Keras in R

In this example, we will apply the package **keras** in R to build neural network models. For installation of the package, see https://keras.rstudio.com/.

```
library(keras)
```

MNIST Data

In this example, we use the zip code image data `MNIST`. In this dataset, the input `x` is 28 x 28 grayscale images of handwritten digits; the input `y` is the labels for each image.

```
dat <- dataset_mnist()

train_x <- dat$train$x
train_y <- dat$train$y

test_x <- dat$test$x
test_y <- dat$test$y
```

To use the images as features, we convert each 28 x 28 image matrix to a 784 dimensional vector. We also convert the grayscale values from integers ranging between 0 to 255 into floating point values ranging between 0 and 1. In the interest of time, we only use the first 10000 image to train the classifier.

```
train_x <- array(as.numeric(train_x), dim = c(dim(train_x)[[1]], 784))
test_x <- array(as.numeric(test_x), dim = c(dim(test_x)[[1]], 784))

train_x <- train_x/255
test_x <- test_x/255

train_x <- train_x[1:10000,]
train_y <- train_y[1:10000]

train_y_c <- to_categorical(train_y, 10)
test_y_c <- to_categorical(test_y, 10)
```

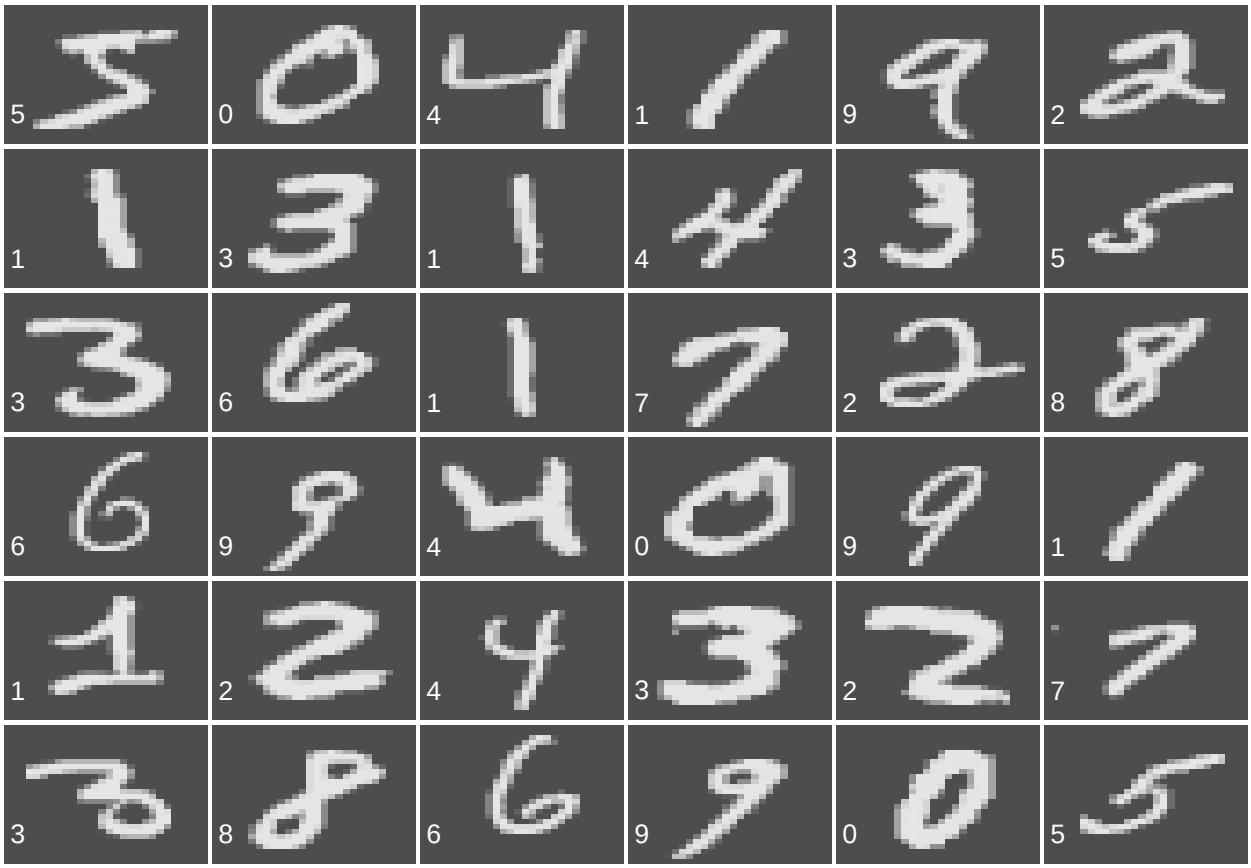
Visualization

```
displayDigit <- function(X)
{
  m <- matrix(unlist(X), nrow = 28, byrow = F)
  m <- t(apply(m, 2, rev))
  image(m, col = grey.colors(255))
}

plotTrain <- function(images)
{
  op <- par(no.readonly = TRUE)
  x <- ceiling(sqrt(length(images)))
  par(mfrow = c(x, x), mar = c(.1, .1, .1, .1))
```

```
  for (i in images)
  {
    m <- matrix(train_x[i,], nrow = 28, byrow = FALSE)
    m <- apply(m, 2, rev)
    image(t(m), col = grey.colors(255), axes = FALSE)
    text(0.05, 0.2, col = "white", cex = 1.2, train_y[i])
  }
  par(op)
}

plotTrain(1:36)
```



