Unit 5 Lecture 2: Neural Networks

November 17, 2022

In this R demo, we'll be fitting fully-connected neural networks to the MNIST handwritten digit data.

First let's load some libraries:

```
library(keras)     # for deep learning
library(cowplot)   # for side-by-side plots
library(stat471)   # for deep learning helper functions
library(tidyverse) # for everything else
```

Next let's load the MNIST data and do some reshaping and rescaling:

```
# load the data
mnist <- dataset_mnist()
```

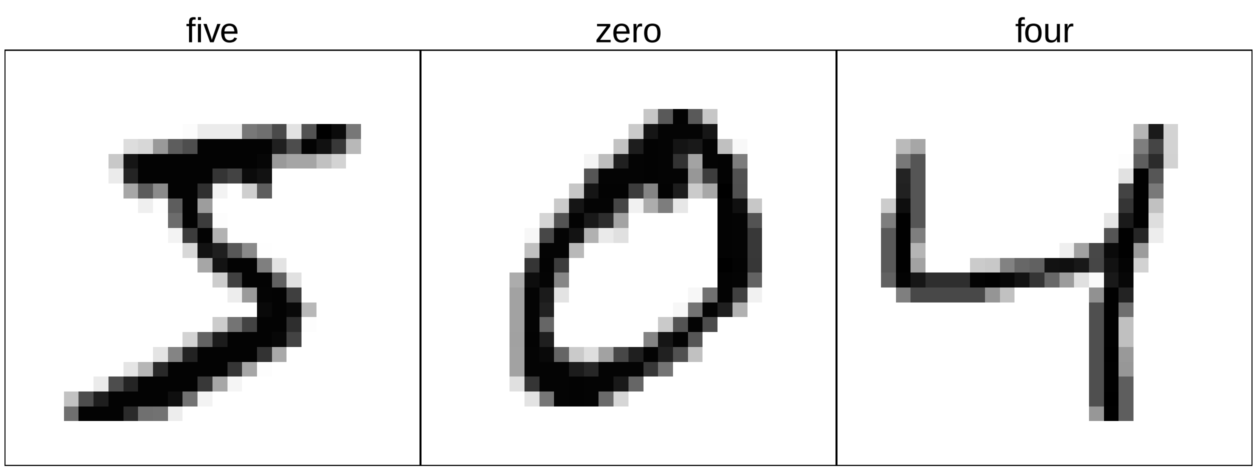
```
## Loaded Tensorflow version 2.9.3
```

```
# extract information about the images
num_train_images <- dim(mnist$train$x)[1]    # number of training images
num_test_images <- dim(mnist$test$x)[1]      # number of test images
img_rows <- dim(mnist$train$x)[2]            # rows per image
img_cols <- dim(mnist$train$x)[3]            # columns per image
num_pixels <- img_rows * img_cols            # pixels per image
num_classes <- length(unique(mnist$train$y)) # number of image classes
max_intensity <- 255                         # max pixel intensity

# normalize and reshape the images
x_train <- array_reshape(
  mnist$train$x / max_intensity,
  c(num_train_images, img_rows, img_cols, 1)
)
x_test <- array_reshape(
  mnist$test$x / max_intensity,
  c(num_test_images, img_rows, img_cols, 1)
)

# extract the responses from the training and test data
g_train <- mnist$train$y
g_test <- mnist$test$y

# recode response labels using "one-hot" representation
y_train <- to_categorical(g_train, num_classes)
y_test <- to_categorical(g_test, num_classes)
```

Quick note that will be helpful for the homework: `plot_grayscale()` accommodates named classes. For example:

```
# tibble of class names, with columns class and name
class_names <- tribble(
```

```
  ~class, ~name,
  0, "zero",
  1, "one",
  2, "two",
  3, "three",
  4, "four",
  5, "five",
  6, "six",
  7, "seven",
  8, "eight",
  9, "nine"
)

p1 <- plot_grayscale(
  image_array = x_train[1, , , ],
  label = g_train[1],
  class_names = class_names
)
p2 <- plot_grayscale(
  image_array = x_train[2, , , ],
  label = g_train[2],
  class_names = class_names
)
p3 <- plot_grayscale(
  image_array = x_train[3, , , ],
  label = g_train[3],
  class_names = class_names
)
plot_grid(p1, p2, p3, nrow = 1)
```



