Module 11: Recommended Exercises

Statistical Learning V2021

alexaoh

23 mai, 2021

Problem 1

a)

Write down the equation that describes and input is related to output in this network, using general activation functions ϕ_o, ϕ_h and ϕ_{h^*} and bias nodes in all layers. What would you call such a network?

The given network is called a 4-4-4-3 feedforward network, with one bias node in each layer (except the output of course). The equation that describes the network can be written as shown in the following. The nodes in the first hidden layer are denoted by

$$z_m(x) = \phi_{h^*}(\alpha_{0m} + \sum_{j=1}^{4} \alpha_{jm} x_j), \quad m = 1, \dots, 4,$$

where α_{jm} is the weight from input j to hidden node m and α_{0m} is the bias term for the m^{th} hidden node. The nodes in the second hidden layer can be denoted in the similarly, e.g. with the letter $y_l(x), \quad l = 1, \dots 4$, with weights γ_{ij} and $l = 1, 2, 3, 4$ nodes. Finally, the output layer can be denoted by

$$y_c(\boldsymbol{x}) = \phi_o(\beta_{0c} + \sum_{l=1}^{4} \beta_{lc} y_l) = \phi_o(\beta_{0c} + \sum_{l=1}^{4} \beta_{lc} \phi_h(\gamma_{0l} + \sum_{m=1}^{4} \alpha_{ml} z_m) = \phi_o(\beta_{0c} + \sum_{l=1}^{4} \beta_{lc} \phi_h(\gamma_{0l} + \sum_{m=1}^{4} \alpha_{ml} \phi_{h^*}(\alpha_{0m} + \sum_{j=1}^{4} \alpha_{jm} x_j))),$$

for $c = 1, 2, 3$, i.e. all the output nodes.

b)

The following image is the illustration of an artificial neural network at Wikipedia.

- What can you say about this network architecture? *Answer*: It is feedforward. Moreover, the hidden layer has four nodes, but no bias node, since there are weights "coming into" each of the nodes. The input layer has either 3 input nodes or 2 input nodes and a bias node. The output layer has two nodes.
- What do you think it can be used for (regression/classification)? *Answer*: It can be used for binary or tertiary classification (depending on how the nodes are defined), since there are two nodes in the output layer. Softmax activation function can be used for two classes. However, for two classes, one output node is most commonly used. Moreover, it can be used for regression with two responses.

c)

What are the similarities and differences beween a feedforward neural network with one hidden layer with `linear` activation and `sigmoid` output (one output) and logistic regression?

- Similarities: The neural network gives approximately the same coefficients as the logistic regression, since the input is “squashed” into the interval $[0, 1]$ when output from the network, because of the sigmoid activation function. They will not give the exact same coefficient estimates, since they use different algorithms to reach minimum.
- Differences: Use different algorithms. The neural network must estimate a lot more coefficients. The logistic regression is much better for interpretation compared to the neural network. However, the hidden layer may find latent structures in the data, with the help of the hidden layer, despite the activation function of the hidden layer being linear.

d)

In a feedforward neural network you may have $10'000$ weights to estimate but only 1000 observations. How is this possible?

This is possible because you may have 1000 observations in the input layer, but an arbitrarily deep neural network, with arbitrarily wide layers. This is possible to calculate because of the use of iterative methods (gradient descent and backpropagation), which means that there are no unique solutions. The network will benefit greatly by adding some sort of regularization, like weight decay and early stopping.

Problem 2

a)

Which network architecture and activation functions does this formula correspond to?

$$\hat{y}_1(\mathbf{x}) = \beta_{01} + \sum_{m=1}^{5} \beta_{m1} \cdot \max(\alpha_{0m} + \sum_{j=1}^{10} \alpha_{jm} x_j, 0)$$

The hidden layer has the ReLu activation function. Moreover, the network has 10 input nodes (in addition to a bias node) and one hidden layer, with 5 nodes (in addition to a bias node).

This formula corresponds to a feedforward network with one input layer (with 10 input nodes and one bias node), one hidden layer and one output layer with one node. The hidden layer has a ReLU activation function and 5 nodes in addition to a bias node.

How many parameters are estimated in this network?

