Lab 5

Welcome to the fifth lab. In this lab, we will learn how to use Keras to build and train neural networks. 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!

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)`.

Exercise 1 : Binary Classification

This exercise is a binary classification task. The goal is to train a neural network in Keras to classify movie reviews as either 'positive' or 'negative' reflecting the sentiment of the reviewers.

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, 50% of the reviews will be 'postive'(and 50% will be '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, we load the Keras package

```r
library(keras)
```

Then, we load the dataset.

```r
# 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":

```r
str(train_data[[1]])
```

```r
train_labels[[1]]
```

Since we restricted ourselves to the top 10,000 most frequent words, no word index will exceed 10,000:

```r
max(sapply(train_data, max))
```

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

```r
# 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".
```

```r
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)
```

Preparing the data

Before we feed the reviews to the network, we first 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.

```r
vectorize_sequences <- function(sequences, dimension = 10000) {
  # TODO encode each sequence to a bag-of-words vector of the given dimension,
  # and set them as rows of a matrix
}



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

Here's a sample from the transformed data

```r
str(x_train[1,])
```

Finally, we convert the 'integer' labels to 'numeric' labels

```r
y_train <- as.numeric(train_labels)
y_test <- as.numeric(test_labels)
```

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 a scalar value (between 0 and 1) representing the probability of the review being 'positive'.

Keras has two main APIs for building neural nets: Sequential and Functional. In this exercise, we'll work with the Sequential API. The most important concepts in Keras are 'layers' and 'models'. As you'd expect, a layer in Keras reprents a layer in the neural network and a model represents the entire network. Keras has fully connected layers, convolutional layers, pooling layers, LSTMs, etc. To build a neural net using the Sequential API, you essentially "stack" a number these layers sequentially.

Models are created with `keras_model_sequential`, and dense layers with `layer_dense`.

```r
model <- keras_model_sequential() %>%
    layer_dense(
      # TODO parameters for the first layer: 16 units and relu activation.
      # Mind the `input_shape` parameter!
    ) %>%
    layer_dense(
      # TODO parameters for the second layer: again, 16 units and relu activation.
    ) %>%
    layer_dense(
      # TODO parameters for the output layer: a single unit with simoid activation
    )
```

Compile

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. This will be covered in detail in a later lecture.)
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.

```r
model %>% compile(
  # TODO use the 'rmsprop' optimizer, the appropriate loss for binary classification,
  # and the monitor the accuracy via the 'metrics' parameter
)
```

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.

```r
# TODO create training and validation sets
```

Fit

Let's train the model that we've compiled by calling the `fit` function.

Recall that a neural net is trained iteratively on 'batches' of the training data. One full sweep over the training set is called an 'epoch'. The corresponding parameters of `fit` are `batch_size` and `epochs`. Training a neural network on the entire dataset at the same time is computationally not feasible, and training it for too many epochs can result in overfitting. Therefore, it is very important to choose the right values for these parameters. `fit` returns an object which contains the values of the losses and metrics at the end of every epoch. If a validation set is passed, the corresponding values on the validation set are also returned.

We now 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`.

```r
history <- model %>% fit(
  # TODO add the parameters
)
```

Let's inspect the history object:

```
str(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)
```

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.

An easy fix is to automatically stop training once the validation accurary starts to increase. We will look at a simple way to do this later using 'callback' objects in Keras. For now, let's just train a new network for just 4 epochs.

```
# TODO build, compile and train a neural network as you just did, but use only
# four epochs and train on the entire training set `x_train` and `x_test`
```

Evaluate

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

```
results = model %>% evaluate(
  # TODO pass the test data to `evaluate`
)
```