Next, we define a neural network model with one hidden layer with 256 units and dropout rate 0.5.

```
model_nn <- keras_model_sequential() %>%
  layer_flatten(input_shape = c(img_rows, img_cols, 1)) %>% # flatten the input
  layer_dense(units = 256, activation = "relu") %>%         # first hidden layer
  layer_dropout(rate = 0.5) %>%                             # specify dropout as "layer"
  layer_dense(units = 10, activation = "softmax")           # output layer with 10 units
```

Let's print the summary of this neural network:

```r
summary(model_nn)
```

```
## Model: "sequential"
## ________________________________________________________________________________
##  Layer (type)                       Output Shape                    Param #
## ================================================================================
##  flatten (Flatten)                  (None, 784)                     0
##  dense_1 (Dense)                    (None, 256)                     200960
##  dropout (Dropout)                  (None, 256)                     0
##  dense (Dense)                      (None, 10)                      2570
## ================================================================================
## Total params: 203,530
## Trainable params: 203,530
## Non-trainable params: 0
## ________________________________________________________________________________
```

How do we arrive at the total number of parameters in this network?

To train this neural network, we must first define what loss function to use, which optimizer to use, and which metrics to track. We do this by *compiling* the model.

```r
model_nn %>% compile(
  loss = "categorical_crossentropy",  # which loss function to use
  optimizer = optimizer_adagrad(),    # which flavor of stochastic gradient descent to use
  metrics = c("accuracy")             # which metric to track on validation set
)
```

Finally, we can train the model! We use 10 epochs, (mini-)batch size 128, and reserve 20% of our training data for validation.

```r
history <- model_nn %>%
  fit(
    x_train,                # supply training features
    y_train,                # supply training responses
    epochs = 10,            # number of epochs for stochastic gradient descent
    batch_size = 128,       # mini-batch size for stochastic gradient descent
    validation_split = 0.2  # use 20% of the training data for validation
  )
```

```
Epoch 1/10
375/375 [==============================] - 3s 6ms/step - loss: 0.9042 - accuracy: 0.7397 - val_loss: 0.4263 - val_accuracy: 0.8947
Epoch 2/10
375/375 [==============================] - 3s 7ms/step - loss: 0.4929 - accuracy: 0.8600 - val_loss: 0.3365 - val_accuracy: 0.9104
Epoch 3/10
375/375 [==============================] - 3s 7ms/step - loss: 0.4153 - accuracy: 0.8813 - val_loss: 0.2967 - val_accuracy: 0.9190
Epoch 4/10
375/375 [==============================] - 2s 6ms/step - loss: 0.3709 - accuracy: 0.8933 - val_loss: 0.2707 - val_accuracy: 0.9260
Epoch 5/10
375/375 [==============================] - 3s 7ms/step - loss: 0.3395 - accuracy: 0.9040 - val_loss: 0.2521 - val_accuracy: 0.9302
Epoch 6/10
375/375 [==============================] - 3s 7ms/step - loss: 0.3177 - accuracy: 0.9095 - val_loss: 0.2362 - val_accuracy: 0.9356
Epoch 7/10
375/375 [==============================] - 3s 7ms/step - loss: 0.2984 - accuracy: 0.9155 - val_loss: 0.2229 - val_accuracy: 0.9391
Epoch 8/10
375/375 [==============================] - 2s 6ms/step - loss: 0.2816 - accuracy: 0.9213 - val_loss: 0.2123 - val_accuracy: 0.9415
Epoch 9/10
375/375 [==============================] - 2s 6ms/step - loss: 0.2678 - accuracy: 0.9248 - val_loss: 0.2040 - val_accuracy: 0.9438
Epoch 10/10
375/375 [==============================] - 2s 6ms/step - loss: 0.2601 - accuracy: 0.9265 - val_loss: 0.1960 - val_accuracy: 0.9463
```

The number 375 represents the number of mini-batches in one epoch. Why are there 375 of these? The output printed while training gives us information about the metrics on the training and validation data, as well as the time (in seconds) for each epoch and the average time (in milliseconds) for each stochastic gradient step for each epoch.

