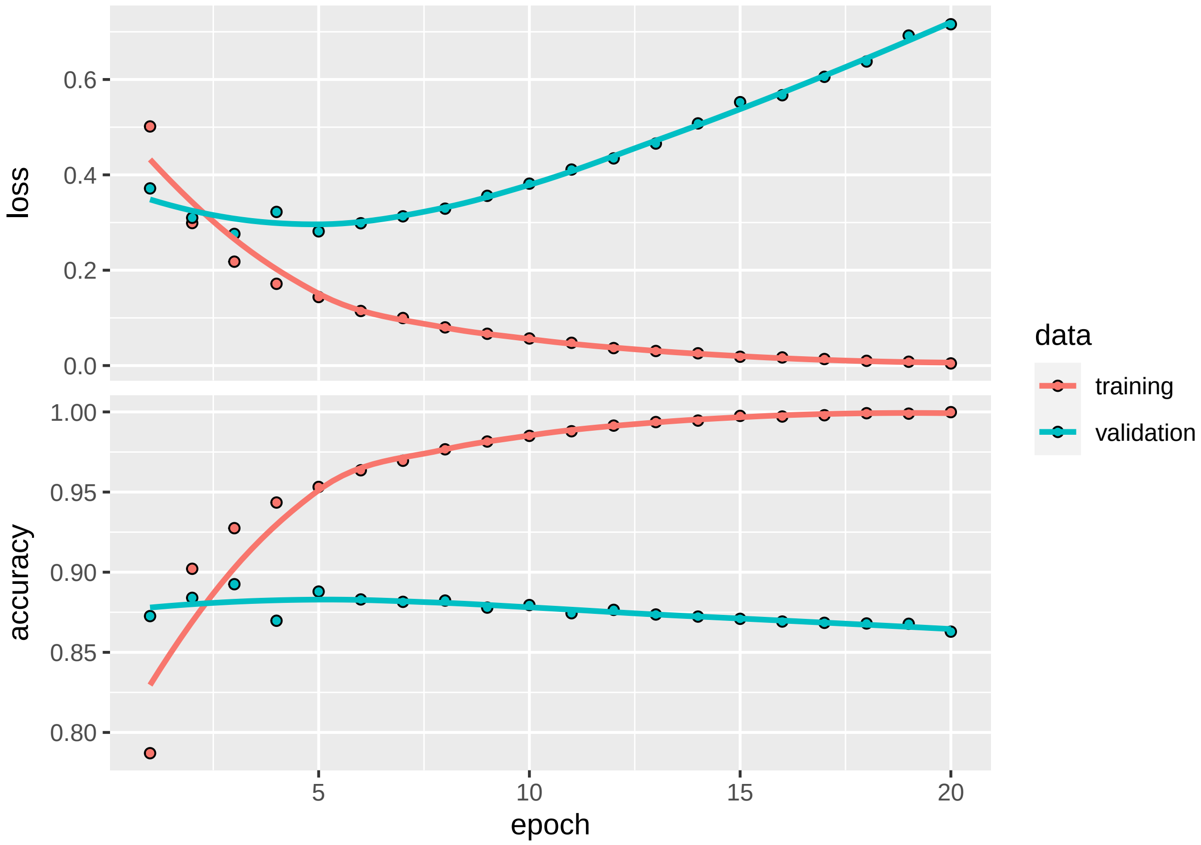
```
## List of 2
##  $ params :List of 3
##   ..$ verbose: int 0
##   ..$ epochs : int 20
##   ..$ steps  : int 30
##  $ metrics:List of 4
##   ..$ loss        : num [1:20] 0.502 0.299 0.218 0.171 0.144 ...
##   ..$ accuracy    : num [1:20] 0.787 0.902 0.927 0.943 0.953 ...
##   ..$ val_loss    : num [1:20] 0.372 0.31 0.276 0.322 0.281 ...
##   ..$ val_accuracy: num [1:20] 0.873 0.884 0.892 0.87 0.888 ...
##  - attr(*, "class")= chr "keras_training_history"
```

As you can see, it contains both the parameters passed to `fit` and the losses and metrics at the end of each epoch (`history$metrics`)

It's very convenient to plot the metrics by calling the `plot` function of the history object

```
plot(history)
```

```
## `geom_smooth()` using formula 'y ~ x'
```



As expected, the training loss decreases with each epoch (and training accuracy increases). However, the validation loss decreases initially and then begins to increase after the fourth epoch. Therefore, the network has overfit.

Evaluate

Let's evaluate the performance of the model on the test set by calling the `evaluate` function:

```
results = model %>% evaluate(
  x_test, test_labels, verbose=0
)
```

The performance on the test set is:

```
results
```

```
##      loss  accuracy
## 0.7927746 0.8477600
```

Our simple model does reasonably well. It achieves an accuracy of approximately 88%.

Predict

Finally, to generate the likelihood of the reviews being positive, we call the `predict` function:

```
model %>% predict(
  x_test[1:10,]
)
```

```
##                 [,1]
##  [1,] 3.421694e-03
##  [2,] 1.000000e+00
##  [3,] 8.538648e-01
##  [4,] 9.786876e-01
##  [5,] 9.973484e-01
##  [6,] 9.994779e-01
##  [7,] 5.740535e-01
##  [8,] 8.650247e-06
##  [9,] 9.504053e-01
## [10,] 1.000000e+00
```

Now play around with the code by adding and deleting layers, changing the hidden activation, optimizer, learning rate, batch-size, etc.

Conclusion

Here's what you should take away from this example:

- You usually need to do quite a bit of preprocessing on your raw data in order to be able to feed it – as tensors – into a neural network.
- Stacks of dense layers with `relu` activations can solve a wide range of problems (including sentiment classification), and you'll likely use them frequently.
- As they get better on their training data, neural networks eventually start *overfitting* and end up obtaining increasingly worse results on data they have never seen before. Be sure to always monitor performance on data that is outside of the training set.

Training a neural network in Keras always consists of the following steps:

1. Define the network architecture with `keras_model_sequential` and `layer_dense`.
2. Compile the network by defining the loss, the optimizer, and optional performance metrics with `compile`
3. Fit the network to the dataset with the method `fit`
4. Evaluate the network on the test data with `evaluate`
5. Use the network to make predictions with the `predict` method.

Exercise 2

In this exercise, we will look at a couple of different methods to regularize a neural network in order to prevent overfitting.

Plotting the validation loss is a simple way to determine whether the network has overfit. During the first few epochs of training, both the training loss and the validation loss tend to decrease in tandem but after a certain point, the validation loss starts to increase while the training loss continues to decrease. It is at this point that the network begins to overfit.

Training multiple networks

In order to get a feel for the overfitting behaviour of neural networks, we will train 3 different architectures and observe the training and validation losses.

Create the first model with two hidden layers, each with 16 units and relu activation.

```
original_model = (
  keras_model_sequential() %>%
    layer_dense(units = 16, activation = "relu", input_shape = c(10000)) %>%
    layer_dense(units = 16, activation = "relu") %>%
    layer_dense(units = 1, activation = "sigmoid")
)

compile(
  original_model,
  optimizer = "rmsprop",
  loss = "binary_crossentropy",
  metrics = c("accuracy")
)
```

Our second model will be similar to the first but it will be much smaller; reduce the number of neurons in the hidden layers from 16 to 4, and keep everything else unchanged.

```
smaller_model = (
  keras_model_sequential() %>%
    layer_dense(units = 4, activation = "relu", input_shape = c(10000)) %>%
    layer_dense(units = 4, activation = "relu") %>%
    layer_dense(units = 1, activation = "sigmoid")
)

compile(
  smaller_model,
  optimizer = "rmsprop",
  loss = "binary_crossentropy",
  metrics = c("accuracy")
)
```

