Unit 5 Lecture 3: Convolutional Neural Networks

November 21, 2023

In this R demo, we'll be fitting convolutional 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()

# 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)
```

Next, we define a convolutional neural network model with two convolutional layers, one max pooling layer, and one dense layer.

```
model_cnn <- keras_model_sequential() |>
  layer_conv_2d(filters = 32, kernel_size = c(3,3), activation = 'relu',
                input_shape = c(img_rows, img_cols, 1)) |>
  layer_conv_2d(filters = 64, kernel_size = c(3,3), activation = 'relu') |>
  layer_max_pooling_2d(pool_size = c(2, 2)) |>
  layer_dropout(rate = 0.25) |>
  layer_flatten() |>
```

```
  layer_dense(units = 128, activation = 'relu') |>
  layer_dropout(rate = 0.5) |>
  layer_dense(units = num_classes, activation = 'softmax')
```

Let's print the summary of this neural network:

```
summary(model_cnn)
```

```
## Model: "sequential"
## _________________________________________________________________________
##  Layer (type)                       Output Shape                    Param #
## =========================================================================
##  conv2d_1 (Conv2D)                  (None, 26, 26, 32)              320
##  conv2d (Conv2D)                    (None, 24, 24, 64)              18496
##  max_pooling2d (MaxPooling2D)       (None, 12, 12, 64)              0
##  dropout_1 (Dropout)                (None, 12, 12, 64)              0
##  flatten (Flatten)                  (None, 9216)                    0
##  dense_1 (Dense)                    (None, 128)                     1179776
##  dropout (Dropout)                  (None, 128)                     0
##  dense (Dense)                      (None, 10)                      1290
## =========================================================================
## Total params: 1199882 (4.58 MB)
## Trainable params: 1199882 (4.58 MB)
## Non-trainable params: 0 (0.00 Byte)
## _________________________________________________________________________
```

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.

```
model_cnn |> compile(
  loss = "categorical_crossentropy",
  optimizer = optimizer_adadelta(),
  metrics = c("accuracy")
)
```

Finally, we can train the model! We use 3 epochs, (mini-)batch size 128, and reserve 20% of our training data for validation.

```
model_cnn |>
  fit(x_train,                  # supply training features
      y_train,                  # supply training responses
      epochs = 3,               # an epoch cycles through all mini-batches
      batch_size = 128,         # mini-batch size
      validation_split = 0.2)   # use 20% of the training data for validation
```

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_cnn, "model_cnn.h5")

# save history
saveRDS(model_cnn$history$history, "model_cnn_hist.RDS")
```