Hence, in this network, $11 \cdot 5 + 6 = 61$ parameters are estimated.

b)

Which network architecture and activation functions does this formula give?

$$\hat{y}_1(\mathbf{x}) = (1 + \exp(-\beta_{01} - \sum_{m=1}^{5} \beta_{m1} \max(\gamma_{0m} + \sum_{l=1}^{10} \gamma_{lm} \max(\sum_{j=1}^{4} \alpha_{jl} x_j, 0), 0))^{-1}$$

This corresponds to a network with 4 input nodes, one hidden layer with 10 nodes (in addition to a bias node), another hidden layer with 5 nodes (in addition to a bias node) and one output node. The first and second hidden layers have the ReLU activation function, while the output layer has a sigmoid activation function.

How many parameters are estimated in this network?

Hence, in this network, $(4 \cdot 10) + (10 + 1) \cdot 5 + (5 + 1) = 101$ parameters are estimated.

c)

In a regression setting: Consider

- A sum of non-linear functions of each covariate in Module 7.
- A sum of many non-linear functions of sums of covariates in feedforward neural networks (one hidden layer, non-linear activation in hidden layer) in Module 11.

Explain how these two ways of thinking differ? Pros and cons?

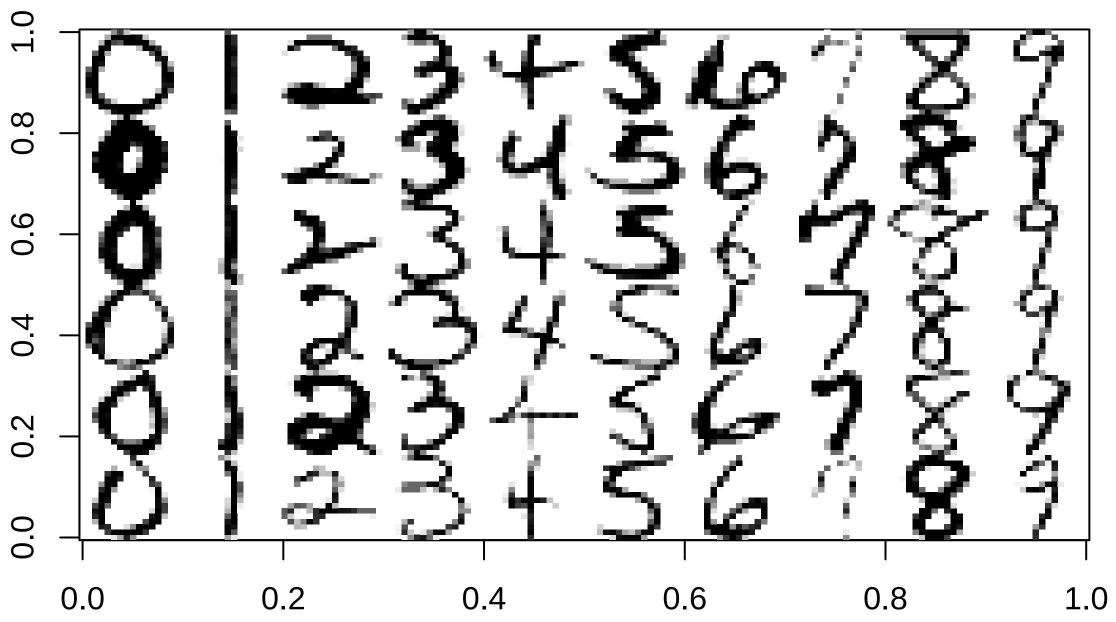
Answer: The first model is simpler, while the second is more complex. The second model gives two transformations of the data, while the first only transforms it once. Pros with the second model is that it opens to as many non-linear transformations of the data as one could desire, since one can add more hidden layers in the network. Cons with the second way is that it needs iterative solvers and it cannot be guaranteed that a global minimizer of the cost function is found. A pro with the first method is that the parameters in the model can be estimated precisely using linear algebra. Another important pro of the first method is that the estimated coefficients are easy to interpret and, hence, it can be used for inference, where as this is extremely hard (if not impossible) in the neural network setting. Interactions are automatically handled in the neural network setting, since there are non-linear functions of sums from the previous layers, while the interactions need to be added manually in the additive model in the first method. Lastly, a neural net with at least one hidden layer and a "squashing type" activation function, can approximate any function (with finite-dimensional domain and co-domain).

Problem 3: Handwritten digit recognition data

The following problem involves classification of handwritten digits from 0 to 9. For more details see `?zip.train` in `ElemStatLearn` R package or see Section 11.7 in the Book Elements of Statistical Learning.