Now that we've had the patience to wait for this model to train, let's go ahead and save it, along with its history, so we don't need to train it again:

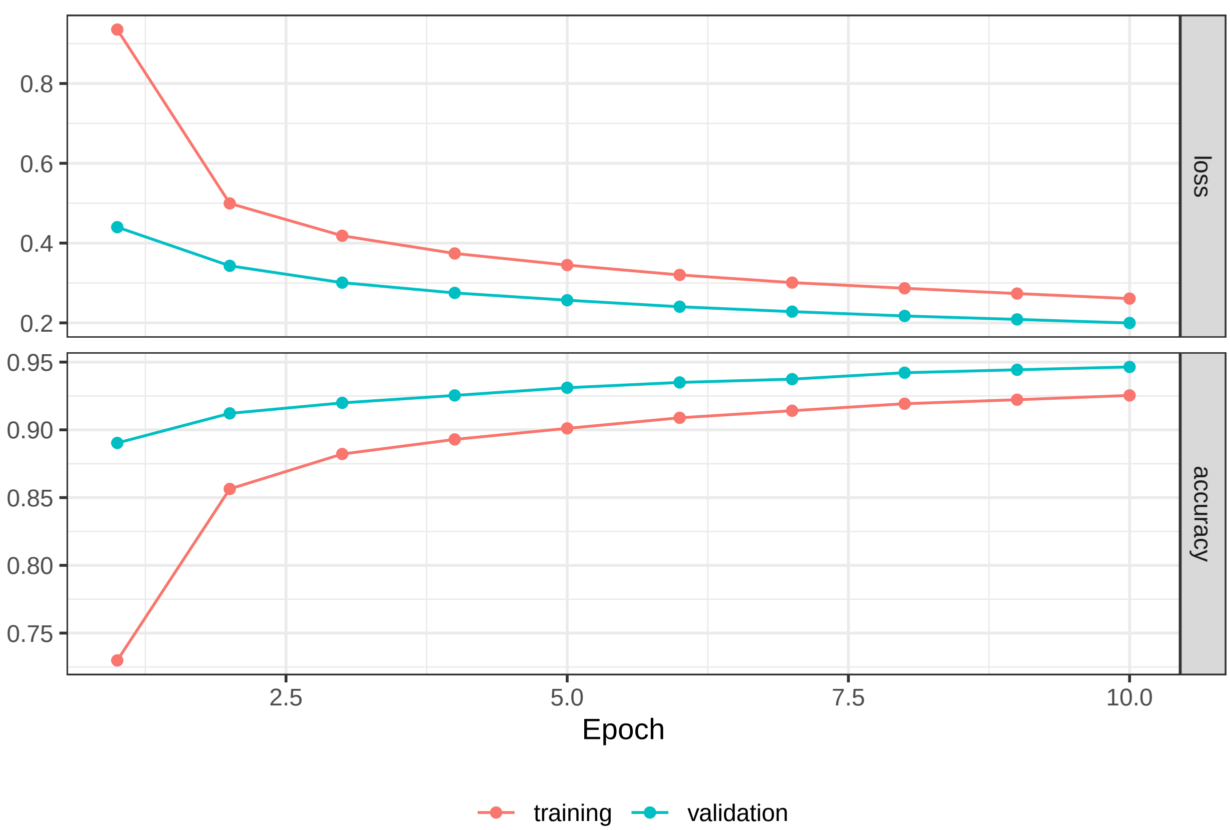
```
# save model
save_model_hdf5(model_nn, "model_nn.h5")

# save history
saveRDS(model_nn$history$history, "model_nn_hist.RDS")
```

We can then load the model and its history into memory again:

```
# load model
model_nn <- load_model_hdf5("model_nn.h5")

# load history
model_nn_hist <- readRDS("model_nn_hist.RDS")
```

We can plot the training history using `plot_model_history()` from `stat471`:

```
plot_model_history(model_nn_hist)
```



Did we observe any overfitting?

