# Final Project

Statistical Learning with Deep Artificial Neural Networks

Alexander J Ohrt

30. mai. 2022

## Contents

# Introduction

In this final project we will build an image recognition application using R. The work is heavily influenced by two blog posts from the blog *For-loops and piep kicks* (Blog post one and Blog post two). The user of the web-based application will be able to upload a photo and receive predictions on what is shown in the photo, based on a machine learning model. In this case, the model is trained on 50 different bird species, which means that this app can be used to predict birds among these species. If any other file is uploaded, the application will give strange results. Notice that if the model makes a prediction with certainty probability of under 45%, it outputs a warning to the user.

The machine learning (ML) model used in this case is a convolutional deep neural network (CNN). Models of this sort are leading in the areas of image recognition and image classification. We use a pre-trained model, with a demonstrated performance on ImageNet. More specifically, we load the Xception network, with weights pre-trained on the ImageNet dataset. This network is used as a baseline model and we add a final hidden layer, which will be fine-tuned on our dataset. In this way we can use **transfer learning** to make a classification model - we use the pre-trained Xception network, which, even though it is not trained in our specific dataset, will probably be able to extract some relevant features from our data. The fine-tuning of our model is done by freezing the weights of the Xception network, meaning that they will not be changed when training, and optimizing (at best) the weights to the final hidden layer and to the output layer.

# Part 1 - Implementing a CNN

## Downloading Data

The first and most critical step in the process is gathering data for training, validating and testing the ML model. In this case, since we will work with bird species, we download a dataset containing 400 bird species from Kaggle. This is a very high quality dataset where there is only one bird in each image and the bird typically takes up at least 50% of the pixels in the image. From the zip file, we extract the first 50 folders of train and test images, which correspond to 50 different species of birds. We save the folders in directories called "train" and "test" respectively. I did this manually using the GUI in my OS, as I found that to be the simplest procedure. Notice that the specific data I have used is available in the parent directory of this report. If you would like to, you can use a larger or smaller amount of species or sample the species randomly instead of simply choosing the first 50. Next, we check that all the bird species are equal in both the directories, to be sure that we don't generate any problems from the get-go. This is done using code that is shown further below. Thus, we have our two (high-quality) datasets and are able to begin working with some models.

## Setting Parameters and Importing Images

After having the data ready, we read all bird names into R by listing the subfolder names of the train directory. Moreover, we check that the subfolder names are the same when reading from the train and test directories, which is our check that the same 50 species are correctly selected from the downloaded zip file. In our case, this is true. We save the label names to disk for convenience, since these names will be used when building the web-app later.

Next we define the image width and height, defining the target size of the images. We also define the number of color channels, which is 3 in this case, since the images are RGB. Moreover, we define the batch size and the epochs, which will be used during the fitting of the model.

```
global.seed <- 2022 # Use this as the seed everywhere.
set.seed(global.seed)
path.train <- "train/"
path.test <- "test/"
label.list <- dir(path.train) # List the bird species names in the train folder.
output.n <- length(label.list) # As we can see, we have 50 different birds in train images.
length(dir(path.test)) # The same is the case for the testing images, as seen below.
```

```
#> [1] 50
```

```
all(label.list == dir(path.test))
```

```
#> [1] TRUE
```

```
save(label.list, file="label_list.RData") # Save the list of names on disk for later.

width <- height <- 224 # This is the original size of the images.
target.size <- c(width, height)
rgb <- 3 # Color channels.

# Set batch size and number of epochs for training later.
batch_size <- 32
epochs <- 6
```

In order to work efficiently with the images, we use image data generators from Keras. In these generators we also define the preprocessing of the images and make a validation split from the training data. We simply rescale the images to have pixel values within [0,1] and set the validation split to 0.2. We do not apply any further image augmentation techniques. Then we use the `flow_images_from_directory()` function to automatically import batches of images. We do this for all three data sets - training, validation and testing.

```
# Make generator for training and validation data.
train.data.gen <- image_data_generator(rescale = 1/255,
                                       validation_split = 0.2)
```

```
#> Loaded Tensorflow version 2.7.2
```

```
# Load batches of training data using the generator.
train.images <- flow_images_from_directory(path.train,
                                           train.data.gen,
                                           subset = "training",
                                           target_size = target.size,
                                           class_mode = "categorical",
                                           shuffle=F,
                                           classes = label.list,
                                           seed = global.seed,
                                           batch_size = batch_size)

# Load batches of validation data using the generator.
validation.images <- flow_images_from_directory(path.train,
                                                train.data.gen,
                                                subset = "validation",
                                                target_size = target.size,
                                                class_mode = "categorical",
                                                classes = label.list,
                                                seed = global.seed,
                                                batch_size = batch_size)

# Make generator for testing data. Do not want the validation_split here.
test.data.gen <- image_data_generator(rescale = 1/255)

# Load batches of testing data using the generator.
test.images <- flow_images_from_directory(path.test,
                                          test.data.gen,
                                          target_size = target.size,
                                          class_mode = "categorical",
                                          classes = label.list,
                                          shuffle = F,
                                          seed = global.seed,
                                          batch_size = 1) # Set batch size to 1 in order to
                                                          # test on one image at a time.
```