The performance on the test set is:

```
results
```

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(
  # TODO pass the first ten test reviews
)
```

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. Sequences of words can be encoded as binary vectors, but there are other encoding options, too.
- Stacks of dense layers with `relu` activations can solve a wide range of problems (including sentiment classification), and you'll likely use them frequently.
- In a binary classification problem (two output classes), your network should end with a dense layer with one unit and a `sigmoid` activation. That is, the output of your network should be a scalar between 0 and 1, encoding a "confidence" (it is not a true probability unless you calibrate the scores).
- The corresponding loss function is `binary_crossentropy`.

- 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 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.

In future lab sessions, we'll build more complex models and add some bells and whistles but the general workflow will be exactly the same.

Exercise 2 : Multi-class Classification

In the previous exercise, we saw how to classify vector inputs into two mutually exclusive classes using a densely-connected neural network. But what happens when you have more than two classes?

In this exercise, we will build a network to classify Reuters newswires into 46 different mutually-exclusive topics. Since we have many classes, this problem is an instance of "multi-class classification", and since each data point should be classified into only one category, the problem is more specifically an instance of "single-label, multi-class classification". If each data point could have belonged to multiple categories (in our case, topics) then we would be facing a "multi-label, multi-class classification" problem.

The Reuters dataset

We will be working with the *Reuters dataset*, a set of short newswires and their topics, published by Reuters in 1986. It's a very simple, widely used toy dataset for text classification. There are 46 different topics; some topics are more represented than others, but each topic has at least 10 examples in the training set.

Like IMDB and MNIST, the Reuters dataset comes packaged as part of Keras. Let's take a look right away:

```r
library(keras)

reuters <- dataset_reuters(num_words = 10000)
c(c(train_data, train_labels), c(test_data, test_labels)) %<-% reuters
```

Like with the IMDB dataset, the argument `num_words = 10000` restricts the data to the 10,000 most frequently occurring words found in the data.

We have 8,982 training examples and 2,246 test examples:

```r
length(train_data)
```

```r
length(test_data)
```

As with the IMDB reviews, each example is a list of integers (word indices):

```r
train_data[[1]]
```

We can decode the reviews as in the previous exercise:

```r
word_index <- dataset_reuters_word_index()
reverse_word_index <- names(word_index)
names(reverse_word_index) <- word_index
decoded_newswire <- sapply(train_data[[1]], function(index) {
  # Note that our indices were offset by 3 because 0, 1, and 2
  # are reserved indices for "padding", "start of sequence", and "unknown".
  word <- if (index >= 3) reverse_word_index[[as.character(index - 3)]]
  if (!is.null(word)) word else "?"
})
```

```
cat(decoded_newswire)
```

The label associated with an example is an integer between 0 and 45: a topic index.

```
train_labels[[1]]
```

Preparing the data

We can vectorize the data with the exact same code as in our previous example:

```
x_train <- vectorize_sequences(train_data)
x_test <- vectorize_sequences(test_data)
```

To vectorize the labels, there are two possibilities: we could just cast the label list as an integer tensor, or we could use a "one-hot" encoding. One-hot encoding is a widely used format for categorical data, also called "categorical encoding". One-hot encoding of our labels consists in embedding each label as an all-zero vector with a 1 in the place of the label index. Note that there is a built-in way to do this in Keras:

```
one_hot_train_labels <- to_categorical(train_labels)
one_hot_test_labels <- to_categorical(test_labels)
```

Building our network

This topic classification problem looks very similar to our previous movie review classification problem: in both cases, we are trying to classify short snippets of text. There is however a new constraint here: the number of output classes has gone from 2 to 46, i.e. the dimensionality of the output space is much larger.

In a stack of dense layers like what we were using, each layer can only access information present in the output of the previous layer. If one layer drops some information relevant to the classification problem, this information can never be recovered by later layers: each layer can potentially become an "information bottleneck". In our previous example, we were using 16-dimensional intermediate layers, but a 16-dimensional space may be too limited to learn to separate 46 different classes: such small layers may act as information bottlenecks, permanently dropping relevant information.

For this reason we will use two larger hidden layers of 64 units with relu activation. For multi-class problems, the activation of the last layer should be the `"softmax"`.

```
# TODO build a model according to the specifications above
```

The loss function to use in this case is `"categorical_crossentropy"`.

```
# TODO compile the model; use the accuracy metric
```

Validating our approach

Let's set apart 1,000 samples in our training data to use as a validation set:

```
val_indices <- 1:1000

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

y_val <- one_hot_train_labels[val_indices,]
partial_y_train = one_hot_train_labels[-val_indices,]
```

Now let's train our network for 20 epochs, with batch size of 512.