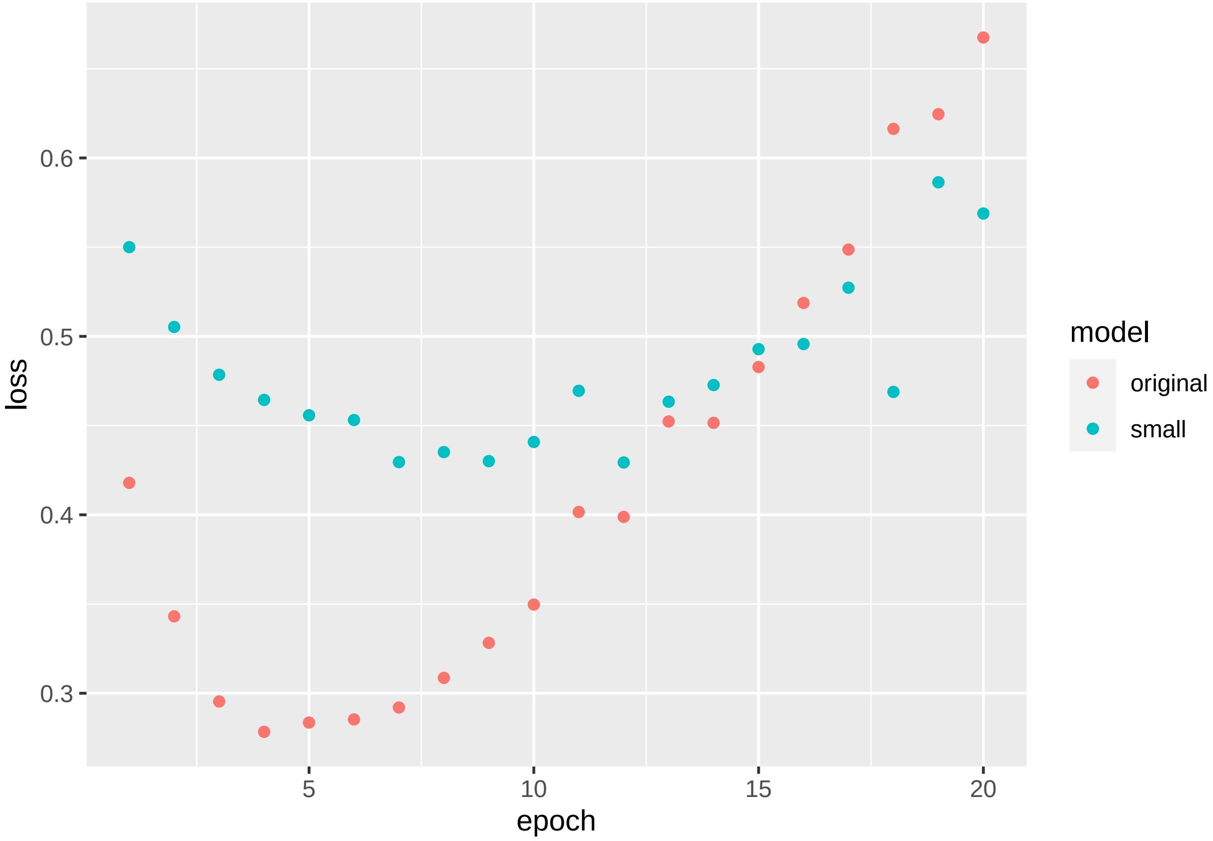
We now train both networks network using the `fit` function for 20 epochs using a batch size of 512; remember to use a separate validation dataset.

```
original_hist = fit(
  original_model,
  partial_x_train,
  partial_y_train,
  batch_size = 512,
  epochs = 20,
  validation_data = list(x_val, y_val),
  verbose = 0
)

smaller_model_hist = fit(
  smaller_model,
  partial_x_train,
  partial_y_train,
  batch_size = 512,
  epochs = 20,
  validation_data = list(x_val, y_val),
  verbose = 0
)
```

Ploting the losses

To compare the losses we will write an R function that takes a named list of loss series and plots it:

```
library(ggplot2)
library(tidyr)

plot_training_losses = function(losses) {
  loss_names = names(losses)
  losses = as.data.frame(losses)
  losses$epoch = seq_len(nrow(losses))
  losses %>%
    gather(model, loss, loss_names[[1]], loss_names[[2]]) %>%
    ggplot(aes(x = epoch, y = loss, colour = model)) +
    geom_point()
}

plot_training_losses(list(
  original=original_hist$metrics$val_loss,
  small=smaller_model_hist$metrics$val_loss
))
```



As you can see, the smaller network starts overfitting later than the original one and its performance degrades much more slowly once it starts overfitting.

Third model

Now we build a third neural network that is even bigger than the original network. If the previous plot is any indication, this new network should overfit even worse than the original model.

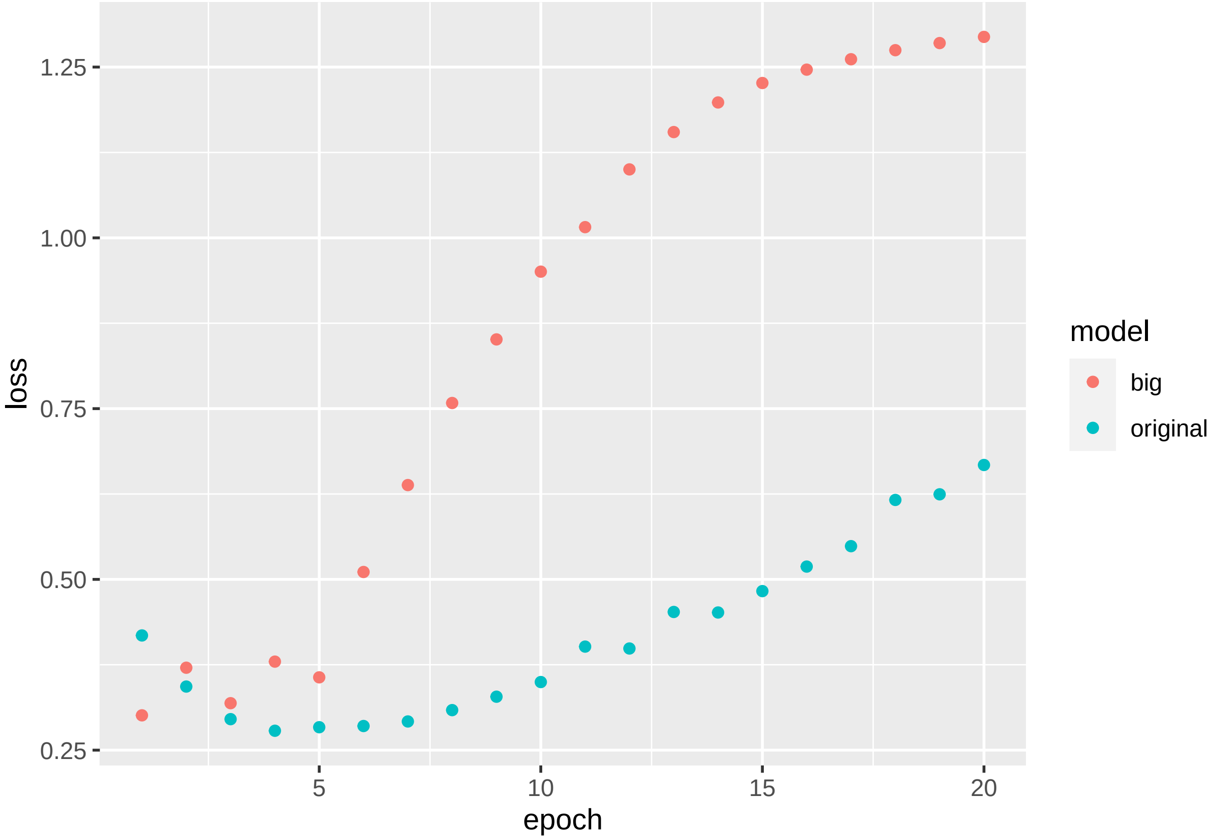
```
bigger_model = (
  keras_model_sequential() %>%
    layer_dense(512, activation = "relu", input_shape = c(10000)) %>%
    layer_dense(512, activation = "relu") %>%
    layer_dense(1, activation = "sigmoid")
)

compile(
  bigger_model,
  optimizer = "rmsprop",
  loss = "binary_crossentropy",
  metrics = c("accuracy")
)
```

