Keras Learning Notes

Qi Wang

2023/2/4

Examples to build up a deep learning model

After setting up the environment in the previous step, now I am going to show how can we build up some machine learning models in keras nested in R. There are actually three datasets we are going to explore: MINST, cifar10 and imdb.

MNIST: Modified National Institute of Standards and Technology

The datset is from: http://yann.lecun.com/exdb/mnist/. The MNIST database of handwritten digits, available from this page, has a training set of 60,000 examples, and a test set of 10,000 examples. It is a subset of a larger set available from NIST. The digits have been size-normalized and centered in a fixed-size image. It is a good database for people who want to try learning techniques and pattern recognition methods on real-world data while spending minimal efforts on preprocessing and formatting.

This dataset looks like this:

```
knitr::include_graphics( here::here("pictures/MnistExamples.png"))
```

And the train set is a 60000*28*28 big array, 60000 observations, 28 by 28 pixels of each observation.

*Note1: "%<-%" can subtract values from a list and assign them to several variables simultaneously, which could be useful when we want to return separate values from a function whose return type is a list in R.

*Note2: More about layer_dense() function, input_shape is a parameter that describes the dimension of the covariates, units is a positive integer, describing the dimensions of the output of the current layer, **i.e, number of hidden nodes in the current layer?**, activation describes the type of activation function. If the layer_dense is followed by a layer_dropout, it simply means that in the fitting progress, the connections between the nodes are dropped randomly at the rate given in this function, it's a type of avoid overfitting in neural networking.

```
# Read in MNIST data
c(c(x_train, y_train), c(x_test, y_test)) %<-% keras::dataset_mnist()

# reshape
x_train <- array_reshape(x_train, c(nrow(x_train), 784))  # So we want to reshape each observation from
x_test <- array_reshape(x_test, c(nrow(x_test), 784))  # Same as prervious step

# Rescale
x_train <- x_train / 255
x_test <- x_test / 255


# Change training data to a matrix format
y_train <- to_categorical(y_train, 10)
colnames(y_train) <- 0:9

y_test <- to_categorical(y_test, 10)
colnames(y_test) <- 0:9

head(y_train,5)
```

```
##      0 1 2 3 4 5 6 7 8 9
## [1,] 0 0 0 0 0 1 0 0 0 0
## [2,] 1 0 0 0 0 0 0 0 0 0
## [3,] 0 0 0 0 1 0 0 0 0 0
## [4,] 0 1 0 0 0 0 0 0 0 0
## [5,] 0 0 0 0 0 0 0 0 0 1
```

Then we can build up the model in this way. The input layer has 256 nodes and will pass 256 values to the next layer, the next layer has 128 nodes, finally, the output layer has 10 outputs and since this is a categorical data, we use the activation function in the last layer to be softmax to output a probability.

```
model <- keras_model_sequential()
model %>%
  layer_dense(units = 256, activation = 'relu', input_shape = c(784)) %>%
  layer_dropout(rate = 0.4) %>%
  layer_dense(units = 128, activation = 'relu') %>%
  layer_dropout(rate = 0.3) %>%
  layer_dense(units = 10, activation = 'softmax')
```

```
summary(model)
```

```
## Model: "sequential"
## _________________________________________________________________________
##  Layer (type)                       Output Shape                    Param #
## =========================================================================
##  dense_2 (Dense)                    (None, 256)                     200960
##  dropout_1 (Dropout)                (None, 256)                     0
##  dense_1 (Dense)                    (None, 128)                     32896
##  dropout (Dropout)                  (None, 128)                     0
##  dense (Dense)                      (None, 10)                      1290
## =========================================================================
```

```
## Total params: 235,146
## Trainable params: 235,146
## Non-trainable params: 0
## _________________________________________________________________________________
```

*What does the output shape "None" mean?

*It said I am using CPU instead of GPU, how can I use GPU then?

How did they get the number of the parameters? First, there are 784 covariates for each node, and there are 256 hidden nodes in layer 1, And remember each node also has an intercept. Therefore, the total number of parameters should be $(784 + 1) \times 256 = 200960$, also for the second layer, there are 128 nodes, and from the last layer, there are 256 covariates plus 1 intercept, so the total parameters within this layer is $(256 + 1) \times 128 = 32896$. Furthermore, in the last layer, there are 10 nodes and 129 parameters in each nodes, so 1290 nodes in total.

After coding the model, we need to compile the model, here we need to specify a loss function and an optimizer. For the compile function, the first argument is the model that built in the previous way. Also, there is an argument called "Loss", this parameter is the loss function used to evaluate the performance of the model, like mean_squared_error, categorical_crossentropy, binary_crossentropy, etc. The other argument "optimizer" is the optimization algorithm used to update the model weights. Some common optimizers include adam, rmsprop, sgd, etc. And metrics are some criteria measuring the performance of the model, like accuracy, precision, recall.