Model

We now create our basic network architecture: two hidden layers with 256 and 128 nodes respectively with both hidden layers using ReLU activation functions. To do this, we create a sequential model and then adding layers using the pipe (%>%) operator. The function `layer_dense()` ddd a densely-connected layer to an output. We use the softmax activation for the output layer to compute the probabilities for the classes. Moreover, dropout is one of the most effective and commonly used approaches to prevent overfitting in neural networks. Dropout randomly drops out (setting to zero) a number of output features in a layer during training. We apply drop out with `layer_dropout()`.

```
model <- keras_model_sequential()
```

```
model %>% layer_dense(units = 100, activation ="relu", input_shape = ncol(train_x))  %>%
          layer_dropout(rate = 0.2) %>%
          layer_dense(units = 50, activation = "relu") %>%
          layer_dropout(rate = 0.2) %>%
          layer_dense(units = 10, activation = "softmax")

summary(model)
```

```
## _________________________________________________________________________
## Layer (type)                 Output Shape              Param #
## =========================================================================
## dense (Dense)                (None, 100)               78500
## _________________________________________________________________________
## dropout (Dropout)            (None, 100)               0
## _________________________________________________________________________
## dense_1 (Dense)              (None, 50)                5050
## _________________________________________________________________________
## dropout_1 (Dropout)          (None, 50)                0
## _________________________________________________________________________
## dense_2 (Dense)              (None, 10)                510
## =========================================================================
## Total params: 84,060
## Trainable params: 84,060
## Non-trainable params: 0
## _________________________________________________________________________
```