Note: the `ElemStatLearn` package has been archived from CRAN, so to get ahold of the data, download the archived package from https://cran.r-project.org/src/contrib/Archive/ElemStatLearn/ElemStatLearn_2015.6.26.tar.gz, and then in RStudio, click the "Packages" tab, then click "Install", select "Install from: Package Archive File", and then select the `.tar.gz` file you downloaded above.

We have images of handwritten digits in grayscale of size 16×16 which are stored as vectors of size 256. There are 7291 training observations and 2007 test observations



---

a)

Preprocessing is important before we fit a neural network. Standardize the features and transform the digits to factors for both train and test data.

Remember that it is important to decide on the scale based on the training data, and then apply the same standardization for both training and test data (i.e., centering around the same mean and transforming to the same variance).

```r
library(ElemStatLearn)
train.data <- zip.train[, -1]
train.labels <- factor(zip.train[, 1])
test.data <- zip.test[, -1]
test.labels <- factor(zip.test[, 1])

# Standardize the data.
mean <- apply(train.data, 2, mean)
std <- apply(train.data, 2, sd)
train.data <- scale(train.data, center = mean, scale = std)
test.data <- scale(test.data, center = mean, scale = std)
```

b)

Use a feedforward-network with one hidden layer to train and predict using the `nnet()` function from the `nnet` R package. Include 5 hidden nodes plus a bias node in the input and hidden layers. What is the number of parameters? Look at the confusion table for the test data. How good does the network perform?

```r
library(nnet)
fit.nnet <- nnet(train.labels ~ ., data = train.data, size = 5, MaxNWts = 3000,
    maxit = 5000)
# summary(fit.nnet)
pred <- predict(fit.nnet, newdata = test.data, type = "class")
library(caret)
confusionMatrix(factor(pred), test.labels)
```

The number of parameters is $(257 \cdot 5) + (6 \cdot 10) = 1345$. The performance of the neural network can be observed in the output.

Problem 4: Deep Learning with Keras

We again work on a handwritten digit recognition problem, but now we use the data from the keras library. This dataset consists of 60000 training data of 28×28 grayscale images and 10000 test data.

For more info see here

If you have already installed `keras` ignore the following chunk of code

```r
# Install the keras R package install.packages('keras') Install the
# core Keras library + TensorFlow
library(keras)
install_keras()
# for machines with NVIDIA GPU install_keras(tensorflow = 'gpu') Did
# not switch to NVIDIA GPU this time.
```

Data Preprocessing

```r
library(keras)
mnist = dataset_mnist()
x_train = mnist$train$x
y_train = mnist$train$y
x_test = mnist$test$x
y_test = mnist$test$y
# reshape
x_train = array_reshape(x_train, c(nrow(x_train), 28 * 28))
x_test = array_reshape(x_test, c(nrow(x_test), 28 * 28))
# rescale
x_train = x_train/255
x_test = x_test/255
y_train = to_categorical(y_train, 10)
y_test = to_categorical(y_test, 10)
```

a)

In order to fit a model in keras we can use the following steps. We fit a densely (or fully) conected neural network with 2 hidden layers. Fill in the missing inputs and run the model.

```r

model <- keras_model_sequential() %>%
  layer_dense(units = 8, activation = 'relu', input_shape = c(28*28)) %>% # fill in the length of the i
  layer_dense(units = 8, activation = 'relu') %>%
  layer_dense(units = 10, activation = "softmax") # fill in the name of the activation function
summary(model)
```

1. Define the model

```
#> Model: "sequential"
#> ________________________________________________________________________________
#> Layer (type)                        Output Shape                        Param #
#> ================================================================================
#> dense_2 (Dense)                     (None, 8)                           6280
#> ________________________________________________________________________________
#> dense_1 (Dense)                     (None, 8)                           72
#> ________________________________________________________________________________
#> dense (Dense)                       (None, 10)                          90
#> ================================================================================
#> Total params: 6,442
#> Trainable params: 6,442
#> Non-trainable params: 0
#> ________________________________________________________________________________
```

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

Possible choices of the activation function are 'sigmoid', 'softmax' or the identity function (which means that we don't have to define any activation argument in the last layer). What is the use of identity function?