Let's train this network:

```
bigger_model_hist = fit(
  bigger_model,
  partial_x_train,
  partial_y_train,
  batch_size = 512,
  epochs = 20,
  validation_data = list(x_val, y_val),
  verbose = 0
)
```

Here's how the bigger network fares compared to the reference one:

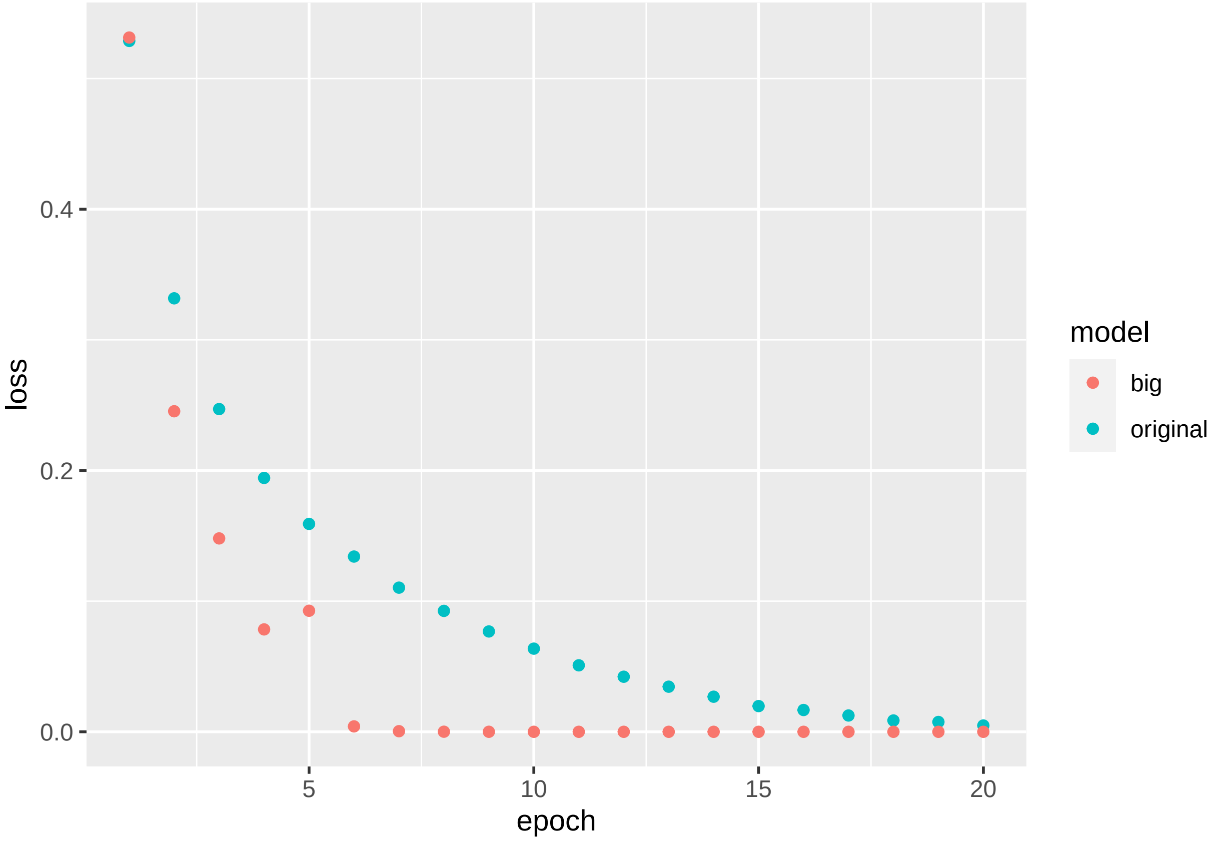
```
plot_training_losses(list(
  original=original_hist$metrics$val_loss,
  big=bigger_model_hist$metrics$val_loss
))
```



The bigger network starts overfitting almost right away, after just one epoch, and overfits much more severely. Its validation loss is also more noisy.

Let's plot the training losses:

```
plot_training_losses(list(
  original=original_hist$metrics$loss,
  big=bigger_model_hist$metrics$loss
))
```



As you can see, the bigger network gets its training loss near zero very quickly. The more capacity the network has, the quicker it will be able to model the training data (resulting in a low training loss), but the more susceptible it is to overfitting (resulting in a large difference between the training and validation loss).

Adding weight regularization

Regularizing a network in Keras is extremely easy. All you need to do is pass a weight regularizer object as the argument to the `kernel_regularizer` parameter of a layer. Keras has 3 built-in regularizers: `regularizer_l1` , `regularizer_l2` and `regularizer_l1_l2`, whose documentation is here.

Modify the original model and add weight decay with coefficient 0.001 to both hidden layers.

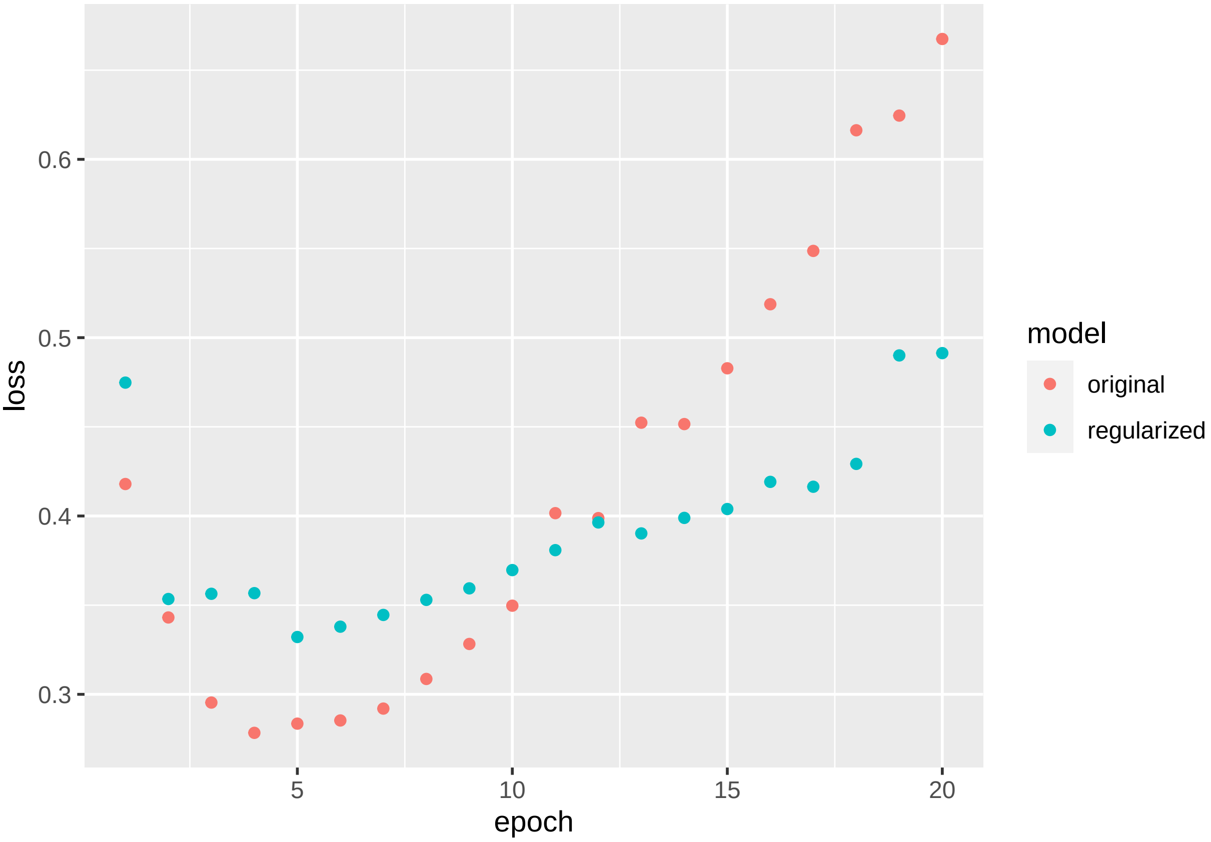
```
l2_model = (
  keras_model_sequential() %>%
    layer_dense(units = 16, activation = "relu", input_shape = c(10000),
                kernel_regularizer = regularizer_l2(0.001)) %>%
    layer_dense(units = 16, activation = "relu",
                kernel_regularizer = regularizer_l2(0.001)) %>%
    layer_dense(units = 1, activation = "sigmoid")
)

compile(
  l2_model,
  optimizer = "rmsprop",
  loss = "binary_crossentropy",
  metrics = c("accuracy")
)
```