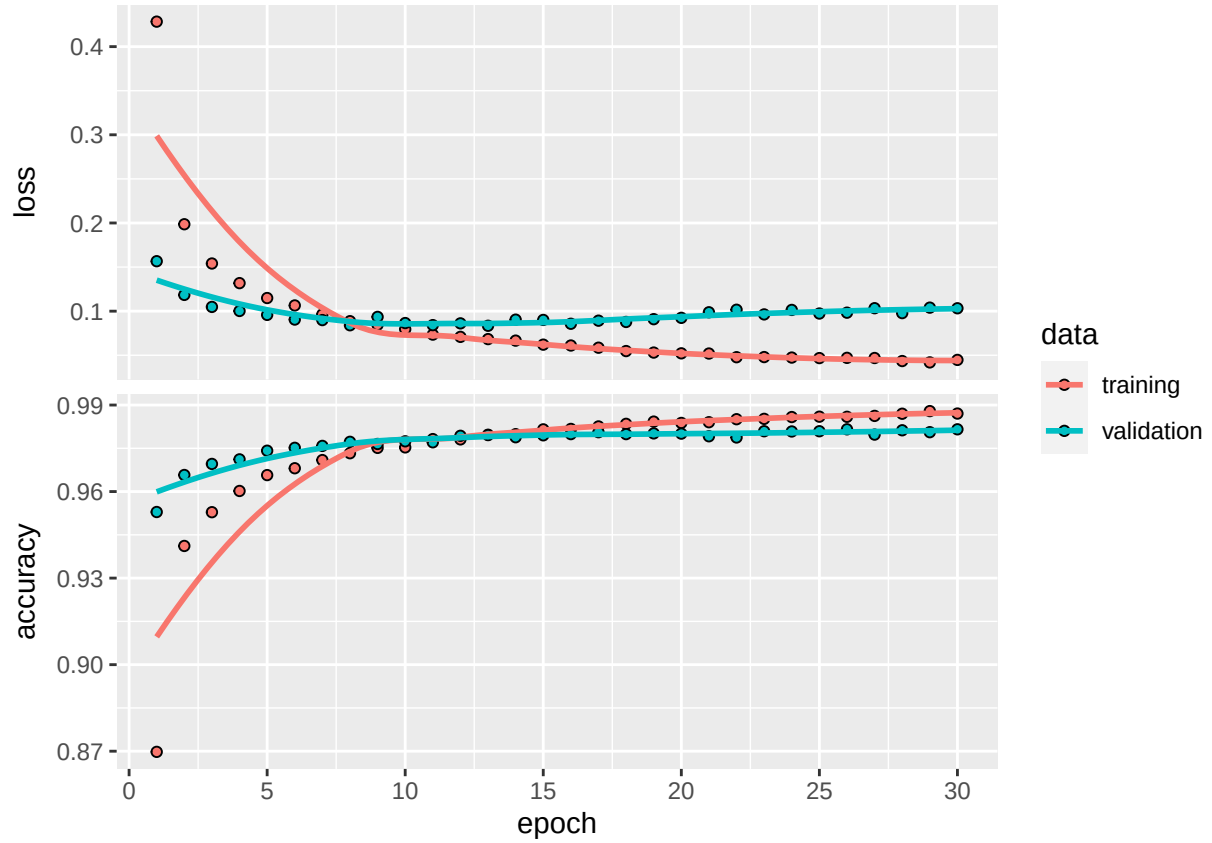
```
model %>% compile(
  loss = 'categorical_crossentropy',
  optimizer = optimizer_rmsprop(),
  metrics = c('accuracy')
)
```

Then, we can fit the model now. We use the fit() function in R. The first argument is the model compiled in the last step, which can be passed by a pipe function. Then, the first argument is the training covariates, and the second argument is the training response. Batch size is the number of samples used to update the gradient each time. Epoch in deep learning means the training set has been all used once, that's an epoch. So the epochs here means how many times should we train the model based on the training set. And the validation split is a parameter between 0 and 1, it describes the fraction of the training data to be used as validation data. It's different from testing because now we are still training the data, not testing the results.

```
history <- model %>% fit(
  x_train, y_train,
  epochs = 30, batch_size = 128,
  validation_split = 0.2
)
```

And then we can plot the data performance:

```
plot(history)
```



However, after training the model, we also need to evaluate the performance of the model based on the testing data, with evaluate function. The first argument is the model we trained in the last step, and then input the testing covariates and testing responses.

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

```
##       loss   accuracy
## 0.09617113 0.98180002
```

*Which model is the function actually using? The last model with 30 epochs? What if it's overfitting? At last, we can predict the test with the function predict_classes(), and the first argument is our model trained in the last step, and another argument we need is the testing covariates made in the last step.

```
model %>% predict(x_test) %>% k_argmax()
```

```
## tf.Tensor([7 2 1 ... 4 5 6], shape=(10000), dtype=int64)
```

*How to get the values? It only return a tf.tensor something, but I want to get the predictions of the picture.