The identity function is used in the output layer for regression problems.

```
# Does not work to compile in a different code block than the model
# definition for some reason (?) model %>% compile(optimizer =
# 'rmsprop', loss = 'categorical_crossentropy', metrics =
# c('accuracy'))
```

2. Compile Possible choices of loss function are 'binary_crossentropy', 'categorical_crossentropy' and 'mse'. For metrics you can use 'mean_absolute_error' or 'accuracy'.

```
history <- model %>% fit(x_train, y_train, epochs = 20, batch_size = 128,
    validation_split = 0.2, verbose = 0)
```

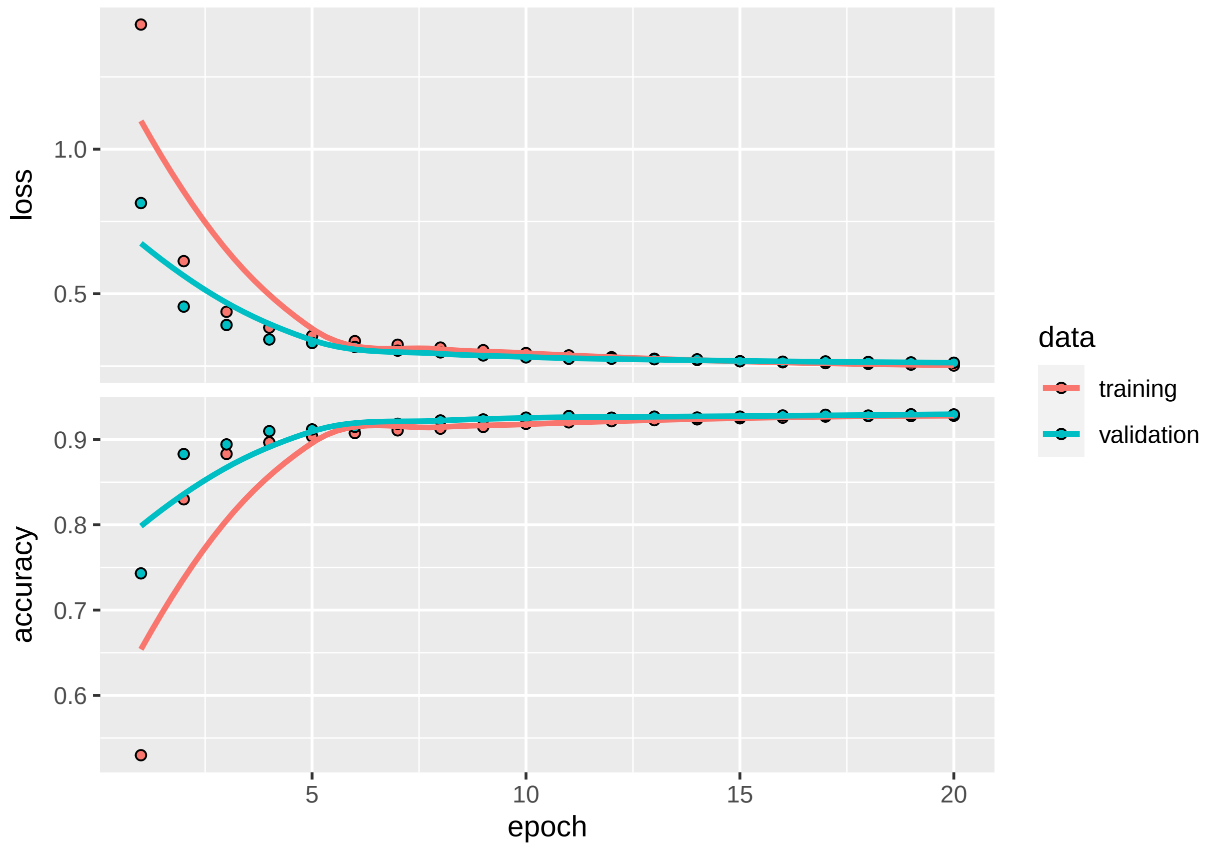
3. Train You can find more information about the different arguments in the `fit()` function here: https://keras.rstudio.com/reference/fit.html

Report the accuracy on the test data.

```
str(history)
```

```
#> List of 2
#>  $ params :List of 3
#>   ..$ verbose: int 0
#>   ..$ epochs : int 20
#>   ..$ steps  : int 375
#>  $ metrics:List of 4
#>   ..$ loss        : num [1:20] 1.431 0.613 0.438 0.383 0.354 ...
#>   ..$ accuracy    : num [1:20] 0.53 0.83 0.883 0.897 0.904 ...
#>   ..$ val_loss    : num [1:20] 0.814 0.456 0.392 0.342 0.33 ...
#>   ..$ val_accuracy: num [1:20] 0.743 0.883 0.894 0.91 0.912 ...
#>  - attr(*, "class")= chr "keras_training_history"
```

```
plot(history)
```



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

```
#>      loss  accuracy
#> 0.2726067 0.9254000
```

The accuracy on the test data is 92.7%, as reported above.

