Carcinoma identification exercise

Jordan Richards

2023-01-27

Dataset

- Exercise developed by Chris Wikle and Dan Pagendam (2019).
- Dataset 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

Dataset

- The original dataset consisted of 162 whole mount slide images of BreastCancer (BCa) specimens scanned at 40x resolution
- Of those slides, 277,524 patches of size 50 x 50 pixels were extracted (198,738 IDC negative and 78,786 IDC positive).
- Our goal is to train a CNN to identify if carcinoma is present within a patch
- Due to computational constraints, we will use only a small subset of the data - 1001 training images and 500 validation images that were randomly sampled from the larger set.

Required packages

We will be using some functions and the images from Dan's github directory, https://github.com/dpagendam/deepLearningRshort

```
remotes::install_github("dpagendam/deepLearningRshort")
```

```
## Skipping install of 'deepLearningRshort' from a github r
##   Use `force = TRUE` to force installation
```

```
library(pbapply)
library(keras)
library(raster)
```

```
## Loading required package: sp
```

```
library(deepLearningRshort)
library(EBImage)
```

```
##
## Attaching package: 'EBImage'

## The following objects are masked from 'package:raster':
```

Load dataset

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

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

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

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

- All images are 50 x 50 pixals, so no need to resize them
- Images can be greyscale if desired

Formatting for Keras

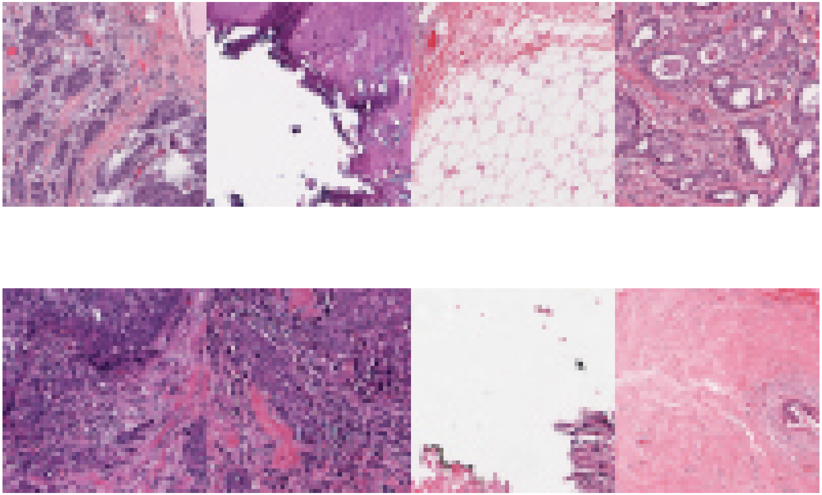
- All data needs to be in a tensor where the first dimension corresponds to the observations/samples.
- Each channel represents an intensity on an RGB scale

```r
if(greyScale) numInputChannels <- 1 else  numInputChannels
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))
```

Plot

Top: positive. Bottom: negative.



Building a CNN Model

We use a fairly typical architecture for a CNN with convolutional, batch normalisation, and max-pooling layers repeated in series.

```r
library(keras)
input.lay <- layer_input(shape = dim(train_array)[-1],
      name = 'input_layer')

model <- input.lay %>% layer_conv_2d(
  kernel_size = c(5,5),
          filters = 8,
          strides = 1, activation = "relu",
          padding = "same",
          data_format="channels_last") %>%
layer_batch_normalization() %>%
layer_max_pooling_2d(pool_size = c(2,2),
                     padding = "same")
```

Building a CNN model

```r
model <- model %>% layer_conv_2d(
  kernel_size = c(5,5),
    filters = 8,
     strides = 1, activation = "relu",
      padding = "same",
          data_format="channels_last") %>%
layer_batch_normalization() %>%
layer_max_pooling_2d(pool_size = c(2,2),
                     padding = "same")  %>%
layer_conv_2d(kernel_size = c(5,5),
              filters = 8,
          strides = 1, activation = "relu",
          padding = "same",
          data_format="channels_last") %>%
  layer_batch_normalization() %>%
layer_max_pooling_2d(pool_size = c(2,2),
                     padding = "same")
```

Building a CNN Model

- After the convolutional layers, the tensors are flattened into a vector
- These features are then put through a dense FFNN with a single node for binary classification.

```
model <- model %>%layer_flatten() %>%
layer_dense(units = 8, activation = "relu") %>%
layer_dense(units = 8, activation = "relu") %>%
layer_dense(units = 8, activation = "relu")
output.lay <- model %>% layer_dense(units = 1,
                      activation = "sigmoid")

  model <- keras_model(
    inputs = c(input.lay),
    outputs = c(output.lay)
  )
```

Compile

- Compile model with standard loss - binary cross entropy or Bernoulli nll
- We will mix things up and use ADAM this time, rather than rmsprop
- We will also ask Keras to print an extra evaluation metric, i.e., accuracy

