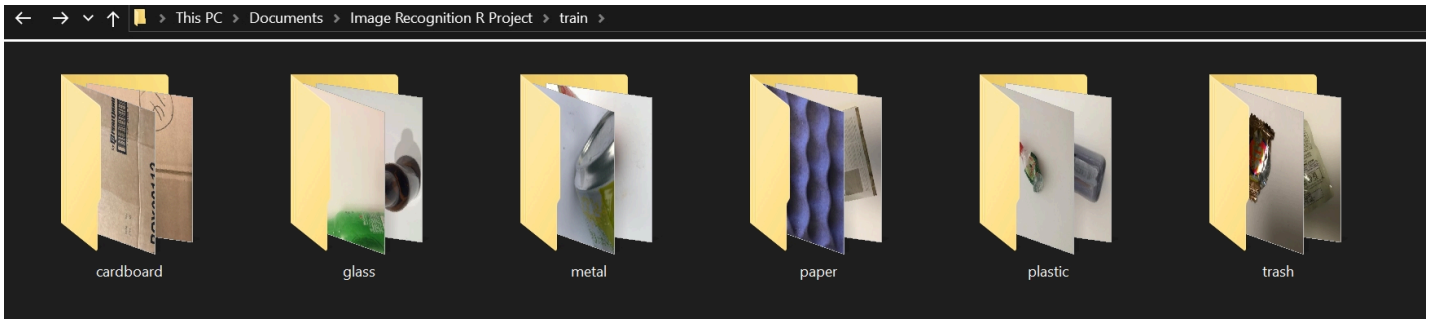
I Spy Garbage: Model Building

Code

I used the data set linked here: https://www.kaggle.com/datasets/asdasdasasdas/garbage-classification (https://www.kaggle.com/datasets/asdasdasasdas/garbage-classification)

I have gone ahead and manually divided the sets of images for each material into training and testing sets, where testing sets compose of the last 20% of images in each folder. None of the images are in any particular order, so this should be fine opposed to doing a randomized split.

Below is an screenshot of how I set up my training data, with each class of images being in their own sub-folder. For this data set, most of the images have the same dimensions, but this is not needed - We will set the images to a specific size later.



It can be quite difficult to gather data manually, which is why I use a data set that is already organized. But in essence, you can repeat the same methodology with your own set of images depending on use case.

Also, ensure you set the working directory to where the training and testing sets of images are saved.

Hide

```
rm(list = ls()) # clear all environmental variables that may be leftover

library(formatR)
# These will just allow better formatting in the knitted document
knitr::opts_chunk$set(tidy.opts = list(width.cutoff = 75), tidy = TRUE)
knitr::opts_chunk$set(warning = FALSE, message = FALSE)
```

Setup for our model

Let's go ahead and import the libraries we need:

Hide

```
# general library in R for data manipulation
library(tidyverse)

# packages for our image processing
# if you have never used these packages before, you need to follow the installation here (http
s://tensorflow.rstudio.com/install/)

library(keras)
library(tensorflow)
library(reticulate) # defines a python virtual environment
```

We will also go ahead and set the random seed. I also go ahead and set an environmental variable to 0 since it caused a warning when I ran this for the first time.

Hide

```
set_random_seed(220, disable_gpu = TRUE) # setting seed for (hopefully) reproducible results
TF_ENABLE_ONEDNN_OPTS=0 # Turning off any potential floating-point round-off errors by setting t
his environment variable to 0
```

Next, I create a list of each label for all our potential material classifications:

Hide

```
label_list <- dir("train/") # get a unique list of all trash types by getting each directory
output_n <- length(label_list)
save(label_list, file="label_list.R")
```

I also go ahead and set some parameters for image rescaling. This will be quite useful since it will mean the images we put into the model after it is built will not require specific formatting from the user.

Hide

```
width <- 224
height<- 224
target_size <- c(width, height)
rgb <- 3 #color channels
```

I also will set the path to our training data.

Hide

```
path_train <- "train/"
```

The next step is to use some data augmentation. This essentially applies random amounts of blurs or rotations to add some variation to our data, which helps to avoid over-fitting. I will also go ahead and set 20% of our data set as validation for when we build the model.

Hide

```
train_data_gen <- image_data_generator(rescale = 1/255,
                                       validation_split = .2)
```

Now we use a function called `flow_images_from_directory` to batch process our images with the generator function we defined above. We link this up to our training and validation data, and set some basic parameters.

