The Carcinoma Dataset

Presenters: Jay Barber and Dan Pagendam

The Carcinoma Dataset

- Exercise developed by Chris Wikle and Dan Pagendam (2019).
- This dataset is taken from Janowczyk, A. and Madabhushi, A. (2016). Deep learning for digital pathology image analysis: A comprehensive tutorial with selected use cases. Journal of Pathology Informatics 7:29.
- https://www.ncbi.nlm.nih.gov/pubmed/27563488.
- The full dataset can be downloaded from http://gleason.case.edu/webdata/jpi-dl-tutorial/IDC_regular_ps50_idx5.zip

The Carcinoma Dataset

- The original dataset consisted of 162 whole mount slide images of Breast Cancer (BCa) specimens scanned at 40x
- From that, 277,524 patches of size 50 x 50 pixels were extracted (198,738 IDC negative and 78,786 IDC positive).
- In this exercise we use a very small subset of this dataset consisting of 2000 training images and 500 validation images that were randomly sampled from the larger set.

The Carcinoma Dataset

Carcinoma Neg.

Carcinoma Pos.

Required Packages

- Install the following:

```
if(Sys.info()['sysname'] == "Linux")
{
  #Install Linux dependencies for those using Google Colab
  system('sudo apt-get install -y libtiff5-dev', intern=TRUE)
  system('sudo apt-get install -y libfftw3-dev', intern=TRUE)
}
install.packages(c("pbapply", "keras", "raster"),
                 repos = "http://cran.us.r-project.org",
                 quiet = TRUE)
```

Required Packages

- Install the following (continued):

```
remotes::install_github("dpagendam/deepLearningRshort")
install.packages("BiocManager",
                 repos = "http://cran.us.r-project.org",
                 quiet = TRUE)
BiocManager::install("EBImage")
library(pbapply)
library(keras)
library(raster)
library(deepLearningRshort)
library(EBImage)
```

Load the Carcinoma Data

- The images are already 50x50 pixels, so no resizing here.
- You can convert the images to grayscale if desired.

```
width <- 50
height <- 50
grayScale <- FALSE
packageDataDir = system.file("extdata", package="deepLearningRshort")
trainData <- extract_feature(paste0(packageDataDir, "/carcinoma/train/"),
                             width, height, grayScale, TRUE)
```

```
## [1] "Start processing 1001 images"
```

```
testData <- extract_feature(paste0(packageDataDir, "/carcinoma/test/"),
                            width, height, grayScale, TRUE)
```

```
## [1] "Start processing 500 images"
```

Data Formatting for a CNN in Keras

- Remember, all of our data needs to be in a tensor where the first dimension corresponds to the observations/samples.

```
numInputChannels <- 3
train_array <- t(trainData$X)
dim(train_array) <- c(width, height, numInputChannels,
                      nrow(trainData$X))
train_array <- aperm(train_array, c(4,1,2,3))

test_array <- t(testData$X)
dim(test_array) <- c(width, height, numInputChannels,
                     nrow(testData$X))
test_array <- aperm(test_array, c(4,1,2,3))
```

Building a CNN Model

- A standard architecture for a CNN are convolutional, batch normalisation, and max-pooling layers repeated in series.

```
model <- keras_model_sequential()

model %>% layer_conv_2d(kernel_size = c(5,5), filters = 8,
          strides = 1, activation = "relu", padding = "same",
          input_shape = c(width, height, numInputChannels),
          data_format="channels_last") %>%
layer_batch_normalization() %>%
layer_max_pooling_2d(pool_size = c(2,2), padding = "same")

model %>% layer_conv_2d(kernel_size = c(5,5), filters = 8,
          strides = 1, activation = "relu", padding = "same",
          input_shape = c(width, height, numInputChannels),
          data_format="channels_last") %>%
layer_batch_normalization() %>%
layer_max_pooling_2d(pool_size = c(2,2), padding = "same")
```

Building a CNN Model (Continued)

- After the deep convolutional parts, the tensors are flattened (i.e. a 2D image to a vector).
- The flattened features are then put through a FFNN with a single node for binary classification.

```
model %>% layer_conv_2d(kernel_size = c(5,5), filters = 8,
          strides = 1, activation = "relu", padding = "same",
          input_shape = c(width, height, numInputChannels),
          data_format="channels_last") %>%
layer_batch_normalization() %>%
layer_max_pooling_2d(pool_size = c(2,2), padding = "same") %>%
layer_flatten() %>%
layer_dense(units = 8, activation = "relu") %>%
layer_dense(units = 8, activation = "relu") %>%
layer_dense(units = 8, activation = "relu") %>%
layer_dense(units = 1, activation = "sigmoid")
```

