Intro to Convolutional Neural Networks

Zachary Blanks

MNIST

For this segment of the course, we are going to work with another classic dataset – MNIST. MNIST is a dataset consisting of 28×28 black-and-white images of the numbers zero through nine. The original motivation for this problem came from the US Postal Service. They wanted to be able to determine a letter's final zipcode without a human intervening due to the increase of mail traffic. Consequently, they turned to machine learning as one way to accomplish this goal. The project succeeded, but it also became a seminal dataset used for neural networks as a way to introduce key concepts, like convolutional networks.

The MNIST data is hosted by Yan LeCun on his website and is commonly used as a benchmark for various deep learning applications, so it's easily accessible and a pretty clean set. Actual data sets, as you might imagine, take quite a bit of work to be ready to run in a neural network. However, that is not the focus of the lecture, and so we will ignore that important aspect of the process.

Let's also restart R because we've been using it for a while and I want to have a fresh start for this portion of the lecture. Once you've done this, we're going to get our data.

```r
library(keras)

mnist = dataset_mnist()

# Get the train-test split that has been defined for us
X_train = mnist$train$x
y_train = mnist$train$y
X_test = mnist$test$x
y_test = mnist$test$y
```

Since our training set has $60,000$ samples, I want to down-sample this to $10,000$ so that we can perform computations more quickly and emphasize the ideas and *not* training time constraints. In the real-world, $60,000$ is fairly small for neural networks, but we have to be able to run our models in a reasonable amount of time on just CPUs.

```r
# Get the indices for our down-sampled data (each label is equally
# represented in the data so we do not need to proportionally down-sample)
set.seed(17)
idx = sample(1:dim(X_train)[1], size = 10000)
X_train = X_train[idx, , ]
y_train = y_train[idx]
```

If you look at the dimensionality of *X_train* right now, you'll see that it is 10000, 28, 28. To work with convolutional networks, the model expects the data to have three dimensions: the length, width, and number of channels for an image. Since our pictures are black-and-white, this implies that they are on a grayscale and thus have only one channel. Therefore, we can re-shape the arrays using the *array_reshape* command. This also has the added benefit of ordering the arrays in a *row_major* format. This is a subtle change, but it will allow us to work more easily with Keras because it's backend, TensorFlow, assumes row-major ordering of data versus column-major which is what R does. These two styles define how data is stored in matrices in the language. For example, suppose you have a vector $\mathbf{v} = (1, 2, 3, 4)$ and want to represent this as a 2×2 matrix. If we stored this as a column-major matrix – like R typically does – we would type

```r
array_reshape(1:4, dim = c(2, 2), order = "F")
```

```
##      [,1] [,2]
## [1,]    1    3
## [2,]    2    4
```

Notice how the matrix was filled by columns; this is what *column-major* ordering means. Conversely if we want *row-major* ordering we would type

```r
array_reshape(1:4, dim = c(2, 2), order = "C")
```

```
##      [,1] [,2]
## [1,]    1    2
## [2,]    3    4
```

As we might expect from the name, *row-major* ordering fills the matrix by rows. This is a subtle matter, but if you work with matrices understanding how they are filled, and therefore stored in memory, is key to getting good performance. Specifically, if your matrices are *column-major* you should iterate over columns and over rows if they are *row-major*.

I claimed that knowing how data is stored can affect performance, but I did not provide evidence. Let me show you with a simple function how this can make a difference.

```r
copy_col = function(x) {
  n = length(x)
  arr = matrix(data = 0., nrow = n, ncol = n)
  for (i in 1:n) {
    arr[, i] = x
  }
  return(NULL)
}

copy_row = function(x) {
  n = length(x)
  arr = matrix(data = 0., nrow = n, ncol = n)
  for (i in 1:n) {
    arr[i, ] = x
  }
  return(NULL)
}

# Let's quickly benchmark the performance difference between these functions
set.seed(17)
x = rnorm(n = 10000)

start_time = Sys.time()
copy_col(x)
```

```
## NULL
```

```r
print(paste("copy_col time:", Sys.time() - start_time))
```

```
## [1] "copy_col time: 1.19549202919006"
```

```r
start_time = Sys.time()
copy_row(x)
```

```
## NULL
```

```r
print(paste("copy_row time:", Sys.time() - start_time))
```

```
## [1] "copy_row time: 2.67292904853821"
```

While this is not a robust way of timing function and also there are more efficient ways of performing the same task using native functions in R, this highlights my ultimate point – knowing how entires are stored in matrices and using this fact can lead to performance gains in your code.

