Functional Map of the World

Zachary Blanks

```
library(keras)
library(purrr)
```

Data Preprocessing

Unlike the MNIST data, our real-world data is not as clean (though I have done quite a bit of cleaning already to help speed things up). As a result we have to do more basic data management to work with it. We will follow the same process as the last example, but this time we will also have to split our data into training and testing sets, and re-shape it to be a more manageable image size.

```
get_image <- function(file){
  # Reads in a image from a file and puts it into a matrix form
  #
  # Args:
  #   file = the file to read in
  # Returns:
  #   a matrix representation of an image

  img <- image_load(path = file, target_size = c(128, 128))
  img <- image_to_array(img = img)
  img <- array(data = img, dim = c(1, 128, 128, 3))
  return(img)
}

# Get a vector of all of the image files
img_files <- list.files(path = 'D:/COS2018', full.names = TRUE,
                        recursive = TRUE)

# Define our X matrix
x <- array(dim = c(length(img_files), 128, 128, 3))

# Populate the X matrix
for (i in 1:length(img_files)){
  x[i, , ,] <- get_image(file = img_files[i])
}

# Normalize the data
x <- (x - mean(x)) / sd(x)

# Build the y vector
y <- map(.x = img_files, .f = ~ grepl(pattern = 'nuclear_powerplant',
                                      x = .x))
y <- unlist(y)
y <- as.integer(y)

# Create a train-test split for our data
set.seed(17)
```

```
idx <- sample(1:length(y), size = floor(.75 * length(y)))
x_train <- x[idx, , ,]
y_train <- y[idx]
x_test <- x[-idx, , ,]
y_test <- y[-idx]
```

ConvNet

Now that we have our data, we are ready to build our first convolutional neural network (CNN). We're going to build a relatively small model for the sake of computation time, but like we showed in the last example, with the Keras framework we can easily scale up as we like.

```
conv <- keras_model_sequential()
conv %>%
  layer_conv_2d(filters = 32, kernel_size = c(5, 5),
                activation = 'relu',
                input_shape = c(128, 128, 3)) %>%
  layer_max_pooling_2d() %>%
  layer_conv_2d(filters = 64, kernel_size = c(5, 5),
                activation = 'relu') %>%
  layer_global_max_pooling_2d() %>%
  layer_dense(units = 128, activation = 'relu') %>%
  layer_dropout(rate = 0.5) %>%
  layer_dense(units = 1, activation = 'sigmoid')
```

So there's quite a bit going on in just ten lines of code. Let me break each part down. First, when we want to make convolutional layers, we need to specify the number of filters and the kernel size. Almost always the kernel size will be an odd integer because this makes the math for convolutions work out much nicer. You can also change the stride and padding if you'd like and those are good hyper-parameters to play with, but we won't worry about those at the moment. Second, we normally put max pooling layers after convolutional layers. Finally before you get to your fully connected layers you need to flatten the input in some way. The most common approaches are to just flatten the input as is, do global max pooling like we did, or to do global average pooling. However you must do this because the fully connected layers require 2-d inputs. Finally, I've included a dropout layer for illustrative purposes. You can tune the dropout rate as you'd like, but in the paper which detailed the method, the author found that 0.5 worked pretty well most of the time.

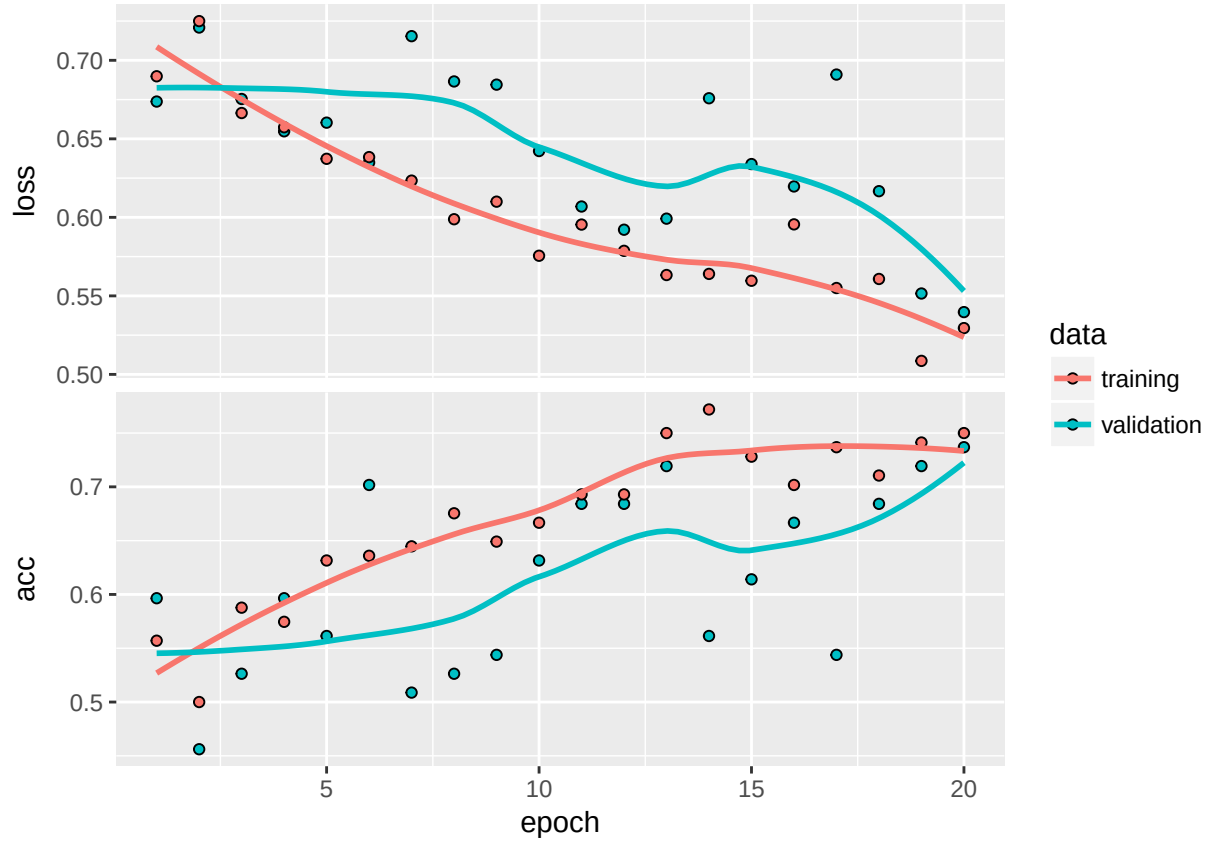
Now we're ready to specify the additional portions of the model and then train it.

```
conv %>% compile(
  loss = 'binary_crossentropy',
  optimizer = optimizer_sgd(),
  metrics = c('accuracy')
)

history <- conv %>% fit(
  x = x_train, y = y_train,
  epochs = 20, batch_size = 32,
  validation_split = 0.2,
  verbose = 0
)
```