Because the penalty term is only added during training (that is, it is not added when evaluating the model on the test set), the training error of this regularized model will be much higher.

```
l2_model_hist = fit(
  l2_model,
  partial_x_train, partial_y_train,
  batch_size = 512,
  epochs = 20,
  validation_data = list(x_val, y_val),
  verbose = 0
)
```

Now, plot the validation losses of the original model (in `original_hist`), and of its regularized counterpart (in `l2_model_hist`).

```
plot_training_losses(list(
  original=original_hist$metrics$val_loss,
  regularized=l2_model_hist$metrics$val_loss
))
```



As you can see, the regularized model does not overfit as much, even though both models have the same number of parameters. Feel free to play with the regularization strength to get a feel on how different settings affect learning. When is regularization preventing the network from learning anything at all? When is regularization so weak it does not make a difference?

Dropout regularization

Dropout is a very popular technique to regularize neural nets. It works by randomly turning off (or "dropping out") the input/hidden neurons in a network. This means that every neuron is trained on a different set of examples. Note that dropout is, in most cases, only used during training time. At test time, all units are used with their activations scaled down by the dropout rate to account for the fact that all neurons were used for the prediction. Normally, dropout is not applied to the inputs.

In Keras, dropout is implemented as its own separate layer: `layer_dropout`, that takes as input the probability to *drop* units. To apply dropout to a layer, place a `layer_dropout` after it while stacking layers.

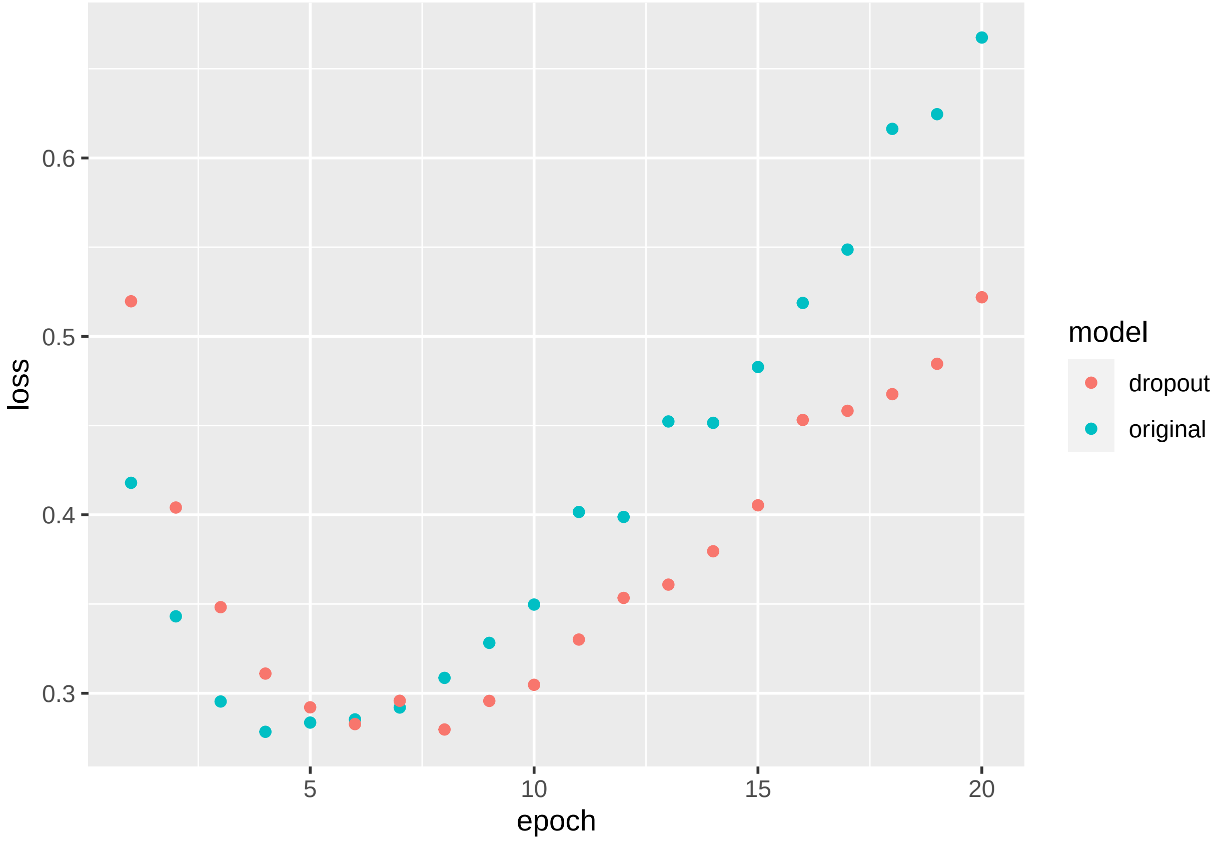
```
dpt_model = (
  keras_model_sequential() %>%
    layer_dense(units = 16, activation = "relu", input_shape = c(10000)) %>%
    layer_dropout(0.5) %>%
    layer_dense(units = 16, activation = "relu") %>%
    layer_dropout(0.5) %>%
    layer_dense(units = 1, activation = "sigmoid")
)

compile(
  dpt_model,
  optimizer = "rmsprop",
  loss = "binary_crossentropy",
  metrics = c("accuracy")
)
```

Let's train the model with dropout:

```
dpt_model_hist =  fit(
  dpt_model,
  partial_x_train, partial_y_train,
  batch_size = 512,
  epochs = 20,
  validation_data = list(x_val, y_val),
  verbose = 0
)
```

And compare it with the original model:

```
plot_training_losses(list(
  original=original_hist$metrics$val_loss,
  dropout=dpt_model_hist$metrics$val_loss
))
```



Once again, we see a marked improvement in the new model.

Early Stopping

Previously, we were training the network and checking *after training* when it started to overfit. But another very popular method to regularize a network is to stop training earlier than the specified number of epochs, by checking when the validation loss starts to increase.

Keras provides a way to do this automatically with the use of *callbacks*. A callback contains methods that are called at several points during the training process: at the beginning and end of each epoch, before and after feeding a new batch to the network, and at the beginning and end of training itself. Keras provides a few useful callbacks; read the documentation to find out what you can do with them. You can also write custom callbacks, more info on this here.