What is the number of parameters?

The number of parameters are 6442.

b)

Now fit a model with 2 hidden layers with 128 units each. What do you see?

```
model2 <- keras_model_sequential() %>% layer_dense(units = 128, activation = "relu",
    input_shape = c(28 * 28)) %>% layer_dense(units = 128, activation = "relu") %>%
    layer_dense(units = 10, activation = "softmax")
summary(model2)
```

```
#> Model: "sequential_1"
#> ________________________________________________________________________________
#> Layer (type)                        Output Shape                      Param #
#> ================================================================================
#> dense_5 (Dense)                     (None, 128)                       100480
#> ________________________________________________________________________________
#> dense_4 (Dense)                     (None, 128)                     16512
#> ________________________________________________________________________________
#> dense_3 (Dense)                     (None, 10)                      1290
#> ================================================================================
#> Total params: 118,282
#> Trainable params: 118,282
#> Non-trainable params: 0
#> ________________________________________________________________________________
```

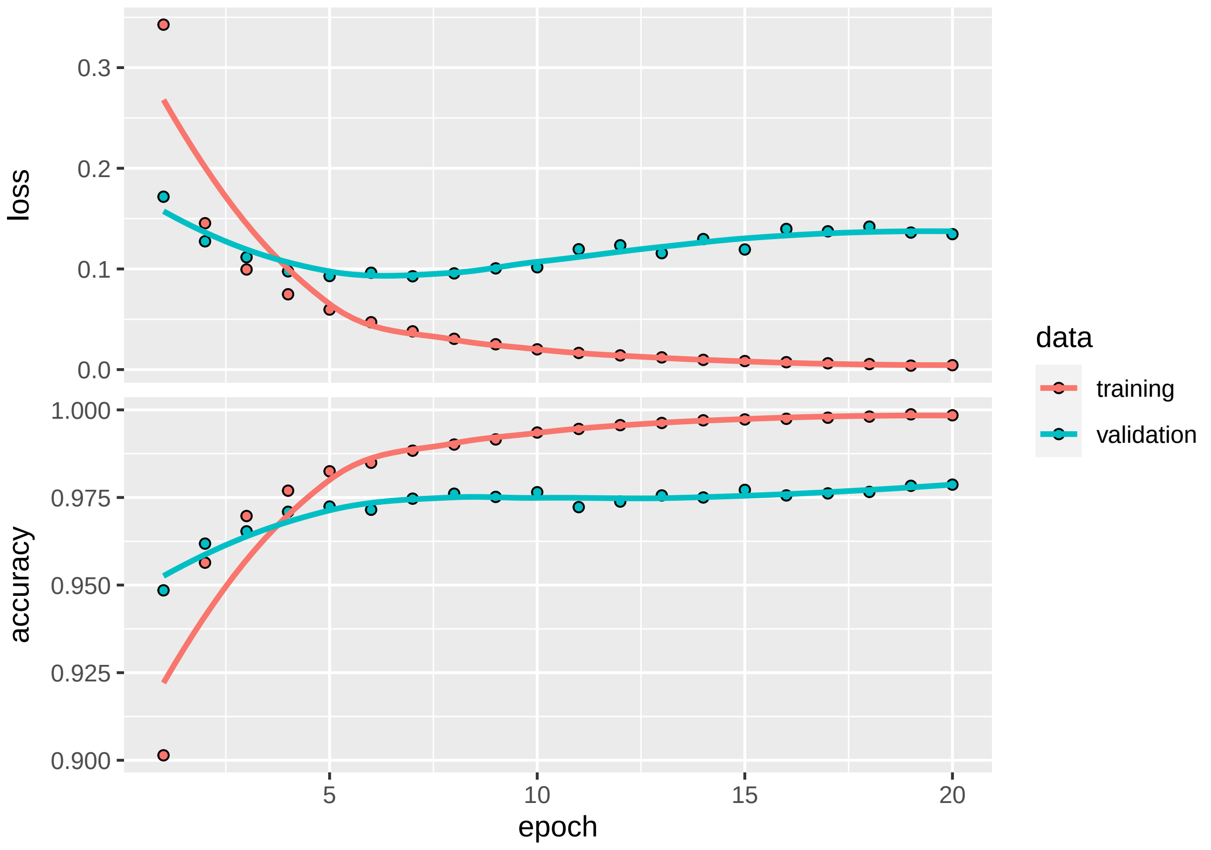
```
# Compile.
model2 %>% compile(optimizer = "rmsprop", loss = "categorical_crossentropy",
    metrics = c("accuracy"))

# Train.
history2 <- model2 %>% fit(x_train, y_train, epochs = 20, batch_size = 128,
    validation_split = 0.2, verbose = 0)

# Evaluate.
str(history2)
```

```
#> List of 2
#>  $ params :List of 3
#>   ..$ verbose: int 0
#>   ..$ epochs : int 20
#>   ..$ steps  : int 375
#>  $ metrics:List of 4
#>   ..$ loss        : num [1:20] 0.3427 0.1454 0.0995 0.0748 0.0596 ...
#>   ..$ accuracy    : num [1:20] 0.901 0.956 0.97 0.977 0.982 ...
#>   ..$ val_loss    : num [1:20] 0.1718 0.1274 0.1116 0.0976 0.0929 ...
#>   ..$ val_accuracy: num [1:20] 0.948 0.962 0.965 0.971 0.972 ...
#>  - attr(*, "class")= chr "keras_training_history"
```

```
plot(history2)
```



```r
model2 %>% evaluate(x_test, y_test)
```

```
#>      loss  accuracy
#> 0.1203172 0.9783000
```