Compiling the model

- For binary classification, the standard loss function is binary cross-entropy.

```
model %>% compile(
  loss = 'binary_crossentropy',
  optimizer = optimizer_rmsprop(learning_rate = 0.0001),
  metrics = c('accuracy')
)
```

Training the Model

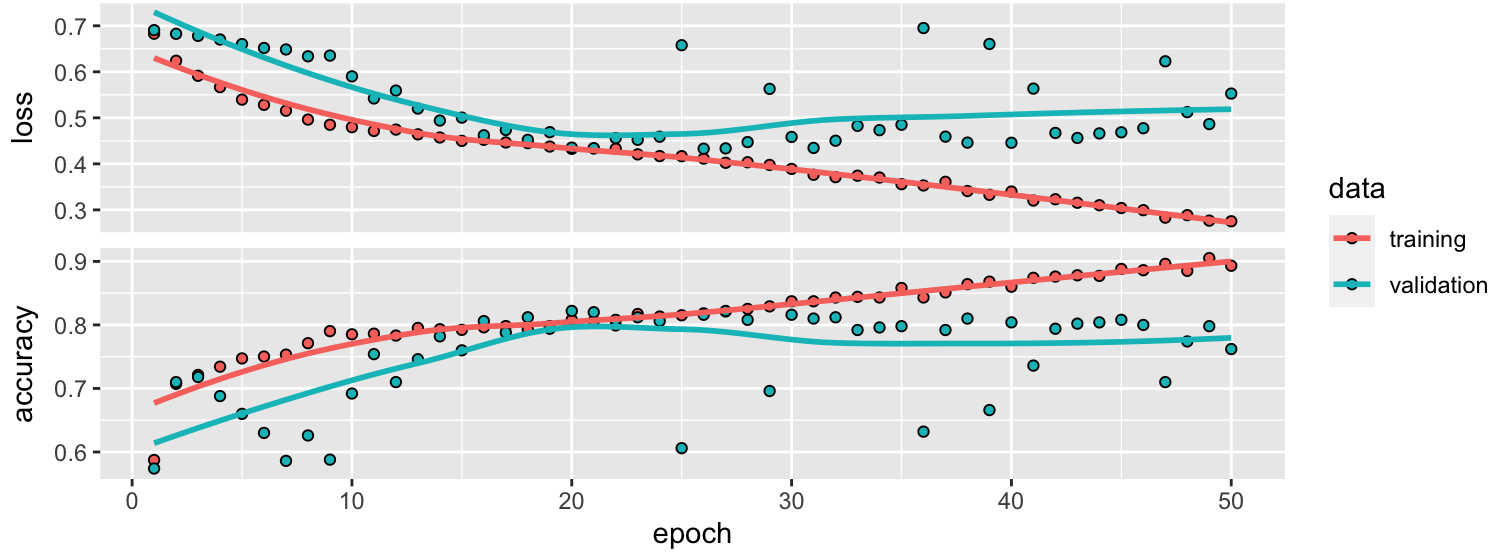
- Let's train the model for 50 epochs and see whether the model overfits.

```
history <- model %>% fit(
  x = train_array, y = as.numeric(trainData$y),
  epochs = 50, batch_size = 32, validation_data =
    list(test_array,as.numeric(testData$y))
)
```

Training the Model

- Let's train the model for 50 epochs and see whether the model overfits.

```
plot(history)
```



Training the Model

- It looks as though the prediction accuracy peaks at around 80% on the validation data.
- It is clear that the loss for the training and validation data start to diverge by around 20 epochs, indicating overfitting.
- We'll re-initialise the model and train it again, but this time specify only 20 epochs.

Rebuilding the CNN Model

```
model <- keras_model_sequential()

model %>% layer_conv_2d(kernel_size = c(5,5), filters = 8,
          strides = 1, activation = "relu", padding = "same",
          input_shape = c(width, height, numInputChannels),
          data_format="channels_last") %>%
layer_batch_normalization() %>%
layer_max_pooling_2d(pool_size = c(2,2), padding = "same")

model %>% layer_conv_2d(kernel_size = c(5,5), filters = 8,
          strides = 1, activation = "relu", padding = "same",
          input_shape = c(width, height, numInputChannels),
          data_format="channels_last") %>%
layer_batch_normalization() %>%
layer_max_pooling_2d(pool_size = c(2,2), padding = "same")
```

Rebuilding the CNN Model (Continued)

- After the deep convolutional parts, the tensors are flattened (i.e. a 2D image to a vector).
- The flattened features are then put through a FFNN with a single node for binary classification.

```
model %>% layer_conv_2d(kernel_size = c(5,5), filters = 8,
          strides = 1, activation = "relu", padding = "same",
          input_shape = c(width, height, numInputChannels),
          data_format="channels_last") %>%
layer_batch_normalization() %>%
layer_max_pooling_2d(pool_size = c(2,2), padding = "same") %>%
layer_flatten() %>%
layer_dense(units = 8, activation = "relu") %>%
layer_dense(units = 8, activation = "relu") %>%
layer_dense(units = 8, activation = "relu") %>%
layer_dense(units = 1, activation = "sigmoid")
```