Let's take a look at our training history because it highlights some interesting things that can happen when you train neural networks using stochastic gradient descent (SGD) on real data.

```
plot(history)
```



Notice how the validation accuracy bounces around sometimes? This is a consequence of the SGD algorithm works. It does not always guarantee that you will improve on any given epoch, and in fact you may do worse. Nevertheless, on average you will improve and usually the algorithm will converge to a good answer. There are ways to mitigate this bouncing so that you can have a smoother and quicker training time. We can discuss these if there is time and if you would like to know more.

Now let's evaluate our model's performance

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

```
## $loss
## [1] 0.4803781
##
## $acc
## [1] 0.8541667
```

That's not bad, but can we do better? There are many ways to try to improve our performance. We could decrease the learning rate and increase the number of epochs, change the optimization algorithm, fine tune the number of filters, and many other options. Another approach is try what is known as transfer learning. For the purposes of this class, this will take longer than acceptable and won't work very well given that the images of most commonly available weights were trained on a very different different set of images, but I would still like to show you the code of how you can do this so that you can run this idea yourself if you'd like.

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 (like us in this instance) and the data is somewhat related to the data that the original model was trained on, you can get quite a boost. In our case it's a bit of a stretch that a nuclear power plant looks like the targets that were used to tune the weights for VGG16.

As an overview, 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. 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.

```
# 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 = 1, activation = 'sigmoid')
```

Overall we're doing the same thing we did in our last model in terms of the fully connected layer. Let's take a look at our model summary so that we can see what we're actually doing.

```
summary(model)
```

```
## _________________________________________________________________________
## Layer (type)                 Output Shape              Param #
## =========================================================================
## vgg16 (Model)                (None, 512)               14714688
## _________________________________________________________________________
## dense_3 (Dense)              (None, 128)               65664
## _________________________________________________________________________
## dropout_2 (Dropout)          (None, 128)               0
## _________________________________________________________________________
## dense_4 (Dense)              (None, 1)                 129
## =========================================================================
## Total params: 14,780,481
## Trainable params: 65,793
## Non-trainable params: 14,714,688
## _________________________________________________________________________
```

Indeed it seems that we are only training the weights for our fully connected layer and the final output layer. Now we're ready to fit our model.

```
model %>% compile(
  loss = 'binary_crossentropy',
  optimizer = optimizer_sgd(),
```

```
  metrics = c('accuracy')
)

model %>% fit(
  x = x_train, y = y_train,
  epochs = 10, batch_size = 16,
  validation_split = 0.2
)
```

You can run the snippet above yourself, but at least when I tried it, the model did not perform well (likely because of how different the data sets are from one another). However, before we end this portion of the lecture, I'd like to show you how to run models with graphical processing units (GPUs). All of the models we've run thus far have involved only CPUs, but as I mentioned at the beginning of the lecture, GPUs can speed up your computation time tremendously.

GPU Models

Because GPUs can speed up the computation time so much, Keras has a pre-built function which allows us to easily leverage GPU computation with only a small amount of code. If your system only has one GPU, then you don't have to do anything. TensorFlow will automatically detect that you have one GPU and utilize it for the computations. However, what if you have more than one GPU? If this is the case then we can easily scale up the code to allow up to eight GPUs. I'll show you a small code snippet that allows you to do this.

```
# We already have models that we've defined above, so we'll just
# use those
parallel_model <- multi_gpu_model(model = conv, gpus = 4)
```

And that's how easy it is to make a model which works with multiple GPUs! There are additional intricacies to utilizing a multi-GPU model that can discussed more if you'd like, but with that single line of code, you've done about 95% of the work required to run a multiple GPU model. What that snippet does, at a high level, is replicate your model on each of the GPUs you have available and then split your batch by by the number of GPUs. So, for example, if your batch size is 128 and you have two GPUs, then each GPU would get 64 samples to work with for a given step.

With the tools we've worked on above, you should have everything you need to jump right into your own neural network project involving image data!