With that brief explanation aside, let's re-shape our arrays appropriately.

```
X_train = array_reshape(X_train, dim = c(dim(X_train), 1))
X_test = array_reshape(X_test, dim = c(dim(X_test), 1))
```

Like last time, we also have to normalize our features. Since they are all on the same scale (since they are pixels) it is not strictly necessary to take this step, but it will improve the speed of convergence. Remember that we can only use the μ and σ from the training set. We NEVER use any information from the test set.

```
mu = mean(X_train)
sigma = sd(X_train)
X_train = (X_train - mu) / sigma
X_test = (X_test - mu) / sigma
```

The final pre-processing step we have to take is to one-hot encode our target vectors, $\mathbf{y}$. What this means in practical terms is that if we have, for example, a target vector like $\mathbf{y} = (1, 2, 3)$ then a one-hot encoded version of $\mathbf{y}$ is

$$\mathbf{Y} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}.$$

This transformation is necessary because the categorical cross-entropy loss function (the loss function we will use) which has the form,

$$\mathcal{L}(\mathbf{Y}, \hat{\mathbf{Y}}) = -\sum_{i=1}^{n}\sum_{j=1}^{C} y_{ij}\log(\hat{y}_{ij})$$

requires the true response, y_{ij} to be either zero or one. This action can be done by using the function, *to_categorical*

```
y_train = to_categorical(y_train)
y_test = to_categorical(y_test)
```

Convolutional Networks

Now that we have the data prepared, our first exercise is going to be making a ConvNet in Keras. I will give you the layer names you need and tell you the experiment settings, but after that it's your turn to make it happen. Specifically, the relevant functions are *layer_conv_2d*, *layer_max_pooling_2d*, and *layer_global_max_pooling_2d*. Specifically, I want the following hyper-parameter settings: * One convolutional layer with 32 filters and 3×3 kernel size and padding = "same" * One pooling layer with default settings * One dense layers with 64 nodes * Standard SGD optimizer * ReLU activation function for each layer * Run the model for 3 epochs with 25% validation split

We will play around with some of these settings, but for now I want to give you more practice with the Keras API so you are prepared for the Kaggle competition at the end of the course.

```r
height = dim(X_train)[2]
width = dim(X_train)[3]
channels = dim(X_train)[4]

model = keras_model_sequential() %>%
  layer_conv_2d(filters = 32, kernel_size = 3, activation = "relu",
                input_shape = c(NULL, height, width, channels),
                padding = "same") %>%
  layer_global_max_pooling_2d() %>%
  layer_dense(units = 64, activation = "relu") %>%
  layer_dense(units = dim(y_train)[2], activation = "softmax")

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

set.seed(17)
res = model %>% fit(
  x = X_train, y = y_train, epochs = 3, verbose = 1, validation_split = 0.25,
  batch_size = 128
)
```

```r
model %>% evaluate(X_test, y_test)
```

```
## $loss
## [1] 2.266964
##
## $acc
## [1] 0.2766
```

Clearly we did not converge to the final model, but I don't want us sitting around for half an hour to get an answer; I just want to introduce the syntax for ConvNets. However, we have also introduced some new hyper-parameters to our model that I want us to get a feel for so we understand what is going on. Let's start with *kernel_size*.

Kernel Size

Effectively what *kernel_size* means is how large of a window should the model use for the convolution operation. For example if *kernel_size = c(3, 3)* then the model is looking at a 3×3 square of pixels for the convolution. To get a feel for this parameter, I will again split you into three groups where we have kernel_sizes of one, seven, and 25. Keep everything else the same.

```r
# Model with kernel_size = 1
model1 = keras_model_sequential() %>%
  layer_conv_2d(filters = 32, kernel_size = 1, activation = "relu",
                input_shape = c(NULL, height, width, channels),
                padding = "same") %>%
  layer_global_max_pooling_2d() %>%
  layer_dense(units = 64, activation = "relu") %>%
  layer_dense(units = dim(y_train)[2], activation = "softmax")
```

```r
model1 %>% compile(
  loss = 'categorical_crossentropy',
  optimizer = optimizer_sgd(),
  metrics = c('accuracy')
)

res1 = model1 %>% fit(
  x = X_train, y = y_train, epochs = 3, verbose = 1, validation_split = 0.25,
  batch_size = 128
)

# Model with kernel_size = 7
model2 = keras_model_sequential() %>%
  layer_conv_2d(filters = 32, kernel_size = 7, activation = "relu",
                input_shape = c(NULL, height, width, channels),
                padding = "same") %>%
  layer_global_max_pooling_2d() %>%
  layer_dense(units = 64, activation = "relu") %>%
  layer_dense(units = dim(y_train)[2], activation = "softmax")

model2 %>% compile(
  loss = 'categorical_crossentropy',
  optimizer = optimizer_sgd(),
  metrics = c('accuracy')
)

res2 = model2 %>% fit(
  x = X_train, y = y_train, epochs = 3, verbose = 1, validation_split = 0.25,
  batch_size = 128
)

# Model with kernel_size = 25
model3 = keras_model_sequential() %>%
  layer_conv_2d(filters = 32, kernel_size = 25, activation = "relu",
                input_shape = c(NULL, height, width, channels),
                padding = "same") %>%
  layer_global_max_pooling_2d() %>%
  layer_dense(units = 64, activation = "relu") %>%
  layer_dense(units = dim(y_train)[2], activation = "softmax")

model3 %>% compile(
  loss = 'categorical_crossentropy',
  optimizer = optimizer_sgd(),
  metrics = c('accuracy')
)

res3 = model3 %>% fit(
  x = X_train, y = y_train, epochs = 3, verbose = 1, validation_split = 0.25,
  batch_size = 128
)
```

```r
model1 %>% evaluate(X_test, y_test)
```

```
## $loss
## [1] 2.302368
##
## $acc
## [1] 0.1135
```

```
model2 %>% evaluate(X_test, y_test)
```

```
## $loss
## [1] 1.983628
##
## $acc
## [1] 0.5492
```

```
model3 %>% evaluate(X_test, y_test)
```

```
## $loss
## [1] 0.4989627
##
## $acc
## [1] 0.9076
```