```
# TODO train the model for 20 epochs with batches of 512 samples.
# Remember to validate the model at each epoch
```

Let's display its loss and accuracy curves:

```
plot(history)
```

The network begins to overfit after eight or nine epochs. Let's train a new network from scratch for nine epochs on the complete training set, then evaluate it on the test set.

```
# TODO re-create the network

# TODO train the network on the complete training set
# with the same settings you used previously

# TODO evaluate the network on the test set
```

```
results
```

Our approach reaches an accuracy of ~78%. With a balanced binary classification problem, the accuracy reached by a purely random classifier would be 50%, but in our case it is closer to 19%, so our results seem pretty good, at least when compared to a random baseline:

```
test_labels_copy <- test_labels
test_labels_copy <- sample(test_labels_copy)
length(which(test_labels == test_labels_copy)) / length(test_labels)
```

Generating predictions on new data

We can verify that the `predict` method of our model instance returns a probability distribution over all 46 topics. Let's generate topic predictions for all of the test data:

```
predictions <- (
  # TODO predict the samples in the test set
)
```

Each entry in `predictions` is a vector of length 46:

```
dim(predictions)
```

The coefficients in this vector sum to 1:

```
sum(predictions[1,])
```

The largest entry is the predicted class, i.e. the class with the highest probability:

```
which.max(predictions[1,])
```

A different way to handle the labels and the loss

We mentioned earlier that another way to encode the labels would be to preserve their integer values. If you do this, you should use a different loss function, `loss_sparse_categorical_crossentropy`. This new loss function is mathematically the same as `loss_categorical_crossentropy`, but it is computed in a different way so as to handle the integer labels.

On the importance of having sufficiently large intermediate layers

We mentioned earlier that since our final outputs were 46-dimensional, we should avoid intermediate layers with much less than 46 hidden units. Now let's try to see what happens when we introduce an information bottleneck by having intermediate layers significantly less than 46-dimensional, e.g. 4-dimensional.

```
# TODO create a network as you did above, but use 4 units in each hidden layer

# TODO train the network for 20 epochs with batches of 512 examples,
# and measure the performance on the validation set at every epoch

plot(history)
```

Our network now seems to peak at ~70% test accuracy, a 9% absolute drop. This drop is mostly due to the fact that we are now trying to compress a lot of information (enough information to recover the separation hyperplanes of 46 classes) into an intermediate space that is too low-dimensional. The network is able to cram *most* of the necessary information into these 8-dimensional representations, but not all of it.

Further experiments

- Try using larger or smaller layers: 32 units, 128 units, etc.
- We were using two hidden layers. Now try to use a single hidden layer, or three hidden layers.
- Try to use the sparse cross entropu loss.

Wrapping up

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

- If you are trying to classify data points between N classes, your network should end with a dense layer of size N.
- In a single-label, multi-class classification problem, your network should end with a `softmax` activation, so that it will output a probability distribution over the N output classes.
- There are two ways to handle labels in multi-class classification:
  - Encoding the labels via "categorical encoding" (also known as "one-hot encoding") and using `categorical_crossentropy` as your loss function.
  - Encoding the labels as integers and using the `sparse_categorical_crossentropy` loss function.
- If you need to classify data into a large number of categories, then you should avoid creating information bottlenecks in your network by having intermediate layers that are too small. Other tricks can be used when dealing with tens or thousands of categories, namely *negative sampling* and *hierarchical softmax*, but they are out of scope for this course.

Exercise 3

1. Show that, for binary classification, the binary and categorical cross entropy losses are equivalent (obviously, the former trained on 0/1 labels and sigmoid output, and the latter using one-hot encoding and softmax output with two units).

2. Let $\mathbf{z}$ be the output of the second-to-last layer of a neural network for multi-class classification, and assume that its distribution conditioned on a certain class C_k belongs to the exponential family, i.e.

$$p(\mathbf{z}|C_k) = \exp\left(A(\mathbf{w}_k) + B(\mathbf{z}, \phi) + \mathbf{w}_k^T \mathbf{z}\right)$$

   with $\mathbf{w}_k$ denoting the incoming weights of the neuron of the output layer that predicts the score of class k. Use Bayes' theorem to obtain $p(C_k|\mathbf{z})$, and show that it can be written as the output of an affine transformation followed by a softmax activation.