Deep Learning Lecture 1 - IMDB: Densily connected NN

MA8701 General Statistical Methods

Thiago G. Martins, Department of Mathematical Sciences, NTNU

Spring 2019

- The IMDB dataset
  - Preparing the data
    - Turning sequence of integers back to english
    - Turning sequence of integers to tensor format
  - Model definition
  - Model compilation
  - Validating your approach
  - Predicting on new data
  - Fighting overfitting
    - Reducing the network's size
    - Adding weight regularization
    - Adding dropout

The IMDB dataset

The objective here is to classify a movie review as either positive or negative.

Preparing the data

The data has already been preprocessed: the reviews (sequences of words) have been turned into sequences of integers, where each integer stands for a specific word in a dictionary.

- The argument `num_words = 10000` keep only the top 10,000 most frequently occurring words in the training data.

```
imdb <- dataset_imdb(num_words = 10000)

train_data <- imdb$train$x
train_labels <- imdb$train$y
test_data <- imdb$test$x
test_labels <- imdb$test$y
```

- Each review is a list of word indices.
- The labels are lists of 0s and 1s, where 0 stands for negative and 1 stands for positive.

- The first review in the list:

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

```
##  int [1:218] 1 14 22 16 43 530 973 1622 1385 65 ...
```

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

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

Turning sequence of integers back to english

Below is the code to turn the reviews from sequence of integers back to english.

```
word_index <- dataset_imdb_word_index()
reverse_word_index <- names(word_index)
names(reverse_word_index) <- word_index
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 "?"
})
```

Turning sequence of integers to tensor format

- The `vectorize_sequences` below will produce a tensor of rank 2 of the form `(samples, features)`
- Each sample is represented by a feature vector of the size of the dictionary being used with values equal to 1 if a particular word is present and 0 if the particular word is absent.

```
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
}

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

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

Model definition

```
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")
```

Model compilation

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

Validating your approach

Create a validation set by setting apart 10,000 samples from the original training data.

```
val_indices <- 1:10000

x_val <- x_train[val_indices,]
partial_x_train <- x_train[-val_indices,]
y_val <- y_train[val_indices]
partial_y_train <- y_train[-val_indices]
```

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

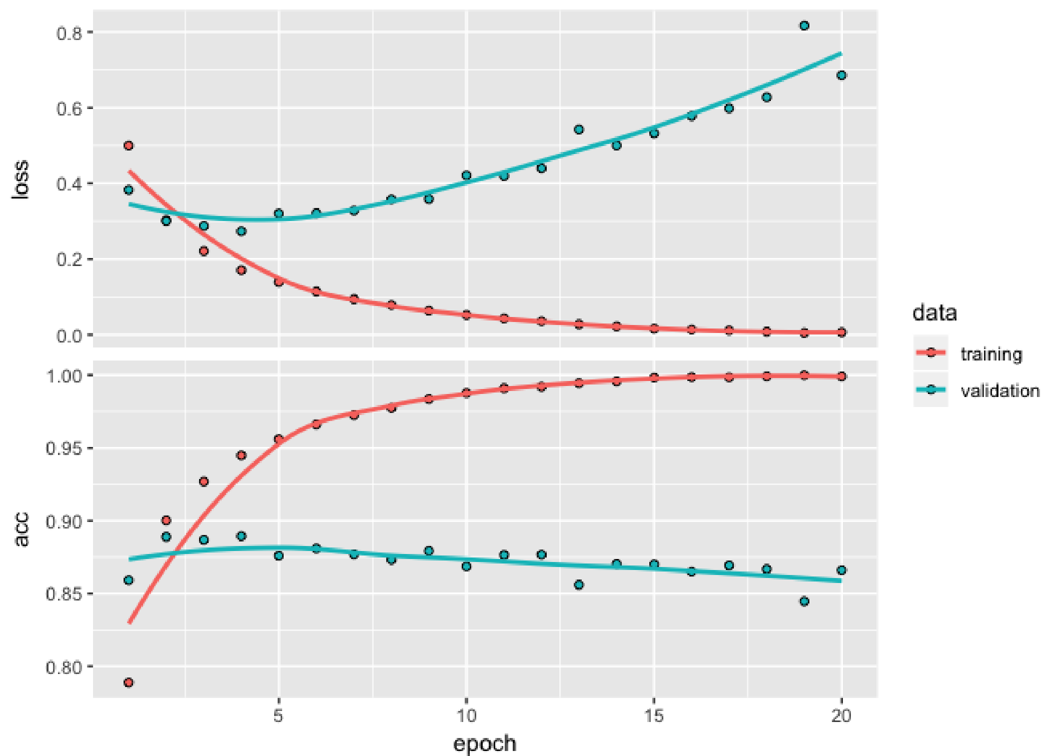
Note that the call to `fit()` returns a history object. Let's take a look at it:

```
str(history)
```

```
## List of 2
##  $ params :List of 8
##   ..$ metrics           : chr [1:4] "loss" "acc" "val_loss" "val_acc"
##   ..$ epochs            : int 20
##   ..$ steps             : NULL
##   ..$ do_validation     : logi TRUE
##   ..$ samples           : int 15000
##   ..$ batch_size        : int 512
##   ..$ verbose           : int 1
##   ..$ validation_samples: int 10000
##  $ metrics:List of 4
##   ..$ acc     : num [1:20] 0.789 0.9 0.927 0.945 0.956 ...
##   ..$ loss    : num [1:20] 0.5 0.302 0.221 0.171 0.139 ...
##   ..$ val_acc : num [1:20] 0.859 0.889 0.887 0.889 0.876 ...
##   ..$ val_loss: num [1:20] 0.383 0.3 0.287 0.274 0.32 ...
##  - attr(*, "class")= chr "keras_training_history"
```

The history object includes parameters used to fit the model (`history$params`) as well as data for each of the metrics being monitored (`history$metrics`).

```
plot(history)
```



- You can customize all of this behavior via various arguments to the `plot()` method.
- We can create custom visualization by using `as.data.frame()` method on the history to obtain a data frame with factors for each metric as well as training versus validation:

```
history_df <- as.data.frame(history)
head(history_df)
```

```
##   epoch     value metric     data
## 1     1 0.4997946   loss training
## 2     2 0.3020927   loss training
## 3     3 0.2211647   loss training
## 4     4 0.1707503   loss training
## 5     5 0.1389315   loss training
## 6     6 0.1142671   loss training
```

This fairly naive approach achieves an accuracy of 88%. With state-of-the-art approaches, you should be able to get close to 95%.

Predicting on new data

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

