Convolutional Neural Networks

Convolutional Neural Networks

Now we're ready to build some actual image recognition models using convolutional neural networks. Note that we're not going to be using black and white pictures any more. CNNs can leverage RGB color information to improve performance. We also will not be flattening our data since CNNs are designed to take advantage of the 2D structure of images. So each example will be a `64x64x3`, where the last dimensions are the red, green, blue (RGB) color channels.

```r
library(keras)
load(url("https://www.dropbox.com/s/ybyqvdb2csh1wq2/cat_dog_data.RData?dl=1"))
dim(x_train)
```

```
## [1] 2000   64   64    3
```

The convolutional neural network will take each image that is a `64x64x3` tensor and perform a 2-dimension convolution across the surface of the image. In fact it will perform many quasi-independent convolutions, each of which is commonly referred to as a *filter*, because it scans local patches of the image for patterns while discarding the rest of the image. First, we are going to initialize the model and set up a simple convolutional/max-pooling block:

```r
model <- keras_model_sequential()
model %>%
  ## First the convolutional block with:
  # - 16 filters
  # - Looking at 3x3 patches of the image at a time (kernal_size)
  # - Each convolution is followed by a rectified linear activation (relu)
  # - We will zero-pad the image to keep the output tensor the same size as the input tensor
  layer_conv_2d(filters = 16,
                kernel_size = c(3,3),
                activation = 'relu',
                input_shape = c(64,64,3),
                padding = "same") %>%
  layer_max_pooling_2d(pool_size = c(2, 2))
```

This sets up the first layer which is a convoluational layer with 16 filters, each of which has a `3x3` receptive field. Each of these 16 filters is going to look at the entire image `3x3 = 9` pixels at time and perform the weighted sum of these 9 pixels followed by the `relu` activation. The pooling layer is going to down sample the result by a factor of 2, since it pools over 2 pixels in both the x and y directions. Let's look at the model summary:

```r
model %>% summary()
```

```
## ___________________________________________________________________________
## Layer (type)                     Output Shape                  Param #
## ===========================================================================
## conv2d_1 (Conv2D)                (None, 64, 64, 16)            448
## ___________________________________________________________________________
## max_pooling2d_1 (MaxPooling2D)   (None, 32, 32, 16)            0
## ===========================================================================
## Total params: 448
## Trainable params: 448
## Non-trainable params: 0
## ___________________________________________________________________________
```

We see that the output shape of the first convolutional layer takes our `64x64x3` input and produces an output tensor with volume `64x64x16`, where the last dimension is the number of filters we specified. Next, the max pooling layer *down samples* this volume by a factor of two by simply taking the maximum activation over `2x2` grids.

Let's finish off the model by flattening the result of the max pool layer (which discards the spatial information) and feeding the result through a dense sigmoid layer:

```
model %>%
  layer_flatten() %>%
  layer_dense(units = 1, activation = 'sigmoid')

## Look at the full model ##
model %>% summary()
```

```
## ___________________________________________________________________________
## Layer (type)                     Output Shape                  Param #
## ===========================================================================
## conv2d_1 (Conv2D)                (None, 64, 64, 16)            448
## ___________________________________________________________________________
## max_pooling2d_1 (MaxPooling2D)   (None, 32, 32, 16)            0
## ___________________________________________________________________________
## flatten_1 (Flatten)              (None, 16384)                 0
## ___________________________________________________________________________
## dense_1 (Dense)                  (None, 1)                     16385
## ===========================================================================
## Total params: 16,833
## Trainable params: 16,833
## Non-trainable params: 0
## ___________________________________________________________________________
```

Now let's compile and train for 10 epochs:

```
model %>%
  compile(
    loss = 'binary_crossentropy',
    optimizer = 'adam',
    metrics = c('accuracy')
  )  %>%
  fit(x_train, y_train,
    batch_size = 128,
    epochs = 10,
    verbose = 1,
    validation_data = list(x_test, y_test)
)
```

Let's evaluate on the test set:

```
results <- model %>% evaluate(x_test,y_test)
print(paste("Validation loss:", results$loss,sep=' '))
```

```
## [1] "Validation loss: 0.610514649391174"
```

```
print(paste("Validation accuracy:", results$acc,sep=' '))
```

```
## [1] "Validation accuracy: 0.653"
```

Already we should be doing much better than any of our previous models, and this model starts making progress much faster than our MLP (which took 100s - 1000s of epochs and overfit pretty badly). This is because CNNs use the properites of images (2D spatial structure) to prevent overfitting.

In class assignment

Now it's your turn again to improve upon this model. Keep track of your best model and validation accuracy, the person/team with the best result will win a prize! Here are some suggestions:

- 2 convolutional layers before the max pooling layer
- More convolutional/maxpooling blocks. Be careful though, this can get computational expensive quickly.
- A dense layer after the flatten layer but before the sigmoid layer
- Dropout at various places
- Regularization

Put your additions in here to improve the model:

```r
model <- keras_model_sequential()
model %>%
  ## Your improvments here

  layer_dense(units = 1, activation = 'sigmoid') %>%
  compile(
    loss = 'binary_crossentropy',
    optimizer = 'adam',
    metrics = c('accuracy')
  )  %>%
  fit(x_train, y_train,
    batch_size = 128,
    epochs = 25,
    verbose = 1,
    validation_data = list(x_test, y_test)
)
```

Bonus

While you are fitting your models, think about how we can perform a convolution using just matrix multiplications.