In order to get an idea of what the data looks like, we create some tables of all the classes and their contents for all

three data sets. Moreover, we plot an arbitrary image from the training data set.

```
# Get an idea of our data.
table(train.images$classes)
```

```
#>
#>   0   1   2   3   4   5   6   7   8   9  10  11  12  13  14  15  16  17  18  19
#> 133 144 144 110 117 110 104 107 129 132 133 100 144 136 127 107 104 144 112 105
#>  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39
#> 108 116 118 118 120 112 120 112 124 123 124 120 126 132 109 110 100 120 128 118
#>  40  41  42  43  44  45  46  47  48  49
#> 106 110 133  99 156 133 104  96 106 109
```

```
table(validation.images$classes)
```

```
#>
#>  0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25
#> 33 36 35 27 29 27 26 26 32 33 33 25 35 34 31 26 26 35 27 26 26 28 29 29 30 27
#> 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49
#> 30 27 30 30 31 29 31 32 27 27 24 30 32 29 26 27 33 24 38 33 26 24 26 27
```

```
table(test.images$classes)
```

```
#>
#>  0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25
#>  5  5  5  5  5  5  5  5  5  5  5  5  5  5  5  5  5  5  5  5  5  5  5  5  5  5
#> 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49
#>  5  5  5  5  5  5  5  5  5  5  5  5  5  5  5  5  5  5  5  5  5  5  5  5
```

```
# Plot image number 17.
plot(as.raster(train.images[[1]][[1]][17,,,]))
```

We can see that all bird species have 5 testing images. Moreover, notice that the number of images per species differs significantly between some of the species in the two other datasets, ranging from 96 to 156 in the training data and from 24 to 38 in the validation data. This can be worth taking into account when considering classification performance.

## Defining and Training First Model

As noted, we decide to use a CNN to classify the images. We load the pretrained Xception network in order to (hopefully) quickly get acceptable baseline results. We freeze all weights in this pre-trained model, but couple it with another fully connected dense layer, which has weights that will be trained on our specific problem. In addition, we add a layer of average pooling and some dropout to prevent overfitting. Notice that the number of nodes in the

fully connected dense layer, the learning rate and the dropout rate are set to somewhat arbitrary values at this moment, as they will be tuned later.

```
# We use the pretrained model to get good results off the bat.
mod.base <- application_xception(weights = 'imagenet',
                                 include_top = FALSE, input_shape = c(width, height, 3))
#summary(mod.base)
freeze_weights(mod.base)
#summary(mod.base) # Freeze the weights of the pretrained xception model.

model.function <- function(learning_rate = 0.001,
                           dropoutrate=0.2, n_dense=1024){
  # Function to add layers to the pre-trained model.
  k_clear_session()

  model <- keras_model_sequential() %>%
    mod.base %>%
    layer_global_average_pooling_2d() %>%
    layer_dense(units = n_dense) %>%
    layer_activation("relu") %>%
    layer_dropout(dropoutrate) %>%
    layer_dense(units=output.n, activation="softmax")

  model %>% compile(
    loss = "categorical_crossentropy",
    optimizer = optimizer_adam(learning_rate = learning_rate),
    metrics = "accuracy"
  )
  return(model)
}
#
model <- model.function()
#summary(model)
#
# # We train the model.
# # This training is left for the larger computer, via ssh.
# hist <- model %>% fit(
#   train.images,
#   steps_per_epoch = train.images$n %/% batch_size,# Integer division.
#   epochs = epochs,
#   validation_data = validation.images,
#   validation_steps = validation.images$n %/% batch_size # Integer division.
# )
#
# # We save the model after fitting, since it took a little while!
# model %>% save_model_hdf5("finetunedXception1.h5")
```