We will briefly discuss the results that we saw during this experiment. Personally I was most suprised by the large kernel_size result; namely, that by having a very large kernel_size, relative to the image size of 28×28 that this gave a big boost in performance. Typically people have kernel sizes in the range of $\{3, 5, 7, 9\}$ and they also keep these values to be odd for image padding reasons. However, what these results suggest is that it is important, especially with neural networks, to try many hyper-parameter values for your model because sometimes you might have surprising results.

Additional Convolutional Layers

Another implicit hyper-parameter that we had for the model was the number of convolutional layers to use. Let's try adding another layer and see what happens. In particular let's use a kernel_size of five (even though, apparently, it is not the best value) and keep the remaining hyper-parameters the same, just add one additional convolutional layer with 64 filters and a ReLU activation function. Between the convolutional layers, add a pooling layer with the default settings. Run the model for three epochs and report the results.

```
model = keras_model_sequential() %>%
  layer_conv_2d(filters = 32, kernel_size = 7, activation = "relu",
                input_shape = c(NULL, height, width, channels),
                padding = "same") %>%
  layer_max_pooling_2d() %>%
  layer_conv_2d(filters = 32, kernel_size = 7, activation = "relu",
                padding = "same") %>%
  layer_global_max_pooling_2d() %>%
  layer_dense(units = 64, activation = "relu") %>%
  layer_dense(units = dim(y_train)[2], activation = "softmax")

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

res = model %>% fit(
  x = X_train, y = y_train, epochs = 3, verbose = 1, validation_split = 0.25,
  batch_size = 128
)
```

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

```
## $loss
## [1] 0.8880824
##
## $acc
## [1] 0.7139
```

It looks like adding an additional convolutional layer can have a big impact on the model's performance. Does anyone have a guess why that might be?

Transfer Learning

For the last part of the ConvNet portion of the lecture, I just want to introduce and provide some code for a concept known as "transfer learning".

The idea of transfer learning is simple – take a model that has already been pre-trained, and use its weights as a way to help to classify a new data set. The advantage of this approach is that it allows us to leverage the extensive tuning that companies like Google have put into models that they have built without too much added computational cost of our own. When you don't have very much data and the data is somewhat related to the data that the original model was trained on, you can get quite a boost in performance.

To provide a code example, VGG16 is a 16 layer CNN. It is a relatively simple model, at least in comparison to some of the other options available, but it worked well in the ImageNet competition and even was the top submission in 2015. What we're going do at a high level is take the VGG16 model and all its corresponding weights, chop off the fully connected layers, globally pool the final layer, freeze all of the model weights, and just train our own fully connected layer. For this example, we have a dataset where the images are $128 \times 128 \times 3$. The final channel corresponds to having an image that has the red, blue, and green color channels. Implementing the transfer programmaticaly we type

```
# Get the initial VGG16 weights
vgg <- application_vgg16(include_top = FALSE,
                         input_shape = c(128, 128, 3),
                         pooling = 'max')

# Freeze the VGG16 weights
freeze_weights(vgg)

# Now we can define our model
model <- keras_model_sequential() %>%
  vgg %>%
  layer_dense(units = 128, activation = 'relu') %>%
  layer_dropout(rate = 0.5) %>%
  layer_dense(units = 128, activation = "relu") %>%
  layer_dense(units = 1, activation = 'sigmoid')
```

And let's take a look at our model

```
summary(model)
```

```
## ___________________________________________________________________________
## Layer (type)                     Output Shape                  Param #
## ===========================================================================
## vgg16 (Model)                    (None, 512)                   14714688
## ___________________________________________________________________________
## dense_11 (Dense)                 (None, 128)                   65664
## ___________________________________________________________________________
## dropout_1 (Dropout)              (None, 128)                   0
## ___________________________________________________________________________
## dense_12 (Dense)                 (None, 128)                   16512
## ___________________________________________________________________________
## dense_13 (Dense)                 (None, 1)                     129
## ===========================================================================
## Total params: 14,796,993
## Trainable params: 82,305
## Non-trainable params: 14,714,688
## ___________________________________________________________________________
```

For our model, as a consequence of using and freezing the weights from the VGG16 model, we have access to almost 15 million additional parameters! In almost every real-world situation, it is highly recommended that you use pre-trained models. The class CS231 run by Standford provides a useful link for information regarding when and how to use transfer learning for real problems.

This covers everything that I would like to discuss for the ConvNet portion of the lecture. Now we're going to turn to another interesting application of neural networks – natural language processing.