As before, we can get the fitted probabilities and predicted classes for the test set using `predict()` and `k_argmax()`:

```
# get fitted probabilities
model_nn %>%
  predict(x_test) %>%
```

```
  head()
```

```
##              [,1]         [,2]         [,3]         [,4]         [,5]
## [1,] 4.432446e-05 1.694478e-06 0.0004147603 0.0007939397 1.411394e-06
## [2,] 2.589406e-03 9.292496e-05 0.9700514078 0.0111381887 5.146796e-07
## [3,] 8.327016e-05 9.763934e-01 0.0067074993 0.0026994499 1.005321e-03
## [4,] 9.983973e-01 6.544514e-07 0.0002865263 0.0001496355 1.050620e-06
## [5,] 5.817412e-04 7.869041e-05 0.0039221221 0.0003440503 9.357622e-01
## [6,] 1.039735e-05 9.909524e-01 0.0016448856 0.0012924505 1.871261e-04
##              [,6]         [,7]         [,8]         [,9]        [,10]
## [1,] 2.680028e-05 3.062761e-07 9.978427e-01 2.230687e-05 8.517543e-04
## [2,] 3.044507e-03 1.117158e-02 8.411126e-08 1.910551e-03 8.908756e-07
## [3,] 1.703593e-03 1.631848e-03 4.926764e-03 4.185070e-03 6.638750e-04
## [4,] 3.423860e-04 3.859953e-04 3.040264e-04 5.175636e-05 8.054014e-05
## [5,] 8.446831e-04 4.206996e-03 3.988254e-03 2.500160e-03 4.777111e-02
## [6,] 1.439664e-04 9.704760e-05 3.745656e-03 1.671639e-03 2.542493e-04
```

```
# get predicted classes
predicted_classes <- model_nn %>%
  predict(x_test) %>%
  k_argmax() %>%
  as.integer()
head(predicted_classes)
```

```
## [1] 7 2 1 0 4 1
```

We can extract the misclassification error / accuracy manually:

```
# misclassification error
mean(predicted_classes != g_test)
```

```
## [1] 0.0552
```

```
# accuracy
mean(predicted_classes == g_test)
```

```
## [1] 0.9448
```

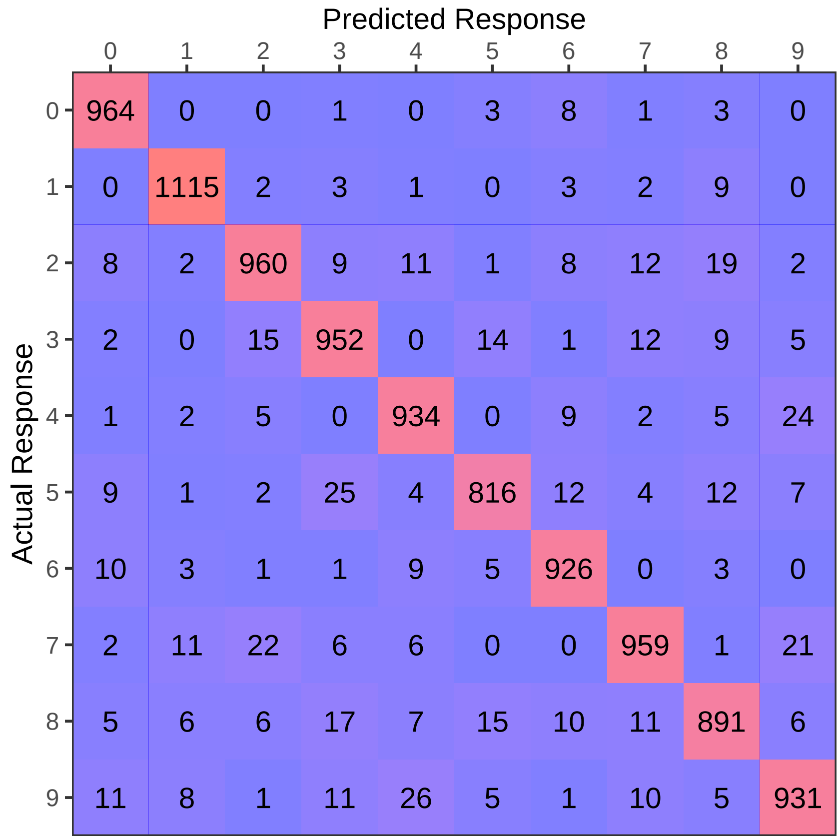
Or we can use a shortcut and call `evaluate`:

```
evaluate(model_nn, x_test, y_test, verbose = FALSE)
```

```
##      loss  accuracy
## 0.1982936 0.9448000
```

In addition to the accuracy / misclassification error, we can take a look at the *confusion matrix* of this classifier using the `plot_confusion_matrix()` function from `stat471`:

```
plot_confusion_matrix(
  predicted_responses = predicted_classes,
  actual_response = g_test
)
```



Let's take a look at an 8 that was misclassified as a 2:

```
misclassifications <- which(predicted_classes == 2 & g_test == 8)
idx <- misclassifications[1]
plot_grayscale(image_array = x_test[idx, , , ], label = "An 8 that was classified as a 2 :(")
```