We see that the validation loss reaches its minimum after 5-10 epochs and increases again after this. Thus, the larger network overfits after the first 5-10 epochs, which is a problem.

c)

In order to avoid the problem we have seen in b), we can use weight regularization (L1 and L2 norms) or dropout. Apply these methods to the network from b).

```r
# Regularization with weight decay with L2 norm.  Build the model
model3 <- keras_model_sequential() %>% layer_dense(units = 128, activation = "relu",
    input_shape = c(28 * 28), kernel_regularizer = regularizer_l2(l = 0.001)) %>%
    layer_dense(units = 128, activation = "relu", kernel_regularizer = regularizer_l2(l = 0.001)) %>%
    layer_dense(units = 10, activation = "softmax")
summary(model3)
```

```
#> Model: "sequential"
#> ________________________________________________________________________________
#> Layer (type)                        Output Shape                    Param #
#> ================================================================================
#> dense_2 (Dense)                     (None, 128)                     100480
#> ________________________________________________________________________________
#> dense_1 (Dense)                     (None, 128)                     16512
#> ________________________________________________________________________________
#> dense (Dense)                         (None, 10)                        1290
#> ================================================================================
#> Total params: 118,282
#> Trainable params: 118,282
#> Non-trainable params: 0
#> ________________________________________________________________________________
```