We train this first model and save the entire model image, i.e. the model architecture, the weights and the state of the optimizer. Notice that I was not able to install the Tensorflow backend onto my NVIDIA GPU, which lead to a relatively slow training process (on the CPUs). Thus, instead of training it on my personal computer, I trained the models on a large computer located at my home university, via ssh. On this computer the training was performed in parallel on GPUs, meaning that it was trained much faster. Moreover, since I was using tmux, which is a terminal multiplexer, I was able to detach the session and log out of the computer without the computations stopping. The reason I saved the entire model image was so that I could copy it to my computer (via scp), load it into R and make predictions. Train the model however you like, but, unless you have a large computer at home, I would recommend following a similar procedure.

## Evaluating First Model

The model image is loaded and evaluated on the test data, in order to see how well it has performed. The accuracy reported is approximately 84%, which is not bad for the initial model.

```
# Load the model fitted on the other computer.
model <- load_model_hdf5("finetunedXception1.h5")

# We evaluate our model on the test data.
model %>% evaluate(test.images,
                               steps = test.images$n)
```

```
#>     loss accuracy
#> 0.506085 0.844000
```

```
# 84% accuracy with first model.
```

Next we predict on another image, which has not been used in training, taken from the Wikipedia page of Abbot's babbler, which is one of the bird species we have trained our model on. The image is cropped such that is has a 1:1 ratio between height and width. It is added to the parent folder of this report, such that the results are reproducible.

```
# Next we test with another image. The image is taken from the
# Wikipedia page on Abbot's babbler.
test.image <- image_load("abbotsBabbler.jpg",
                         target_size = target.size)

# We make an overview of the model's predictions.
x <- image_to_array(test.image)
x <- array_reshape(x, c(1, dim(x))) # Reshape image to expected input dimensions by model.
x <- x/255 # Rescale pixel values.
plot(as.raster(x[1,,,])) # Plot the image.
```

```
pred <- model %>% predict(x) # Make predictions on image.
# Make dataframe of predictions, with names of the respective birds.
pred <- data.frame("Bird" = label.list, "Probability" = t(pred))
# Order the probabilities decreasingly and only show the largest 5.
pred <- pred[order(pred$Probability, decreasing=T),][1:5,]
# Change the probability to percentage.
pred$Probability <- paste(format(100*pred$Probability,2),"%")
knitr::kable(pred, caption = "Predictions on Abbot's babbler from Initial Model")
```

Table 1: Predictions on Abbot's babbler from Initial Model

| | Bird | Probability |
|---|---|---|
| 1 | ABBOTTS BABBLER | 71.933359 % |
| 27 | ANTBIRD | 7.443158 % |
| 43 | BANANAQUIT | 6.662891 % |
| 19 | AMERICAN REDSTART | 5.563317 % |
| 12 | ALTAMIRA YELLOWTHROAT | 2.402629 % |

```
# 72% probability.
# This means that the model would have classified this test image correctly.
```

We can see that the initial model gives Abbot's babbler a probability of approximately 72%. This means that, when using a classifier that classifies according to the largest probability, the model would have classified this bird correctly. Notice however that, the closer to 100% the probability of a class is, the greater certainty the model gives about the species it predicts. So far, not bad for a first model.

Next we have a look at which birds are well identified vs. birds that are not well identified.