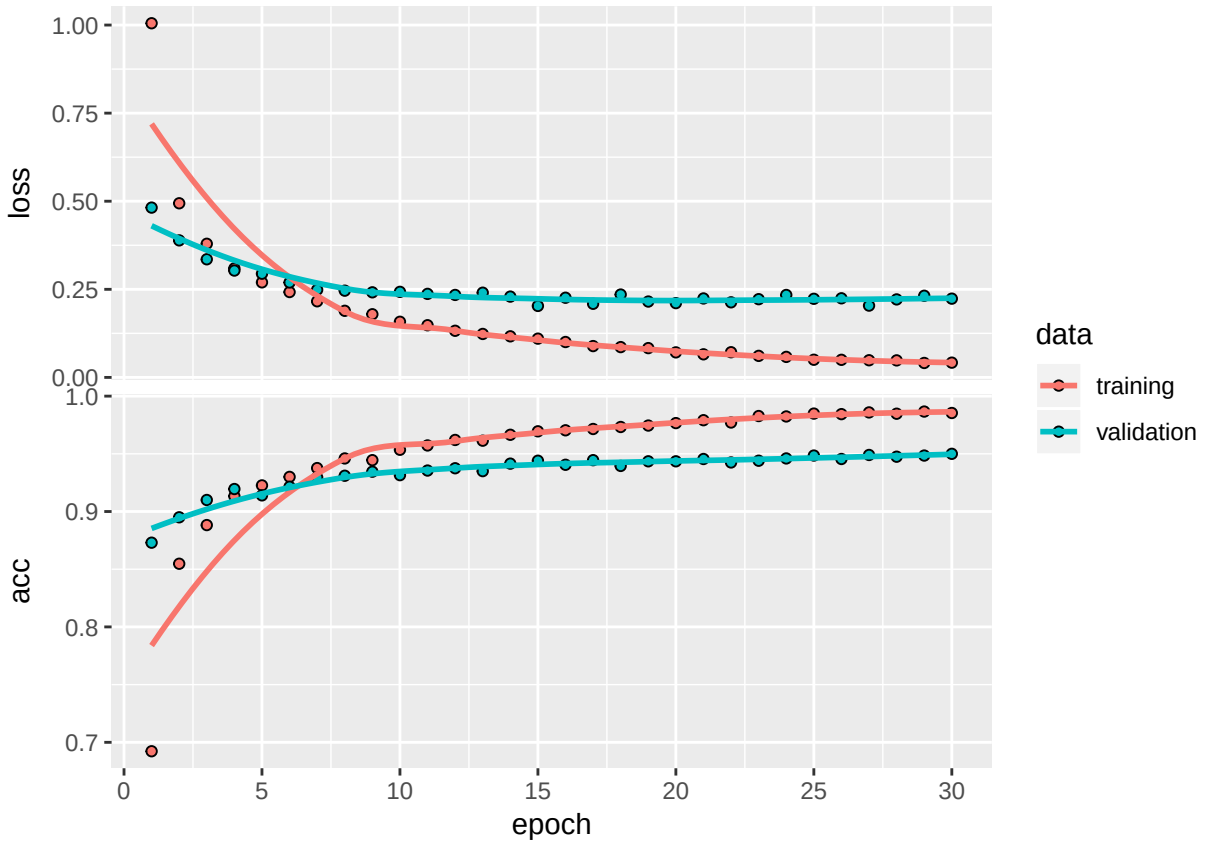
To incorporate backpropagation, we add `compile()` to our code sequence.

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

Now we have created a model, then we just need to train it with our data. We feed our model into the `fit()` function along with our training data. The neural network will run through the mini-batch gradient descent process; the values of `batch_size` are typically provided as a power of two that fit nicely into the memory requirements of the GPU or CPU hardware. An `epoch` describes the number of times the algorithm sees the entire data set. By setting `validation_split = 0.2`, we train our model on 80% of training set and will evaluate the model using the other 20% so that we can compute a more accurate estimate of an out-of-sample error rate.

```
learn <- model %>% fit(train_x, train_y_c, epochs = 30,
                       batch_size = 128,
                       # callbacks = list(callback_tensorboard(log_dir = "logs/run_b")),
                       validation_split = 0.2)

# loss and acuracy metric for each epoch
plot(learn, labels = TRUE)
```




```
# learn$metrics
```

We finally evaluate the model on the test dataset.

```
score <- model %>% evaluate(test_x, test_y_c)
score
```

```
## $loss
## [1] 0.1914149
##
## $acc
## [1] 0.9528
```

```
pred_test <- model %>% predict_classes(test_x)
```

We can also visualize the test performance.

```
plotResults <- function(images, preds)
{
  op <- par(no.readonly = TRUE)
  x <- ceiling(sqrt(length(images)))
  par(mfrow = c(x, x), mar = c(.1,.1,.1,.1))

  for (i in images)
  {
    m <- matrix(test_x[i,], nrow = 28, byrow = FALSE)
    m <- apply(m, 2, rev)
    image(t(m), col = grey.colors(255), axes = FALSE)
    text(0.05, 0.1, col = "red", cex = 1.2, preds[i])
```

```
  }
  par(op)
}

plotResults(201:236, pred_test)
```