```
##                  [,1]
##  [1,] 0.0058080279
```

```
##  [2,] 1.0000000000
##  [3,] 0.7080981731
##  [4,] 0.9868260026
##  [5,] 0.9978235960
##  [6,] 0.9996370077
##  [7,] 0.6307815909
##  [8,] 0.0000171118
##  [9,] 0.9769185185
## [10,] 1.0000000000
```

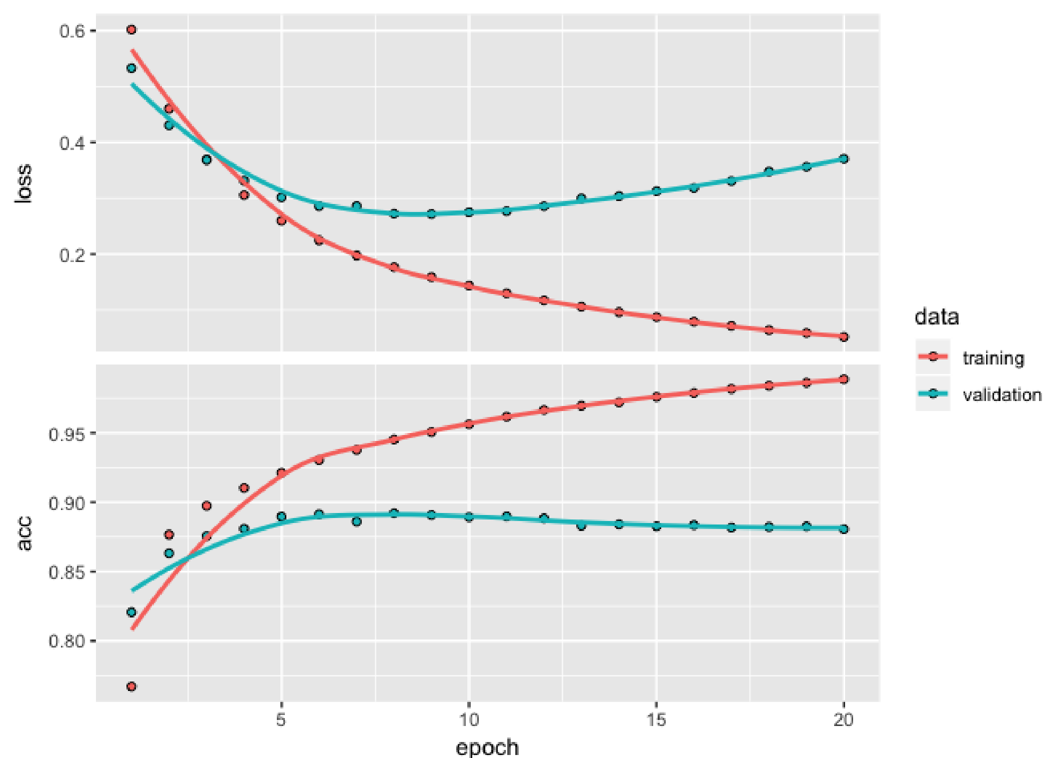
Fighting overfitting

Reducing the network's size

Let's try a smaller network:

```
history <- 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(optimizer = "rmsprop",
    loss = "binary_crossentropy",
    metrics = c("accuracy")) %>%
  fit(
    partial_x_train,
    partial_y_train,
    epochs = 20,
    batch_size = 512,
  validation_data = list(x_val, y_val))
```

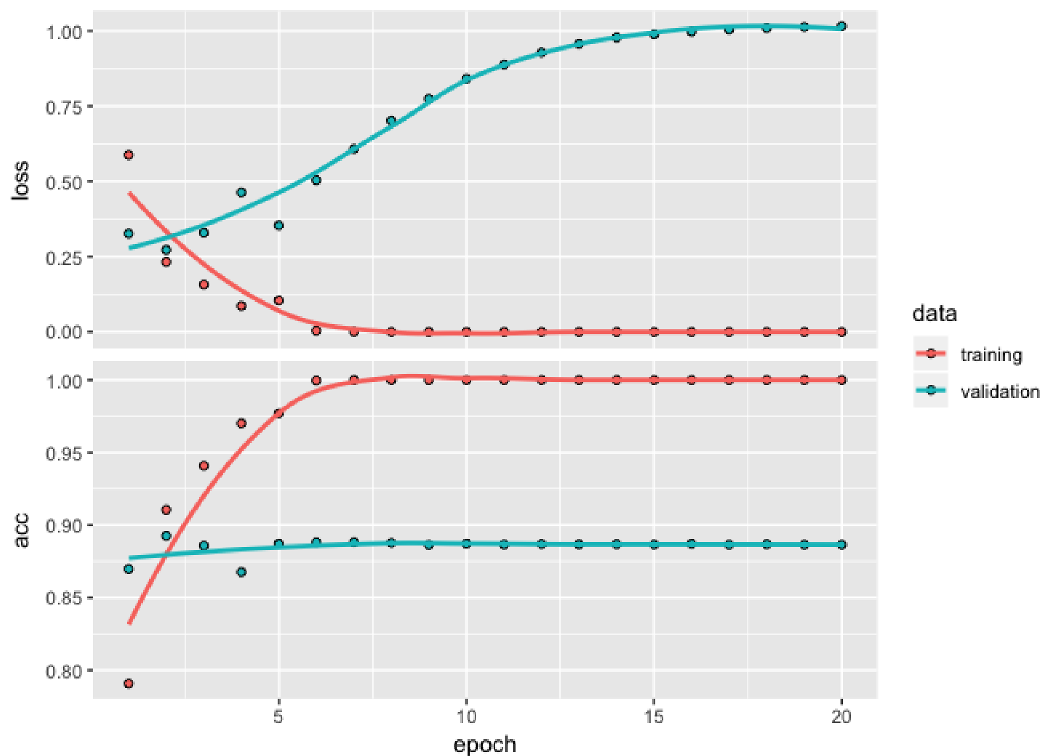
```
plot(history)
```



And a bigger network:

```
history <- keras_model_sequential() %>%
  layer_dense(units = 512, activation = "relu", input_shape = c(10000)) %>%
  layer_dense(units = 512, activation = "relu") %>%
  layer_dense(units = 1, activation = "sigmoid") %>%
  compile(optimizer = "rmsprop",
    loss = "binary_crossentropy",
    metrics = c("accuracy")) %>%
  fit(
    partial_x_train,
    partial_y_train,
    epochs = 20,
    batch_size = 512,
  validation_data = list(x_val, y_val))
```

```
plot(history)
```



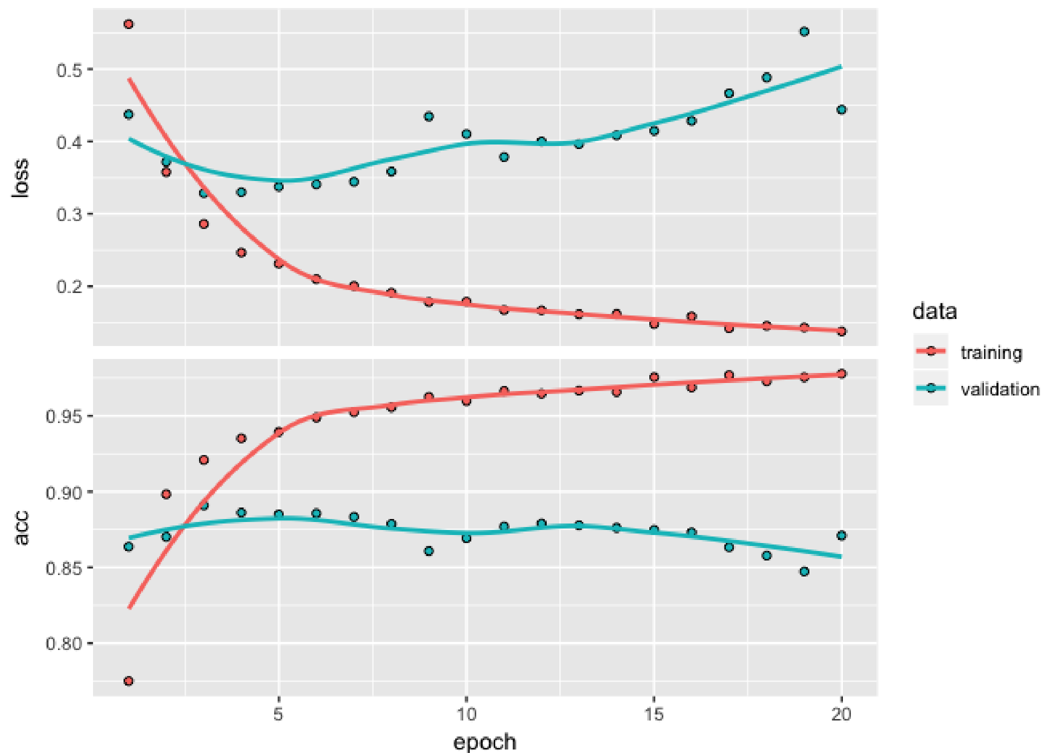
Adding weight regularization

Adding L2 weight regularization to the model:

```
history <- keras_model_sequential() %>%
  layer_dense(units = 16, kernel_regularizer = regularizer_l2(0.001),
              activation = "relu", input_shape = c(10000)) %>%
  layer_dense(units = 16, kernel_regularizer = regularizer_l2(0.001),
              activation = "relu") %>%
  layer_dense(units = 1, activation = "sigmoid") %>%
  compile(optimizer = "rmsprop",
    loss = "binary_crossentropy",
    metrics = c("accuracy")) %>%
  fit(
```

```
    partial_x_train,
    partial_y_train,
    epochs = 20,
    batch_size = 512,
  validation_data = list(x_val, y_val))
```

```
plot(history)
```

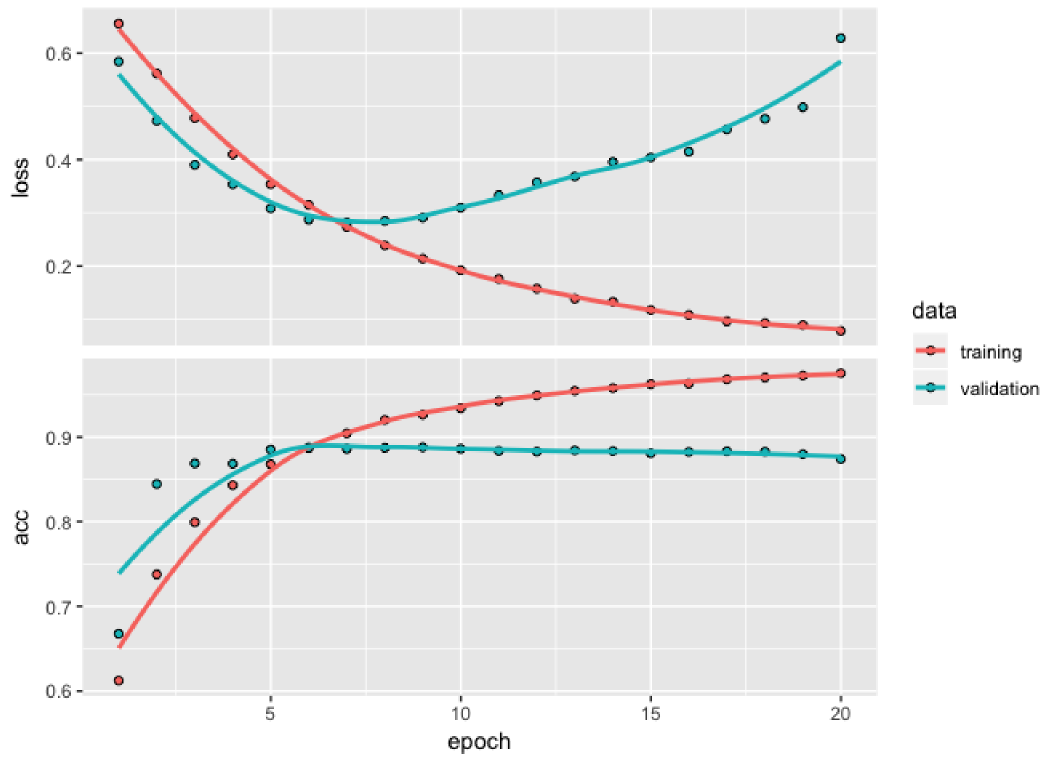


Adding dropout

Let's add two dropout layers in the IMDB network to see how well they do at reducing overfitting.

```
history <- keras_model_sequential() %>%
  layer_dense(units = 16, activation = "relu", input_shape = c(10000)) %>%
  layer_dropout(rate = 0.5) %>%
  layer_dense(units = 16, activation = "relu") %>%
  layer_dropout(rate = 0.5) %>%
  layer_dense(units = 1, activation = "sigmoid") %>%
  compile(optimizer = "rmsprop",
    loss = "binary_crossentropy",
    metrics = c("accuracy")) %>%
  fit(
    partial_x_train,
    partial_y_train,
    epochs = 20,
    batch_size = 512,
  validation_data = list(x_val, y_val))
```

```
plot(history)
```



loss
0.6
0.4
0.2
acc
0.9
0.8
0.7
0.6
5
10
15
20
epoch
data
training
validation