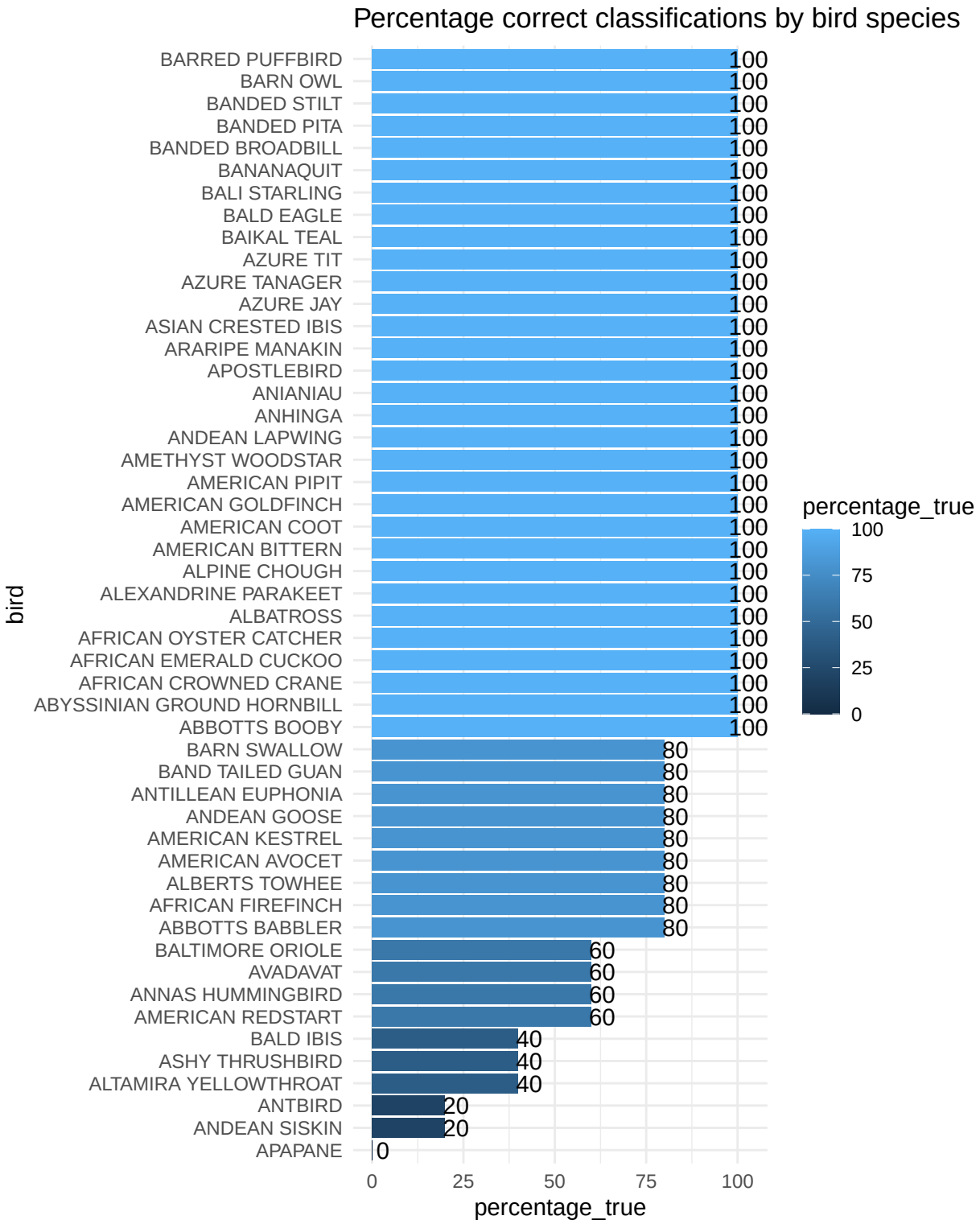
```
# Next we have a look at which birds are well identified vs.
# which birds are not so well identified by the model.

# We predict on the testing images and save the predictions as a data frame.
predictions <- model %>%
  predict(
    test.images,
    steps = test.images$n
  ) %>% as.data.frame
# Each testing image is given in each row, where each column value is the probability
# of the image belonging to that species of bird, according to the model.

# Change the column names to "Class<i>" respectively, i \in \{0,49\}.
names(predictions) <- paste0("Class",0:49)

# Add another column to the dataframe which tells us which class has the highest
# probability. Thus, this is the class that the model would predict.
predictions$predicted_class <-
  paste0("Class",apply(predictions,1,which.max)-1)
# Add the true classes to the dataframe as well.
predictions$true_class <- paste0("Class",test.images$classes)
```

```
# Finally, we count the percentage of correct classifications
# (since each of our classes has exactly 5 test images,
# the values will be either 0, 20, 40, 60, 80 or 100%).
predictions %>% group_by(true_class) %>%
  summarise(percentage_true = 100*sum(predicted_class ==
                                        true_class)/n()) %>%
  left_join(data.frame(bird= names(test.images$class_indices),
                       true_class=paste0("Class",0:49)),by="true_class") %>%
  select(bird, percentage_true) %>%
  mutate(bird = fct_reorder(bird,percentage_true)) %>%
  ggplot(aes(x=bird,y=percentage_true,fill=percentage_true,
             label=percentage_true)) +
  geom_col() + theme_minimal() + coord_flip() +
  geom_text(nudge_y = 3) +
  ggtitle("Percentage correct classifications by bird species")
```




We can see that over half of the species in the test data are 100% correctly classified by our model. However, there are a few species that are mostly misclassified or not classified correctly at all, for instance "Apapane", which is never classified correctly in our test set. We notice that the species "Apapane" is class number 28 and has 124 images in the training set according to the table shown earlier. Thus, the especially bad performance on this species seems not to be because of few training images (compared to the other species), as the number of training images for the species is larger than the median and mean amount per class. However, it might still mean that for this species, the number of images is too low for the model to be able to learn its features.