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

Summary

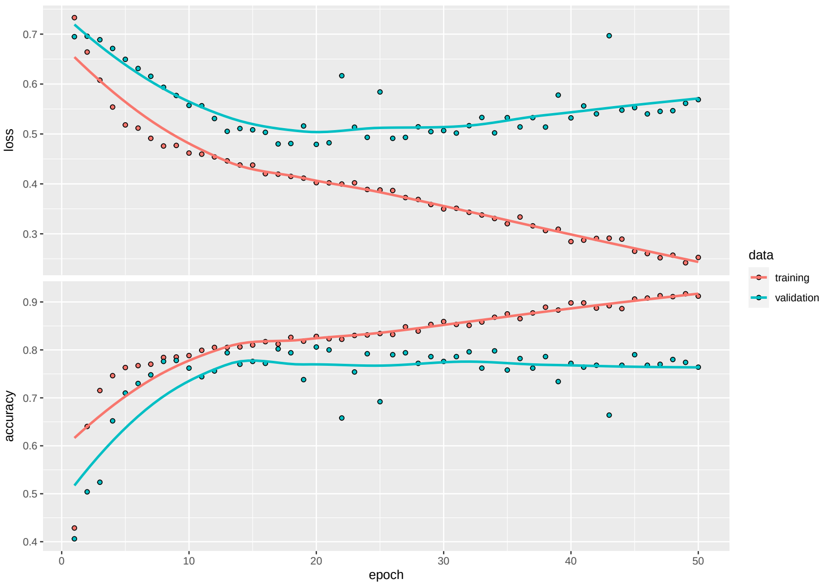
```
summary(model)
```

Model training

Let's train the model

```r
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))
)
```

History



loss
accuracy
epoch
0.7
0.6
0.5
0.4
0.3
0.9
0.8
0.7
0.6
0.5
0.4
0
10
20
30
40
50
data
training
validation

Overfitting

- We can see that the model begins to overfit quite quickly.
- Ideally we want to extract the model with the lowest validation loss, but by default Keras just saves the state of the model at the last epoch
- We can change that with a callback called checkpoints

```r
checkpoint <- callback_model_checkpoint(
  filepath = "model_weights",
  monitor = "val_loss",
  verbose = 0,
  save_best_only = TRUE,
  save_weights_only = TRUE,
  mode = "min",save_freq = "epoch")
```

Overfitting

We now retrain the model with the checkpoint and we can extract the best predictive model at the end of training

- Note that the model weights are saved to the R objects denoting the layers. We need to recreate the model to train from scratch

Checkpoints

- We will also use the learning rate reduction callback from the previous exercise

```r
history <- model %>% fit(
  x = train_array, y = as.numeric(trainData$y),
  callback = list(checkpoint,
      callback_reduce_lr_on_plateau(
        monitor = "val_loss",
      factor = 0.25, patience = 5)),
  epochs = 50, batch_size = 32, validation_data =
    list(test_array,as.numeric(testData$y))
)
```

Checkpoints

```
#Then load the saved weights
model <- load_model_weights_tf(model,
                               filepath="model_weights")
```

Now the final fitted model minimises the validation loss.

- Early-stopping is an alternative. You can set training to stop if the (validation) loss has not decreased. This is less computationally expensive than checkpointing, but may not be optimal for particularly volatile loss functions, e.g., complex likelihoods.

Checking performance

- We can obtain the outputs of the model (sigmoid activation)

```r
probabilities <- predict(model, test_array)
```

- We can then obtain the predicted classes for the test/validation dataset

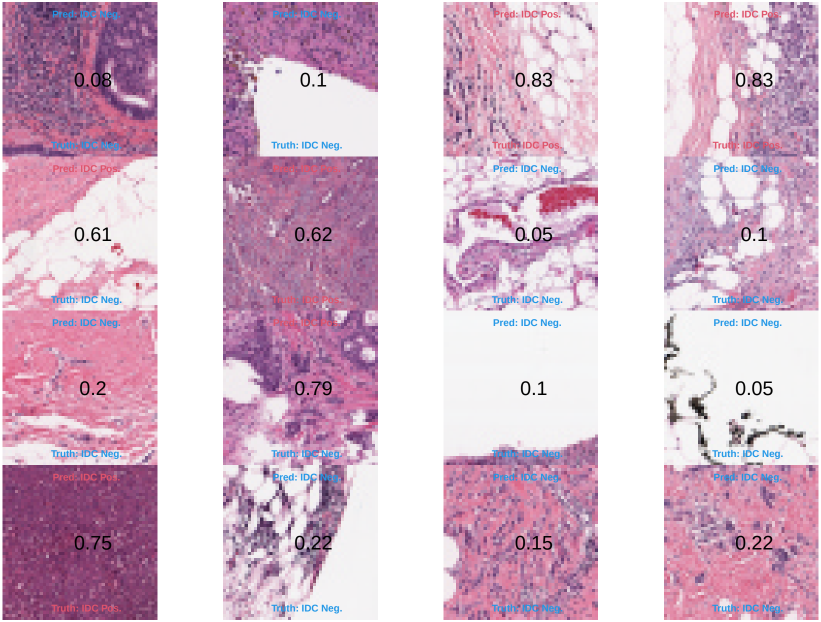
```r
predictions <- as.numeric(probabilities %>%
`>`(0.5) %>% k_cast("int32"))
```

- and check the accuracy of the predictions

```r
truth <- testData$y
propCorrect <- sum(predictions == truth)/length(truth)
print(propCorrect)
```

```
## [1] 0.792
```

Checking performance



Extensions

- 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?