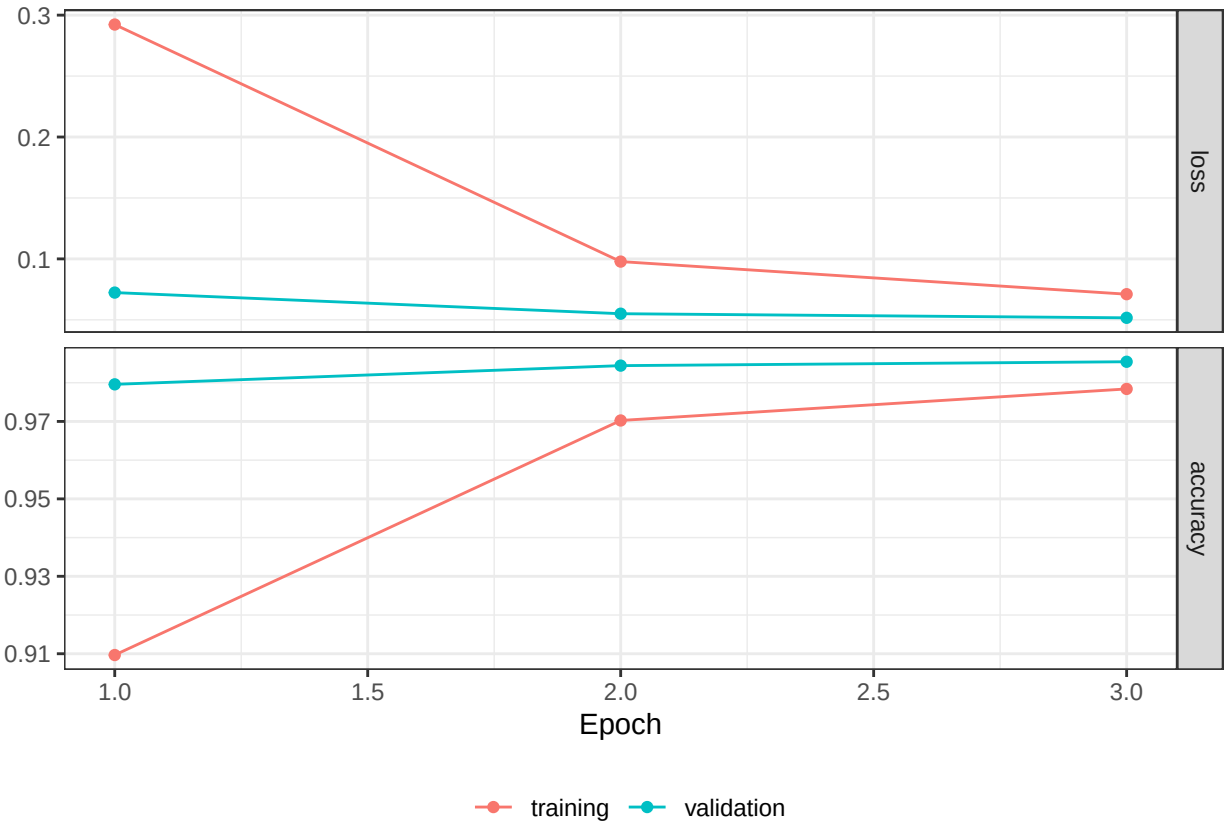
We can then load the model and its history into memory again:

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

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

We can plot the training history using `plot_model_history()` from `deep_learning_helpers.R`:

```r
plot_model_history(model_cnn_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()`:

```r
# get fitted probabilities
model_cnn |> predict(x_test) |> head()
```

```
## 313/313 - 2s - 2s/epoch - 6ms/step
```

```
##              [,1]         [,2]         [,3]         [,4]         [,5]
## [1,] 8.270945e-09 2.402259e-08 5.643851e-07 5.102264e-07 4.791978e-10
## [2,] 5.107172e-08 1.475255e-05 9.999850e-01 5.775118e-09 5.193917e-11
## [3,] 6.900711e-07 9.997985e-01 5.700446e-05 1.166381e-06 1.098542e-05
## [4,] 9.988708e-01 2.350781e-05 5.113242e-04 6.604245e-06 5.909470e-06
## [5,] 5.815071e-08 6.513170e-07 3.464976e-07 1.088213e-07 9.995261e-01
## [6,] 3.514075e-07 9.998734e-01 1.819806e-05 3.111203e-07 6.858348e-06
##              [,6]         [,7]         [,8]         [,9]        [,10]
## [1,] 3.405647e-09 7.080605e-12 9.999985e-01 3.184898e-09 4.077877e-07
## [2,] 1.482013e-10 7.821462e-08 3.875816e-10 3.101941e-09 1.122056e-10
```

```
## [3,] 7.076193e-06 4.956763e-05 6.727723e-05 7.317705e-06 5.616620e-07
## [4,] 4.324708e-05 2.495269e-04 1.634244e-04 2.327995e-05 1.022258e-04
## [5,] 1.637148e-07 3.126416e-07 4.762293e-07 6.184019e-07 4.711363e-04
## [6,] 2.275251e-06 1.150010e-05 8.381818e-05 2.606106e-06 6.744660e-07
```

```
# get predicted classes
predicted_classes = model_cnn |> predict(x_test) |> k_argmax() |> as.integer()
```

```
## 313/313 - 2s - 2s/epoch - 6ms/step
```

```
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] 0.0164
```

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

```
## [1] 0.9836
```

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

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

```
##      loss  accuracy
## 0.0461193 0.9836000
```