Recompiling the model

- For binary classification, the standard loss function is binary cross-entropy.

```
model %>% compile(
  loss = 'binary_crossentropy',
  optimizer = optimizer_rmsprop(learning_rate = 0.0001),
  metrics = c('accuracy')
)
```

Retraining the Model

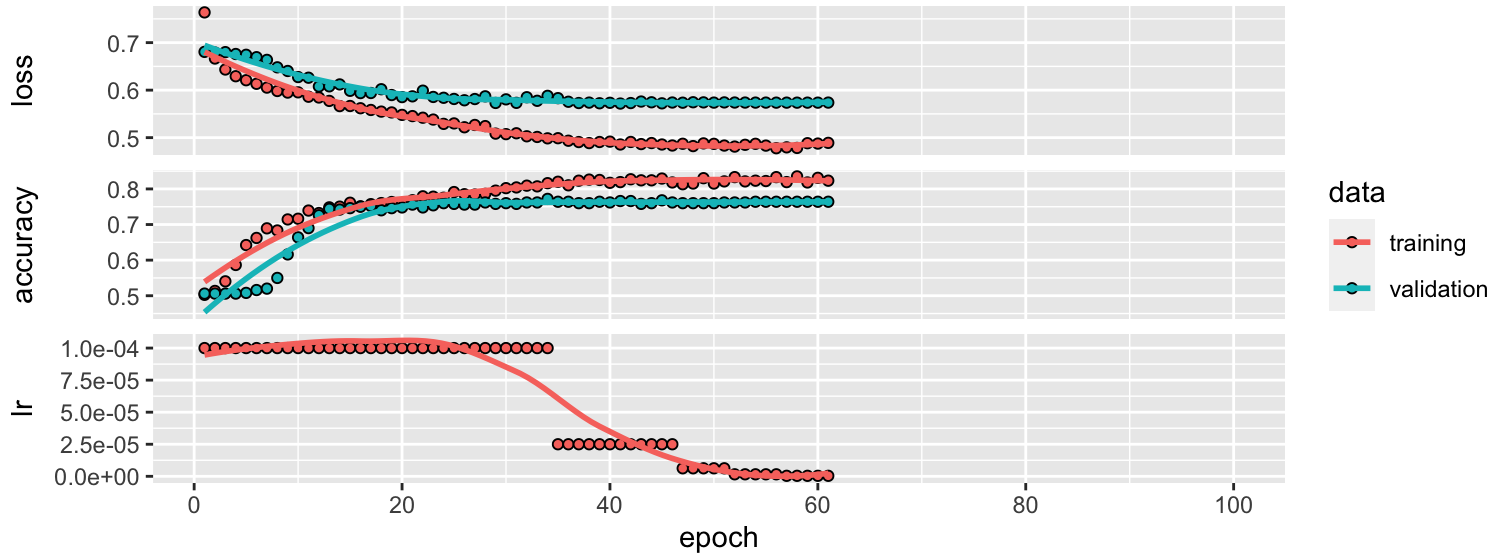
- Let's train the model with "early stopping".

```
history <- model %>% fit(
  x = train_array, y = as.numeric(trainData$y),
  epochs = 100, batch_size = 32, validation_data =
    list(test_array,as.numeric(testData$y)),
  callbacks = list(callback_early_stopping(
            monitor = "val_loss", min_delta = 0, patience = 20),
            callback_model_checkpoint(filepath = "carcinoma.hd5",
            save_best_only = TRUE, save_weights_only = FALSE),
            callback_reduce_lr_on_plateau(monitor = "val_loss",
           factor = 0.25, patience = 5))
)

bestModel <- load_model_hdf5(filepath = "carcinoma.hd5")
```

Retraining the Model

```
plot(history)
```



Retraining the Model

```
# We can obtain the outputs of the model (sigmoid activation)
probabilities <- predict(model, test_array)
# We obtain the predicted classes for the test/validation dataset
predictions <-  as.numeric(predict(model, test_array) %>%
                             `>`(0.5) %>% k_cast("int32"))
truth <- testData$y
propCorrect <- sum(predictions == truth)/length(truth)
print(propCorrect)
```

```
## [1] 0.764
```

Checking Model Performance

Some things to try

- Does converting the images to grayscale have any impact on predictive performance?
- How do your results change if you reduce the number of filters in the convolutional layers or the number of units in the dense layers? Does the model overfit more dramatically (i.e. the history plot) when the model has more parameters?
- Where could you incorporate dropout layers into the model?