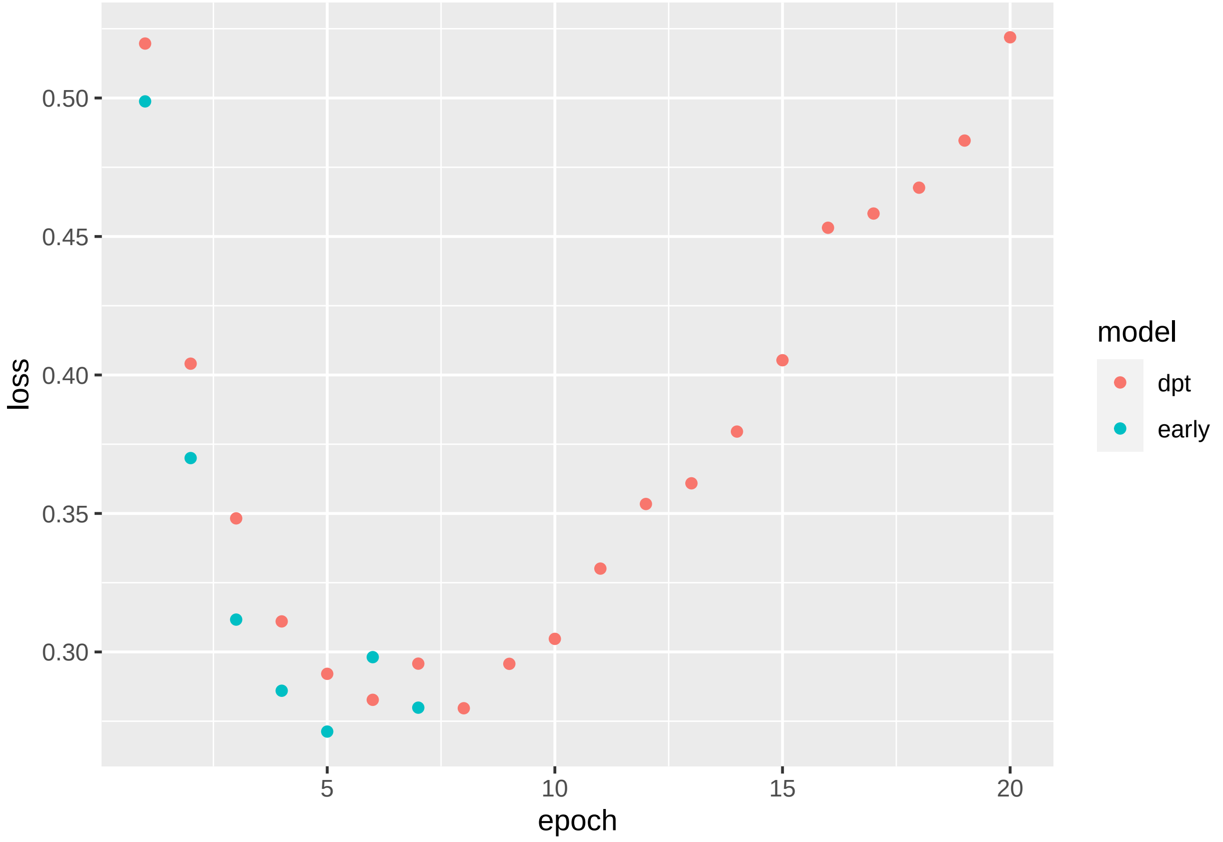
We will now re-train the dropout-regularized model, and make use of `callback_early_stopping` to interrupt training after the validation loss has stopped improving. In particular, the parameter `patience` indicates how many epochs to wait for an improvement of the validation loss. If there is no improvement for more than `patience` epochs, training is interrupted. Moreover, the setting the parameter `restore_best_weights` to `TRUE` will make sure that the network's parameters will be restored to the best performing on the validation set.

```
dpt_model_early = (
  keras_model_sequential() %>%
    layer_dense(units = 16, activation = "relu", input_shape = c(10000)) %>%
    layer_dropout(0.5) %>%
    layer_dense(units = 16, activation = "relu") %>%
    layer_dropout(0.5) %>%
    layer_dense(units = 1, activation = "sigmoid")
)

compile(
  dpt_model_early,
  optimizer = "rmsprop",
  loss = "binary_crossentropy",
  metrics = c("accuracy")
)

dpt_model_early_hist = fit(
  dpt_model_early,
  partial_x_train, partial_y_train,
  batch_size = 512,
  epochs = 20,
  validation_data = list(x_val, y_val),
  callbacks = list(
    callback_early_stopping(patience = 2, restore_best_weights = TRUE)
  ),
  verbose = 0
)
```

And the comparison:

```
# pad to the same length with NA's
pad = length(dpt_model_hist$metrics$val_loss)-length(dpt_model_early_hist$metrics$val_loss)
plot_training_losses(list(
  early=c(dpt_model_early_hist$metrics$val_loss, rep(NA, pad)),
  dpt=dpt_model_hist$metrics$val_loss
))
```

```
## Warning: Removed 13 rows containing missing values (geom_point).
```



As you can see, the early stopping callback worked, and the model was trained for only seven epochs. Now, evaluate this model on the test data:

```
dpt_model_early %>% evaluate(x_test, test_labels, verbose = 0)
```

```
##      loss  accuracy
## 0.2871268 0.8857600
```

As you can see, the loss is close to the lowest loss in the graph.

The take-home message for this exercise is: large neural networks can easily overfit, especially with small training sets. This means that the network learns spurious patterns that are present in the training data and, therefore, fails to generalize to unseen examples. In such a scenario, your options are:

1. Get more training data
2. Reduce the size of the network
3. Regularize the network

Exercise 3

Consider an error function of the form:

$$E = \frac{1}{2}\lambda_1 x_1^2 + \frac{1}{2}\lambda_2 x_2^2$$

With $\lambda_1 \geq 0$ and $\lambda_2 \geq 0$. First, show that the global minimum of E is at $x_1 = x_2 = 0$, then find the matrix $\mathbf{H}$ such that $E = 1/2 \cdot \mathbf{x}^T\mathbf{H}\mathbf{x}$. Show that the two eigenvectors $\mathbf{u}_1$ and $\mathbf{u}_2$ of this matrix are axis-aligned, and have λ_1 and λ_2 as eigenvalues.

Note that any vector $\mathbf{x}$ can be expressed as

$$\mathbf{x} = \sum_i \alpha_i \mathbf{u}_i$$

where α_i is the distance from the origin to $\mathbf{x}$ along the i-th axis (assuming the eigenvectors have unit length). Now find the gradient of E with respect to $\mathbf{x}$, and express it in terms of α_i, λ_i and $\mathbf{u}_i$.

Then, use this gradient to perform one step of gradient descent, i.e. compute

$$\mathbf{x}' = \mathbf{x} - \eta\nabla_{\mathbf{x}}E$$

And show that

$$\alpha_i' = (1 - \eta\lambda_i)\alpha_i$$

Which means that the distances from the origin to the current location evolve independently for each axis, and at every step the distance along the direction $\mathbf{u}_i$ is multiplied by $(1 - \eta\lambda_i)$. After T steps, we have

$$\alpha_i^{(T)} = (1 - \eta\lambda_i)^T \alpha_i^{(0)}$$

So that, as long as $|1 - \eta\lambda_i| < 1$ for every i, $\mathbf{x}^{(T)}$ converges to the origin as T goes to infinity.

Now, find the largest learning rate that guarantees convergence along all directions, and show that, when using this learning rate, the slowest direction of convergence is along the eigenvector with the smallest eigenvalue. Also show that the rate of convegence along this direction is:

$$\left(1 - 2\frac{\lambda_{\text{min}}}{\lambda_{\text{max}}}\right)$$

Where λ_{min} and λ_{max} are the smallest and largest eigenvalues of $\mathbf{H}$.

This exercise shows that the largest eigenvalue determines the maximum learning rate, and that the relationship between smallest and largest eigenvalues determines the speed of convergence. Note that the ratio $\lambda_{\text{max}}/\lambda_{\text{min}}$ is known as the *condition number* of $\mathbf{H}$, and plays an important role in numerical analysis. Matrices with large condition number make optimization algorithms slower and/or more imprecise.

Solution

Since both λ_1 and λ_2 are positive, and are multiplied with the squares of the coordinates, E is positive everywhere but at $x_1 = x_2 = 0$, where $E = 0$.