## Hyperparameter Tuning

We want to improve the performance of the model by tuning some of the hyperparameters; we will tune the number of nodes in the fully connected dense layer, the learning rate and the dropout rate over discrete grids of values. In the blog post, the author used a brute force, self-made approach, using simple for-loops over the parameters he wanted to test. I will use tfruns, which is very similar, but seems slightly more sophisticated compared to his approach. Notice that I also ran the tfruns hyperparameter tuning on the computer located at my home university, naturally taking a longer time than the original computations, since a lot more models are trained. We explore the following grids of each of the hyperparameters

```
knitr::kable(c(0.001,0.0001), caption = "Values of Learning Rate", format = "simple", col.names = "")
```

Table 2: Values of Learning Rate

| |
|---|
| 1e-03 |
| 1e-04 |

```
knitr::kable(c(0.3,0.2), caption = "Values of Dropout Rate", format = "simple", col.names = "")
```

Table 3: Values of Dropout Rate

| |
|---|
| 0.3 |
| 0.2 |

```
knitr::kable(c(1024,256), caption = "Values of Number of Dense Nodes", format = "simple", col.names = "")
```

Table 4: Values of Number of Dense Nodes

| |
|---|
| 1024 |
| 256 |

The hyperparameter tuning is performed using two external R files; the file "tfruns.R" calls on the file "finetuning.R" for every scenario on the grid of values and finally compares all the runs to each other.

The **runs** directory as well as the saved **performance_table** are copied from the remote computer to my local computer via scp after the tfruns hyperparameter tuning is done. The results from this tuning is shown below.

```
# Read the performance table produced by the remote computer.
performance.table <- read.csv("performance_table.csv")

# Compare the two runs with the highest validation accuracy.
#compare_runs(c(performance.table[1,2], performance.table[2,2]))
# This comparison cannot be done when compiling the Rmd.
# However, we can see what the values of the models, ordered by
# best validation accuracy, below.
df.tf <- cbind(performance.table$metric_val_accuracy,
               performance.table$flag_dropout_rate,
               performance.table$flag_learning_rate,
               performance.table$flag_n_dense)
colnames(df.tf) <- c("Validation accuracy", "Dropout rate", "Learning rate", "Dense nodes")
knitr::kable(df.tf, caption = "Validation Accuracy and Hyperparameter Values of all Trained Models")
```

Table 5: Validation Accuracy and Hyperparameter Values of all Trained Models

| Validation accuracy | Dropout rate | Learning rate | Dense nodes |
|---|---|---|---|
| 0.8556 | 0.2 | 1e-04 | 1024 |
| 0.8417 | 0.3 | 1e-04 | 1024 |
| 0.8194 | 0.3 | 1e-03 | 1024 |
| 0.8160 | 0.2 | 1e-04 | 256 |
| 0.8083 | 0.2 | 1e-03 | 1024 |
| 0.7944 | 0.3 | 1e-03 | 256 |
| 0.7889 | 0.2 | 1e-03 | 256 |
| 0.7847 | 0.3 | 1e-04 | 256 |

We can see that the best model according to the runs is the model fitted with hyperparameters

- Learning rate: 0.0001
- Dropout rate: 0.2
- Dense nodes: 1024

We train the model with these hyperparameters on the remote computer and save it as the final model that will be used in the app.

## Evaluating New Model

We evaluate the final model, to quantify the improvements when comparing to the initial model.

```
# Load the model fitted on the other computer.
mod <- load_model_hdf5("tunedHypParamXception.h5")

# We evaluate our model on the test data.
mod %>% evaluate(test.images,
                              steps = test.images$n)
```

```
#>      loss  accuracy
#> 0.5479541 0.8680000
```