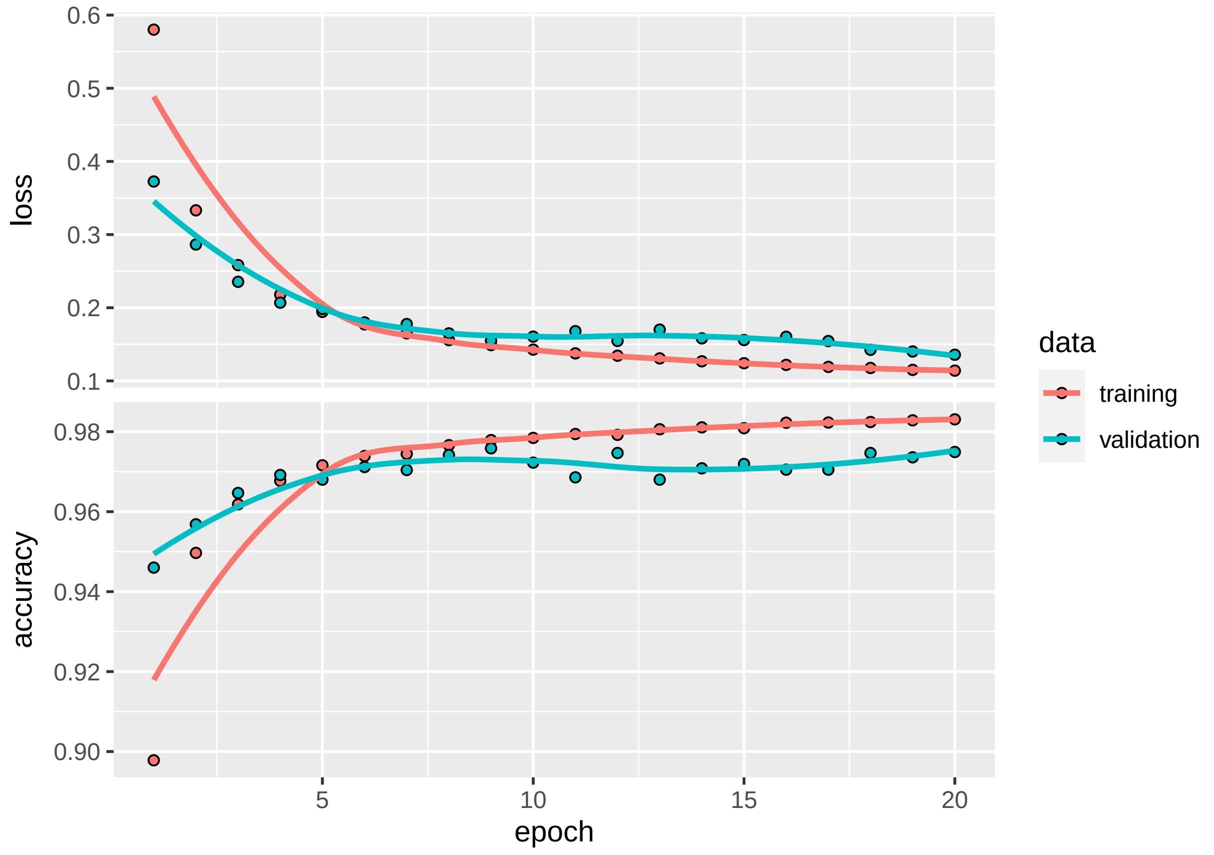
```r
# Compile.
model3 %>% compile(optimizer = "rmsprop", loss = "categorical_crossentropy",
    metrics = c("accuracy"))

# Train.
history3 <- model3 %>% fit(x_train, y_train, epochs = 20, batch_size = 128,
    validation_split = 0.2, verbose = 0)

# Evaluate.
str(history3)
```

```
#> List of 2
#>  $ params :List of 3
#>   ..$ verbose: int 0
#>   ..$ epochs : int 20
#>   ..$ steps  : int 375
#>  $ metrics:List of 4
#>   ..$ loss        : num [1:20] 0.58 0.333 0.258 0.218 0.194 ...
#>   ..$ accuracy    : num [1:20] 0.898 0.95 0.962 0.968 0.972 ...
#>   ..$ val_loss    : num [1:20] 0.373 0.286 0.235 0.207 0.198 ...
#>   ..$ val_accuracy: num [1:20] 0.946 0.957 0.965 0.969 0.968 ...
#>  - attr(*, "class")= chr "keras_training_history"
```

```r
plot(history3)
```



```r
model3 %>% evaluate(x_test, y_test)
```

```
#>      loss  accuracy
#> 0.1332086 0.9770000
```

```r
# Regularization with dropout.  Build the model.
model4 <- keras_model_sequential() %>% layer_dense(units = 128, activation = "relu",
    input_shape = c(28 * 28)) %>% layer_dropout(rate = 0.4) %>% layer_dense(units = 128,
    activation = "relu") %>% layer_dropout(rate = 0.4) %>% layer_dense(units = 10,
    activation = "softmax")
summary(model4)
```

```
#> Model: "sequential_1"
#> ____________________________________________________________________________
#> Layer (type)                        Output Shape                  Param #
#> ============================================================================
#> dense_5 (Dense)                     (None, 128)                   100480
#> ____________________________________________________________________________
#> dropout_1 (Dropout)                 (None, 128)                   0
#> ____________________________________________________________________________
#> dense_4 (Dense)                     (None, 128)                   16512
#> ____________________________________________________________________________
#> dropout (Dropout)                   (None, 128)                   0
#> ____________________________________________________________________________
#> dense_3 (Dense)                     (None, 10)                    1290
#> ============================================================================
#> Total params: 118,282
#> Trainable params: 118,282
#> Non-trainable params: 0
#> ________________________________________________________________________________
```