The matrix $\mathbf{H}$ is

$$\mathbf{H} = \begin{vmatrix} \lambda_1 & 0 \\ 0 & \lambda_2 \end{vmatrix}$$

So that

$$\begin{aligned} E &= \frac{1}{2}\mathbf{x}^T\mathbf{H}\mathbf{x} && (1) \\ &= \frac{1}{2}\sum_i\sum_j x_i x_j h_{ij} && (2) \\ &= \frac{1}{2}(x_1x_1h_{11} + x_1x_2h_{12} + x_2x_1h_{21} + x_2x_2h_{22}) && (3) \\ &= \frac{1}{2}\lambda_1 x_1^2 + \frac{1}{2}\lambda_2 x_2^2 && (4) \end{aligned}$$

To find the eigenvalues, solve $\det(\mathbf{H} - \beta_i\mathbf{I}) = 0$, i.e. $(\lambda_1 - \beta_1)(\lambda_2 - \beta_2) = 0$, which shows that the eigenvalues are λ_1 and λ_2. To find the corresponding eigenvectors, solve $\mathbf{Hx} = \lambda_1\mathbf{x}$, which gives:

$$\begin{vmatrix} \lambda_1 x_1 \\ \lambda_2 x_2 \end{vmatrix} = \begin{vmatrix} \lambda_1 x_1 \\ \lambda_1 x_2 \end{vmatrix}$$

Note that $\lambda_2 x_2 = \lambda_1 x_2$ is satisfied for $x_2 = 0$, while $\lambda_1 x_1 = \lambda_1 x_1$ is true for every x_1. We can choose $x_1 = 1$ and $x_2 = 0$ for the first eigenvector $\mathbf{u}_i$. A similar reasoning gives $x_1 = 0$ and $x_2 = 1$ for $\mathbf{u}_2$.

The gradient of E is

$$\nabla_{\mathbf{x}} E = \begin{vmatrix} \lambda_1 x_1 \\ \lambda_2 x_2 \end{vmatrix} = \sum_i \lambda_i \alpha_i \mathbf{u}_i$$

Performing one step of gradient descent gives

$$\mathbf{x}' = \mathbf{x} - \eta\nabla_{\mathbf{x}} E = \sum_i \alpha_i\mathbf{u}_i - \eta\sum_i \lambda_i\alpha_i\mathbf{u}_i = \sum_i \alpha_i(1 - \eta\lambda_i)\mathbf{u}_i = \sum_i \alpha_i'\mathbf{u}_i$$

Now we want $|1 - \eta\lambda_i| < 1$ for every i. We solve this inequality to find admissible values for η:

$$\begin{aligned} |1 - \eta\lambda_i| < 1 &\Leftrightarrow -1 < 1 - \eta\lambda_i < 1 && (5) \\ &\Leftrightarrow -2 < -\eta\lambda_1 < 0 && (6) \\ &\Leftrightarrow -\frac{2}{\lambda_i} < -\eta < 0 && (7) \\ &\Leftrightarrow 0 < \eta < \frac{2}{\lambda_i} && (8) \end{aligned}$$

This must be true for every eigenvalue, which means that the learning rate must be smaller than $2/\lambda_{\max}$. We now use this upper bound in the update equation for α_i:

$$\alpha_i' = (1 - \eta\lambda_i)\alpha_i = \left(1 - 2\frac{\lambda_{\mathrm{i}}}{\lambda_{\max}}\right)\alpha_i$$

From this, you can make two observations:

1. The smallest rate of convergence occurs for $\lambda_i = \lambda_{\min}$, i.e., along the direction with the smallest eigenvalue, and
2. When $\eta = 2/\lambda_{\max}$, $\alpha'_{\max} = -\alpha_{\max}$, i.e., gradient descent will perpetually oscillate between $\alpha_{\max}\mathbf{u}_{\max}$ and $-\alpha_{\max}\mathbf{u}_{\max}$, neither converging nor diverging.

Exercise 4

In this exercise we play a bit with the quadratic error surfaces that we analyzed in the previous exercise. We will apply apply the insights we got, and test different forms of gradient descent. The purpose is to get an intuitive feeling for how these things work, and for this some playful interaction is required from your side.

Remember that the error function was:

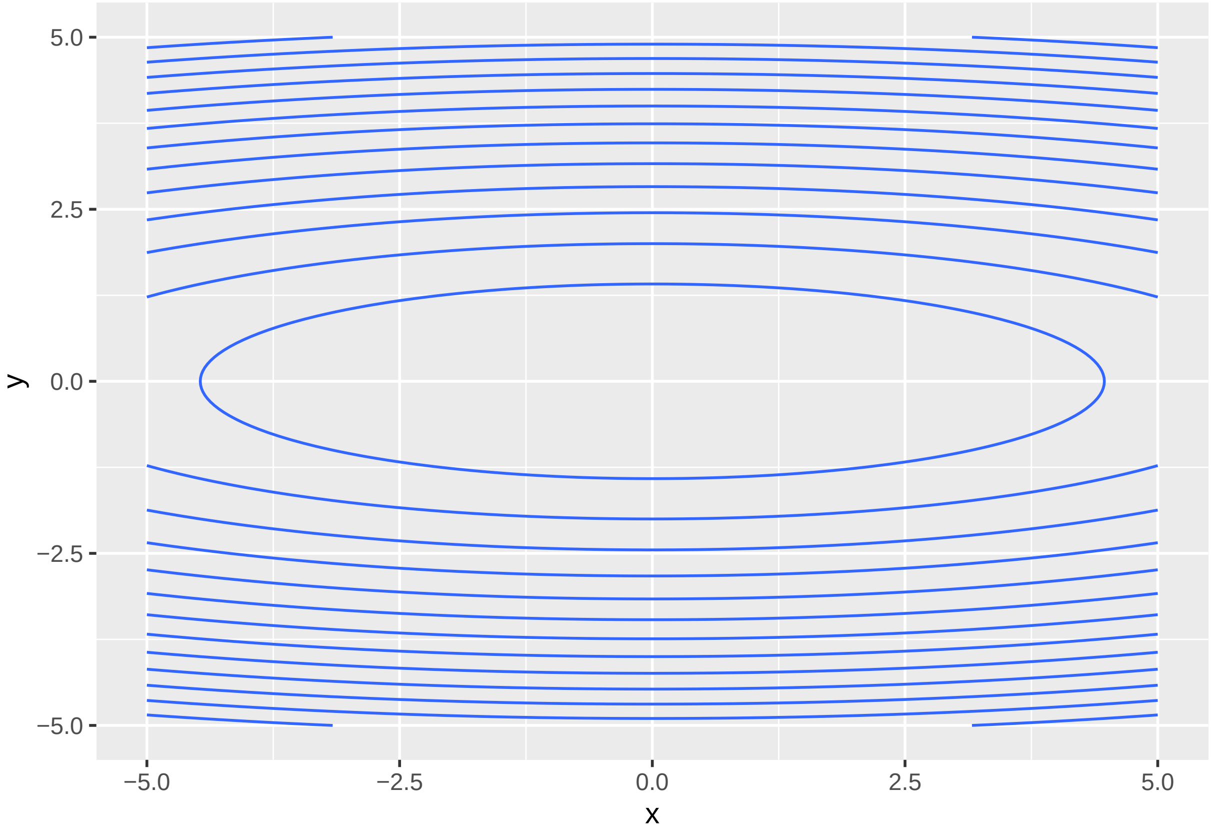
$$E = \frac{1}{2}\lambda_1 x_1^2 + \frac{1}{2}\lambda_2 x_2^2$$

We first create functions to compute E and its gradient:

```
lambda1 = 1
lambda2 = 10

make_function = function(lambda1, lambda2) {
  function(p) {
    lambda1 * p[1]^2 / 2 + lambda2 * p[2]^2 / 2
  }
}

func = make_function(lambda1 = 1, lambda2 = 10)


make_gradient = function(lambda1, lambda2) {
  function(p) c(
    lambda1 * p[1], lambda2 * p[2]
  )
}

grad = make_gradient(lambda1 = 1, lambda2 = 10)
```

And we visualize a contour plot of the surface:

```
make_contours = function(lambda1, lambda2) {
  halfres = 100
  data = as.matrix(expand.grid(-halfres:halfres,-halfres:halfres)) / halfres * 5
  data.frame(x = data[,1], y = data[,2], z = apply(data, 1, func))
}

contours = make_contours(lambda1 = 1, lambda2 = 10)

library(ggplot2)
ggplot(contours) +
  geom_contour(aes(x, y, z=z))
```