This model reports an accuracy of approximately 87%, which is slightly better than the initial model. Evaluation on the image of the Abbot's babbler is shown below.

```
# Next we test with another image. The image is taken from the
# Wikipedia page on Abbot's babbler.
test.image <- image_load("abbotsBabbler.jpg",
                         target_size = target.size)

# We make an overview of the model's predictions.
x <- image_to_array(test.image)
x <- array_reshape(x, c(1, dim(x))) # Reshape image to expected input dimensions by model.
x <- x/255 # Rescale pixel values.
plot(as.raster(x[1,,,])) # Plot the image.
```

```
pred <- mod %>% predict(x) # Make predictions on image.
# Make dataframe of predictions, with names of the respective birds.
pred <- data.frame("Bird" = label.list, "Probability" = t(pred))
# Order the probabilities decreasingly and only show the largest 5.
pred <- pred[order(pred$Probability, decreasing=T),][1:5,]
# Change the probability to percentage.
pred$Probability <- paste(format(100*pred$Probability,2),"%")
knitr::kable(pred, caption = "Predictions on Abbot's babbler from New Model")
```

Table 6: Predictions on Abbot's babbler from New Model

| | Bird | Probability |
|---|---|---|
| 1 | ABBOTTS BABBLER | 65.470839 % |
| 43 | BANANAQUIT | 10.877670 % |
| 19 | AMERICAN REDSTART | 7.057343 % |
| 9 | ALBERTS TOWHEE | 6.188409 % |
| 27 | ANTBIRD | 4.056965 % |

```
# We can see that the first model gives Abbot's babbler a 65% probability.
# This means that the model would have classified this test image correctly.
```

The correct species would still have been classified by this model, but it gives a smaller certainty than the initial model. Finally, we have a look at which birds are well identified vs. birds that are not well identified.

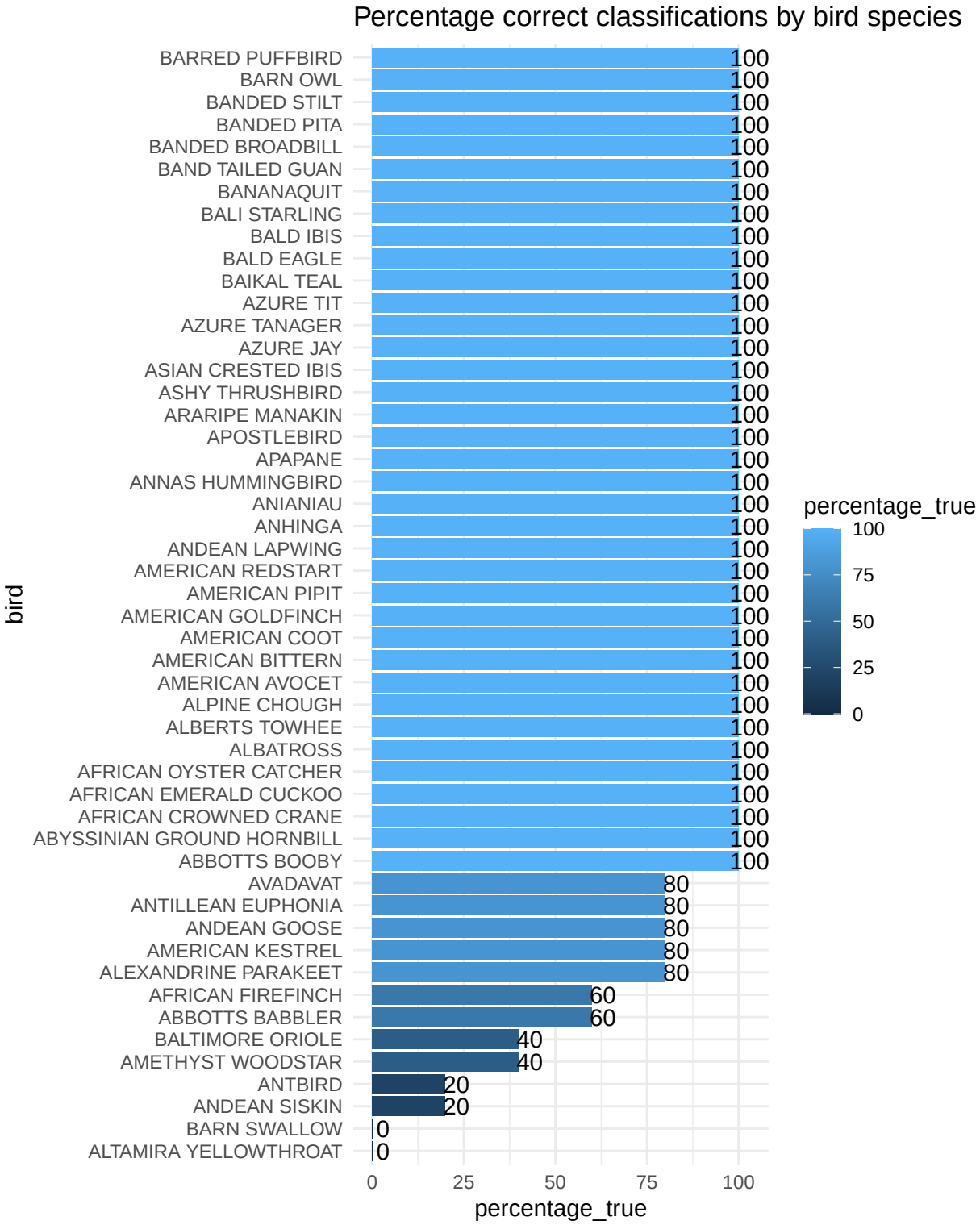
```
# Next we have a look at which birds are well identified vs.
# which birds are not so well identified by the model.

# We predict on the testing images and save the predictions as a data frame.
predictions <- mod %>%
  predict(
    test.images,
    steps = test.images$n
  ) %>% as.data.frame
# Each testing image is given in each row, where each column value is the probability
# of the image belonging to that species of bird, according to the model.

# Change the column names to "Class<i>" respectively, i \in \{0,49\}.
names(predictions) <- paste0("Class",0:49)
```

```
# Add another column to the dataframe which tells us which class has the highest
# probability. Thus, this is the class that the model would predict.
predictions$predicted_class <-
  paste0("Class",apply(predictions,1,which.max)-1)
# Add the true classes to the dataframe as well.
predictions$true_class <- paste0("Class",test.images$classes)
```

```
# Finally, we count the percentage of correct classifications
# (since each of our class has exactly 5 test images,
# the values will be either 0, 20, 40, 60, 80 or 100%).
predictions %>% group_by(true_class) %>%
  summarise(percentage_true = 100*sum(predicted_class ==
                                        true_class)/n()) %>%
  left_join(data.frame(bird= names(test.images$class_indices),
                       true_class=paste0("Class",0:49)),by="true_class") %>%
  select(bird, percentage_true) %>%
  mutate(bird = fct_reorder(bird,percentage_true)) %>%
  ggplot(aes(x=bird,y=percentage_true,fill=percentage_true,
             label=percentage_true)) +
  geom_col() + theme_minimal() + coord_flip() +
  geom_text(nudge_y = 3) +
  ggtitle("Percentage correct classifications by bird species")
```