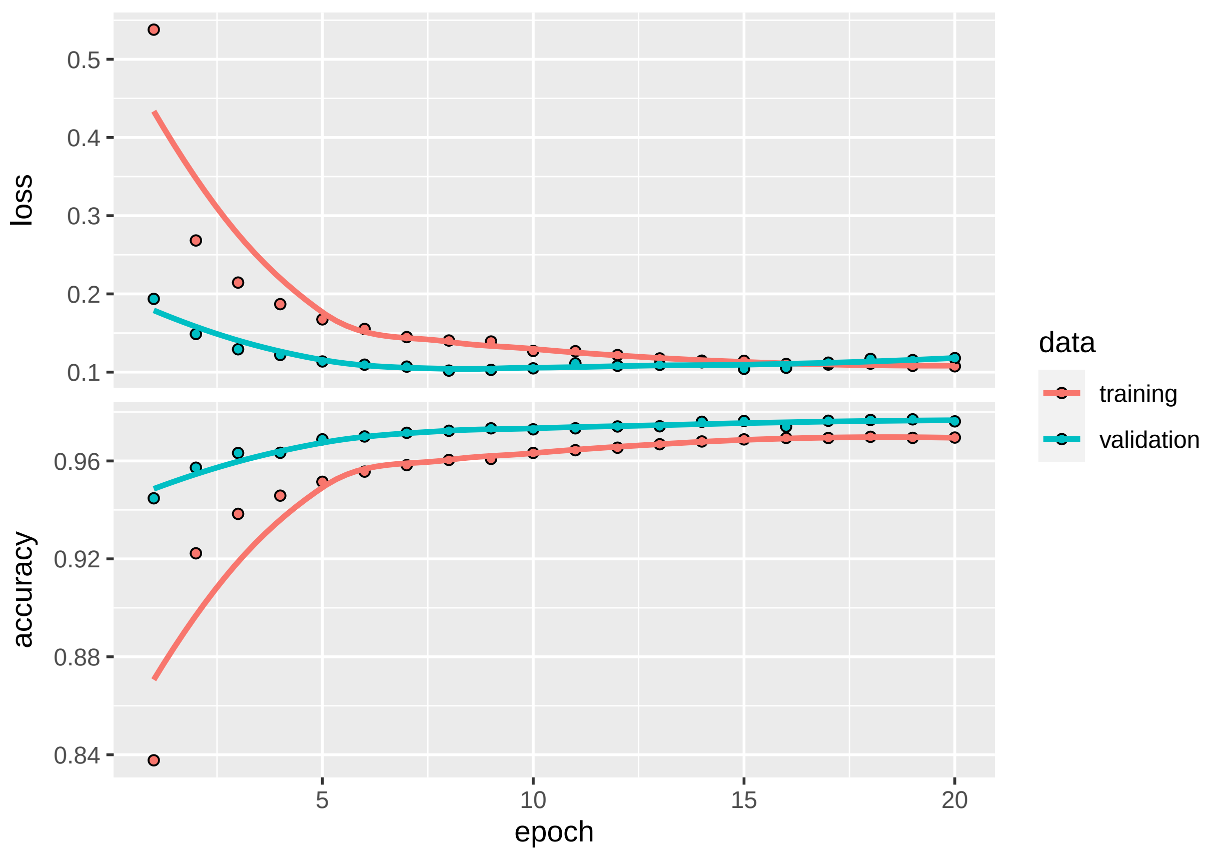
```r
# Compile.
model4 %>% compile(optimizer = "rmsprop", loss = "categorical_crossentropy",
    metrics = c("accuracy"))

# Train.
history4 <- model4 %>% fit(x_train, y_train, epochs = 20, batch_size = 128,
    validation_split = 0.2, verbose = 0)

# Evaluate.
str(history4)
```

```
#> List of 2
#>  $ params :List of 3
#>   ..$ verbose: int 0
#>   ..$ epochs : int 20
#>   ..$ steps  : int 375
#>  $ metrics:List of 4
#>   ..$ loss        : num [1:20] 0.538 0.268 0.214 0.187 0.167 ...
#>   ..$ accuracy    : num [1:20] 0.838 0.922 0.938 0.946 0.951 ...
#>   ..$ val_loss    : num [1:20] 0.194 0.149 0.129 0.122 0.114 ...
#>   ..$ val_accuracy: num [1:20] 0.945 0.957 0.963 0.963 0.969 ...
#>  - attr(*, "class")= chr "keras_training_history"
```

```r
plot(history4)
```



```r
model4 %>% evaluate(x_test, y_test)
```

```
#>     loss accuracy
#> 0.112734 0.975700
```

Now the validation curves look better than before.

For more details on regularization, see here.