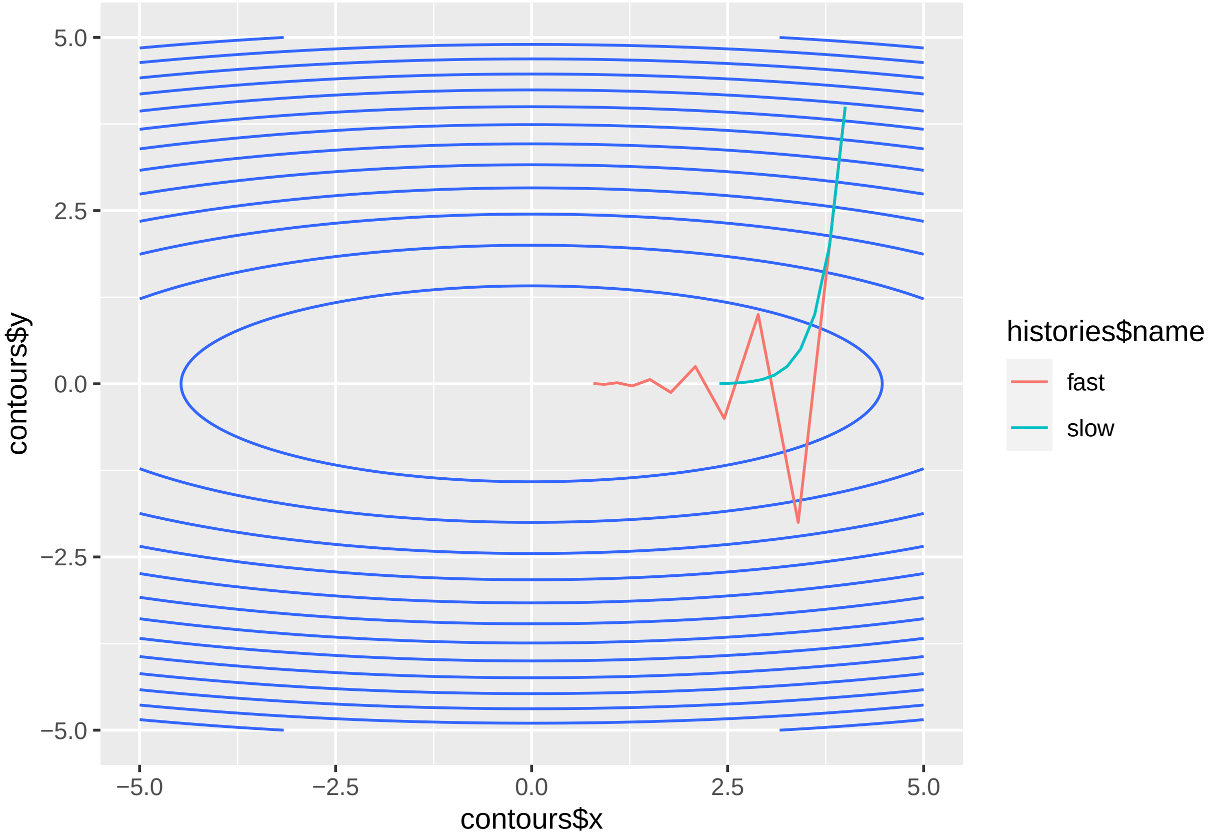


We now create a vanilla gradient descent optimizer that returns all points visited during the process:

```
gradient_descent_optimizer = function(x0, max_steps, lrate) {
  point = x0
  sapply(1:(max_steps + 1), function(i) {
    old = point
    point <<- point - lrate * grad(point)
    old
  })
}

hist_slow = gradient_descent_optimizer(c(4, 4), 10, 0.05)
hist_fast = gradient_descent_optimizer(c(4, 4), 10, 0.15)
```

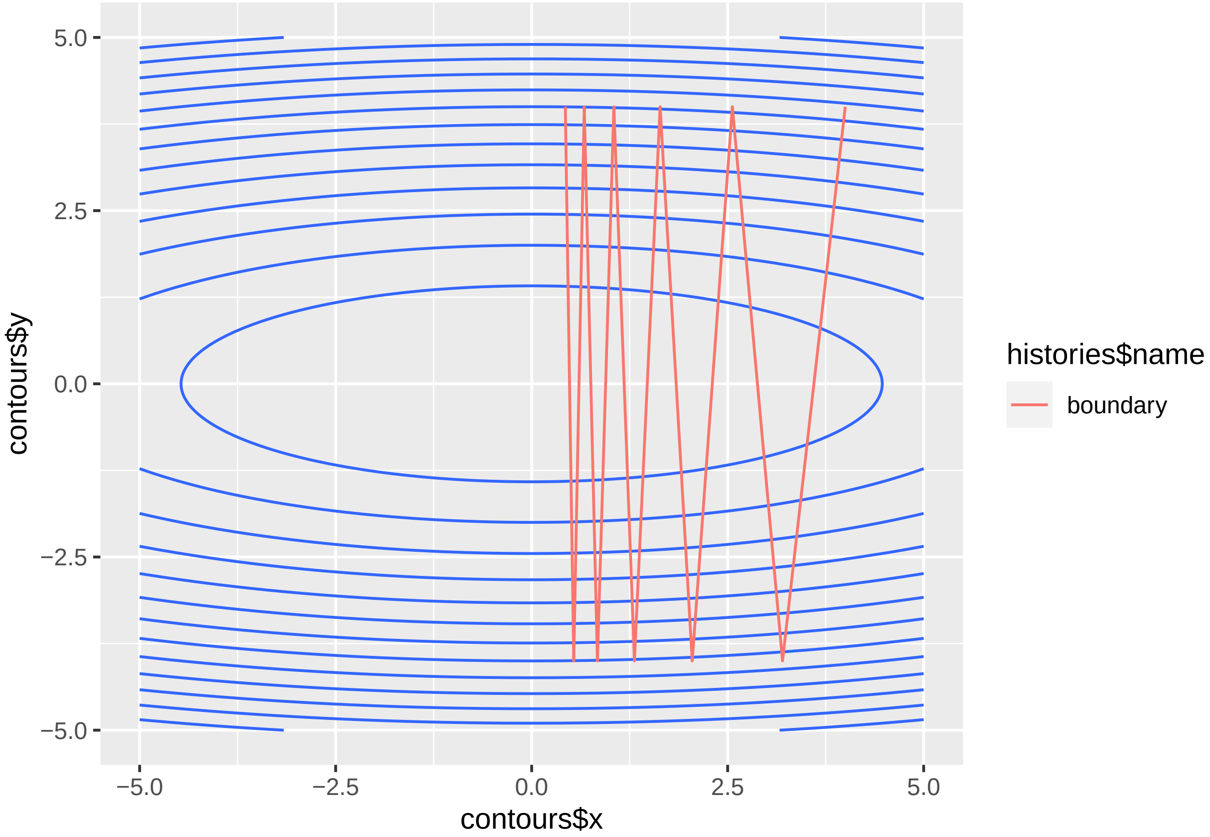
And a function that plots several traces together, so that we can compare them:

```
plot_histories = function(histories) {
  histories = do.call("rbind", lapply(names(histories), function(name) {
    hh = get(name, histories)
    data.frame(x = hh[1,], y = hh[2,], name = rep(name, length(hh)))
  }))

  ggplot() +
    geom_contour(aes(contours$x, contours$y, z = contours$z)) +
    geom_line(aes(histories$x, histories$y, colour = histories$name))
}

plot_histories(list(
  fast = hist_fast,
  slow = hist_slow
))
```