In this case we can see that a large majority of the species are classified correctly in all testing cases. However, we still have some species that the model has trouble classifying. Notice that "Apapane" which was the worst performing bird in the first model, is classified 100% correctly in this model. Moreover, we notice that several of the worst performers are the same in both models, e.g. the "Antbird", the "Andean Siskin" and the "Altamira Yellowthroat" are among the least frequently correctly classified species in both models. Performing a similar rundown of the number of training images per species as earlier, we observe that the "Altamira Yellowthroat" has 100 images, the "Barn Swallow has 106, the"Antbird" has 120 and "Andean Siskin" has 118. In this case, the two first species are on the lower end of number of images in the training data, which might be a contributing factor to why the performance

on them is bad. The two latter species are hovering around the median (118) and mean (119.04) number of images. It could still be that they have too few images to train on for the "complexity" of the bird or perhaps the images of these species are slightly lower quality than the others.

We end this section by noting that the model could (and should) be tuned further, to further increase its performance, before putting it into production. Some examples of approaches could be increasing the size of the grid of hyperparameters we have tuned, tuning other hyperparameters like the batch size or the number of epochs, changing the optimizer or adding callbacks to the fitting.

# Part 2 - Implementing a Shiny App

We implement a web app with Shiny. The first step is to make a directory called "birdapp", with a subdirectory called "www". Since we call the main directory of our app "birdapp" this will automatically be the name of the app when deploying to Shinyapps, which will be done later.

## Implementing App Locally

The first step of the implementation is to copy the final model we trained into the "www" subdirectory of the "birdapp" directory. We also copy the label list, i.e. the list of birds, into the subdirectory. These files will be used in order to get the classifier running in the application and display the results.