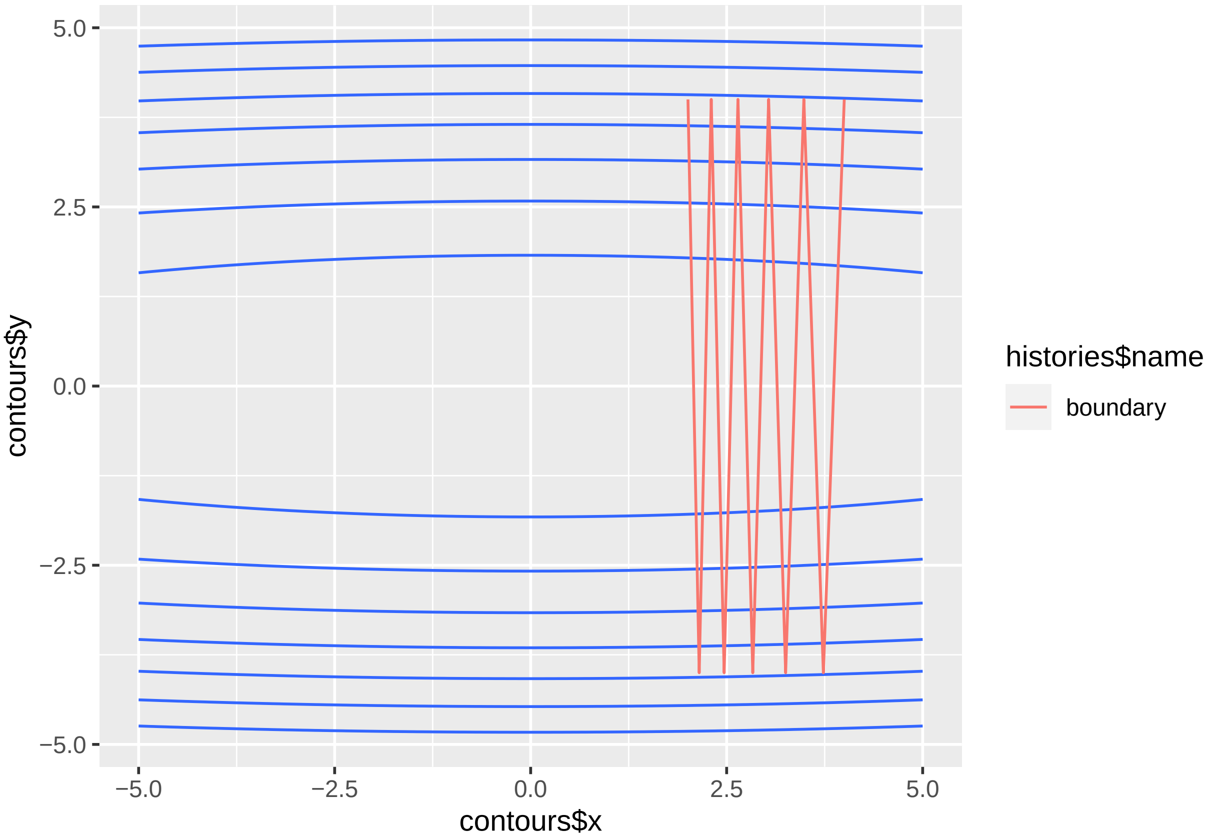
Now, recall from the previous exercise that the learning rate cannot be larger than $2\lambda_{\text{min}}/\lambda_{\text{max}}$. Compute this upper bound for the example here, use it to optimize the error starting from $\mathbf{x} = |4, 4|^T$, and plot the resulting trajectory. What can you notice? Try to slightly reduce it and increase it, and verify that when it is larger than the upper bound, the procedure diverges.

```
max_learning_rate = (
  2 / max(lambda1, lambda2)
)

plot_histories(list(
  boundary = gradient_descent_optimizer(c(4, 4), 10, max_learning_rate)
))
```



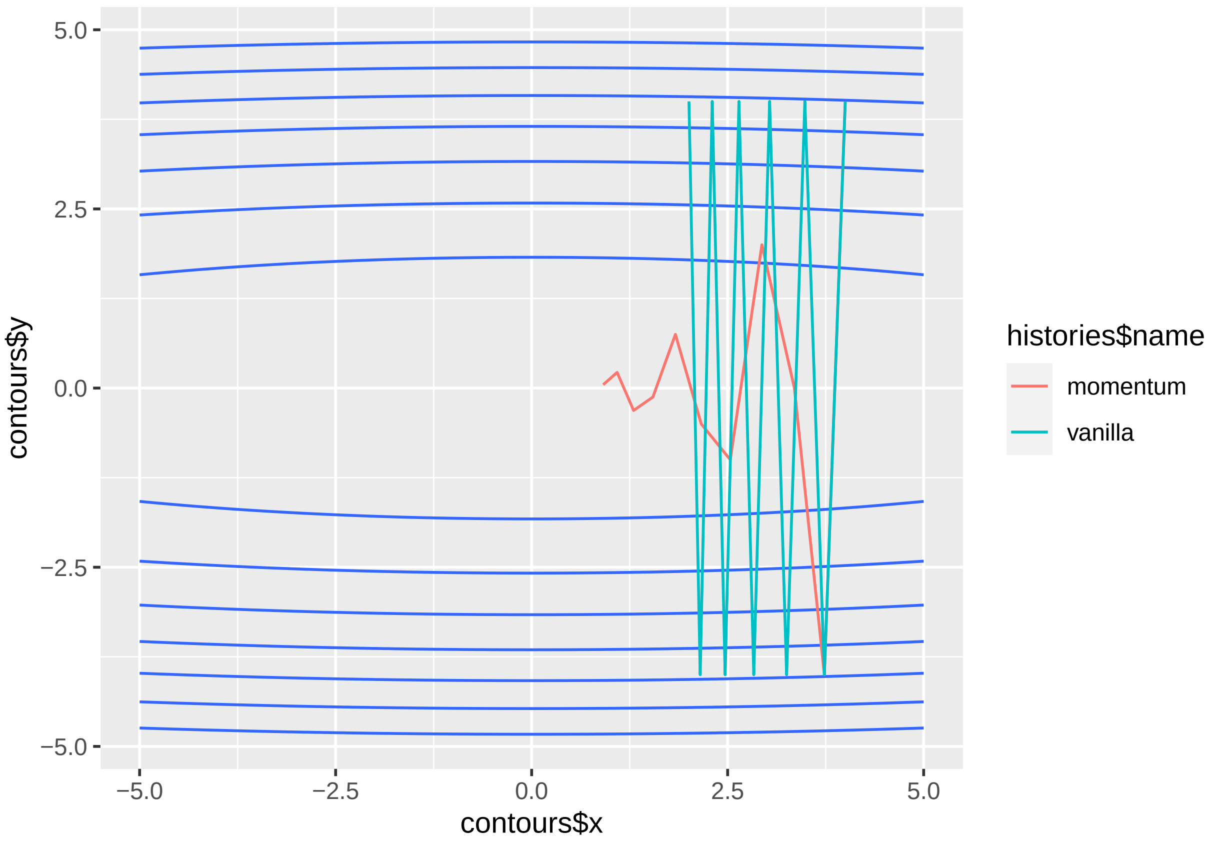
Now try to change the eigenvalues so as to increase the condition number, and verify that convergence becomes slower as the condition number increases:

```
func = make_function(lambda1 = 1, lambda2 = 30)
grad = make_gradient(lambda1 = 1, lambda2 = 30)
contours = make_contours(lambda1 = 1, lambda2 = 30)

max_learning_rate = (
  2 / 30
)

sgd_history = gradient_descent_optimizer(c(4, 4), 10, max_learning_rate)

plot_histories(list(
  boundary = sgd_history
))
```



Finally, modify the optimizer to use momentum, and verify that convergence becomes faster:

```
momentum_optimizer = function(x0, max_steps, lrate, momentum) {
  point = x0
  velocity = c(0, 0)
  history = sapply(1:(max_steps + 1), function(i) {
    old = point
    velocity <<- momentum * velocity - lrate * grad(point)
    point <<- point + velocity
    old
  })
}

momentum = (
  0.5
)
momentum_history = momentum_optimizer(c(4, 4), 10, max_learning_rate, momentum)


plot_histories(list(
  vanilla = sgd_history,
  momentum = momentum_history
))
```



Now explore the convergence behavior as momentum and learning rate change. Does momentum bring any improvement when the condition number is one (i.e. the eigenvalues are identical)?