Next we implement the web application itself. This is done in an R file called "app.R" in the "birdapp" folder. First we define the ui object (User interface). We use the dashboardPage function to create a dashboard page for the Shiny app. Inside the dashboardPage we define a dashboardHeader, a dashboardSidebar and a dashboardBody. Next we create the server object, which contains the interactive elements of the app. Inside the server function we load the image that is uploaded to the webapp by the user. Then we use the model we trained earlier to predict on the newly uploaded image. We create a dataframe with the top five predicted bird species after prediction, similar to what we did when testing the model earlier. Then we render it to the ui as a table, using the renderTable function. A warning text is defined, which displays a warning to the visitor if the model has highest predicted probability below 45%. Finally we display the image that was loaded from the user and delete the file from memory. All the code for the application is shown below.

```
# Shiny app for Birdapp main file.
#setwd("/home/ajo/gitRepos/DNN/birdapp")

library(shiny)
library(shinydashboard)
library(rsconnect)
library(keras)
library(tensorflow)
library(tidyverse)

# Could make a button in the app to switch between the two models!
# Could be interesting to do this, for fun!

#mod <- load_model_hdf5("www/finetunedXception1.h5") # This is the initial model.
mod <- load_model_hdf5("www/tunedHypParamXception.h5")
load("www/label_list.RData")
target.size <- c(224,224,3)
options(scipen=999) # Prevent scientific number formatting.

# Define the ui object.
ui <- dashboardPage(
  skin = "black",
  title = "BirdApp - Classify your Bird!",

  #(1) Header
```

```
  dashboardHeader(title=tags$h1("BirdApp - Classify your Bird!",
                                style="font-size: 120%; font-weight: bold; color: black"),
                  titleWidth = 350,
                  tags$li(class = "dropdown"),
                  dropdownMenu(type = "notifications",
                               icon = icon("question-circle", "fa-1x"), badgeStatus = NULL,
                               headerText="",
                               tags$li("Created by alexaoh, but"),
                               tags$li(a(href = "https://forloopsandpiepkicks.wordpress.com",
                                         target = "_blank",
                                         tagAppendAttributes(icon("exclamation-circle"),
                                                             class = "info"),
                                         "(heavily) inspired by (click here)"))

                  )),


  #(2) Sidebar

  dashboardSidebar(
    width=350,
    fileInput("input_image","File" ,accept = c('.jpg','.jpeg')),
    tags$br(),
    tags$p("Upload the image here (We only accept jpg and jpeg formats at this point).")
  ),


  #(3) Body

  dashboardBody(

    h4("Instruction:"),
    tags$br(),tags$p("1. Take a picture of a bird."),
    tags$p("2. Crop image so that bird fills out most of the image."),
    tags$p("3. Upload image with menu on the left."),
    tags$br(),

    fluidRow(
      column(h4("Image:"),imageOutput("output_image"), width=6),
      column(h4("Result:"),tags$br(),textOutput("warntext",), tags$br(),
             tags$p("This bird is probably a:"),tableOutput("text"),width=6)
    ),tags$br()

  ))


# We create the server object.
server <- function(input, output) {

  image <- reactive({image_load(input$input_image$datapath, target_size = target.size[1:2])})

  prediction <- reactive({
    if(is.null(input$input_image)){return(NULL)}
    x <- image_to_array(image())
    x <- array_reshape(x, c(1, dim(x)))
    x <- x/255
```

```
    pred <- mod %>% predict(x)
    pred <- data.frame("Bird" = label.list, "Prediction" = t(pred))
    pred <- pred[order(pred$Prediction, decreasing=T),][1:5,]
    pred$Prediction <- sprintf("%.2f %%", 100*pred$Prediction)
    pred
  })

  output$text <- renderTable({
    prediction()
  })

  output$warntext <- renderText({
    req(input$input_image)

    if(as.numeric(substr(prediction()[1,2],1,4)) >= 45){return(NULL)}
    warntext <- "Warning: I am not sure about this bird!"
    warntext
  })


  output$output_image <- renderImage({
    req(input$input_image)

    outfile <- input$input_image$datapath
    contentType <- input$input_image$type
    list(src = outfile,
         contentType=contentType,
         width = 400)
  }, deleteFile = TRUE)

}

shinyApp(ui, server)
#rsconnect::deployApp() # Deploy to shinyapps after following instructions.
```

## Deployment

We make a user on shinyapps.io and deploy the app there. The instructions for deployment are very clearly stated on the website after making the user. In short, we use the package `rsconnect` and the function `deployApp()` within the package to deploy the app, after correctly setting the account credentials. Visit the app here!