Hide

```
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 = 220)
```

```
Found 1617 images belonging to 6 classes.
```

Hide

```
validation_images <- flow_images_from_directory(path_train,
                                                train_data_gen,
                                                subset = 'validation',
                                                target_size = target_size,
                                                class_mode = "categorical",
                                                classes = label_list,
                                                seed = 220)
```

```
Found 403 images belonging to 6 classes.
```

I will note that I could have just used separate folders for training and validation, but I just chose to let `keras` do it with a random split.

To check our work so far, let's output a table of our classes in our training images.

Hide

```
table(train_images$classes)
```

```
  0   1   2   3   4   5
256 320 263 381 309  88
```

This corresponds to the number of pictures in each folder, so everything looks great so far!

We can even display an example image:

Hide

```
plot(as.raster(train_images[[1]][[1]][17,,,]))
```

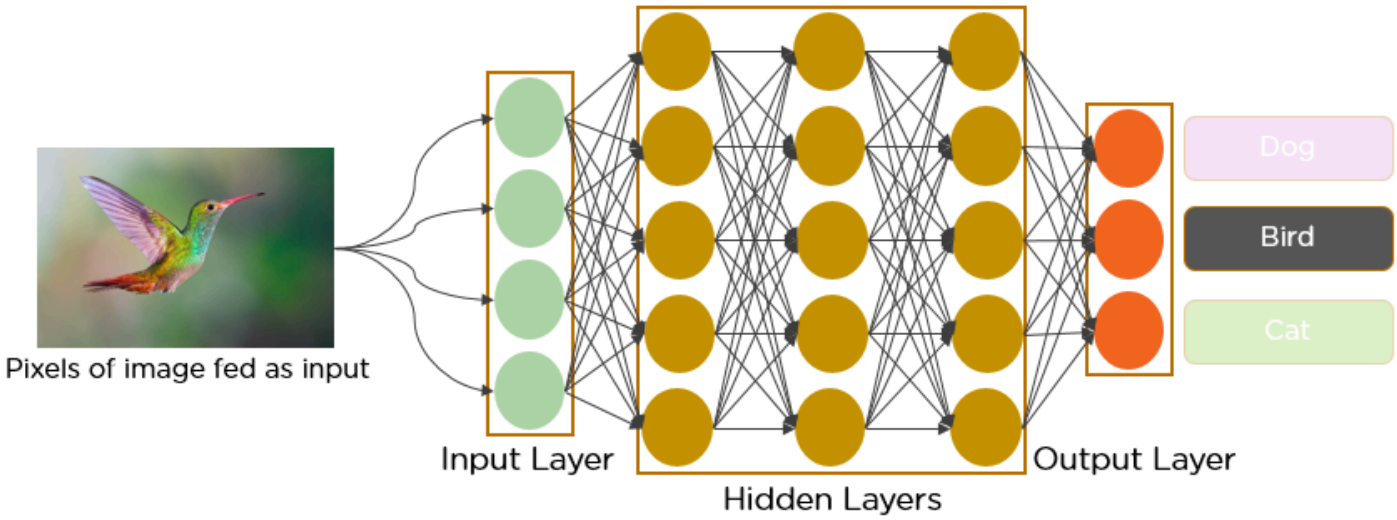
Note that the first element in the `train_images` object has the pixel values of each image which is a 4D-tensor (number of image, width, height, rgb channel), so we are plotting image number 17 above.

Training the model:

Given the nature of this project, we are going to be using a convoluted neural network (CNN). I found the following resources helpful to understand how it works, but I will also give my own brief explanation below regarding the basic functionality:

- Video: “But what is a convolution?” by 3Blue1Brown (https://youtu.be/KuXjwB4LzSA?si=voPVrXqOyJnRBUGI (https://youtu.be/KuXjwB4LzSA?si=voPVrXqOyJnRBUGI))
- Video: “Convolutional Neural Networks (CNNs) explained” by deeplizard (https://www.youtube.com/watch?v=YRhxdVk_sIs (https://www.youtube.com/watch?v=YRhxdVk_sIs))
- Article: “All about convolutions, kernals, features in CNN” by Abhishek Jain (https://medium.com/@abhishekjainindore24/all-about-convolutions-kernels-features-in-cnn-c656616390a1 (https://medium.com/@abhishekjainindore24/all-about-convolutions-kernels-features-in-cnn-c656616390a1))

My explanation: So with images, they are made out of pixels, and are effectively just a grid of pixels (with numbers representing each pixel). For a given convolution layer on a CNN, we take what is considered a kernal, most often a square matrix of values inside, and “slide” (or convolute) it over each group of pixels to get a resulting dot product. This transforms the overall image into something else, which can help identify any particular features, such as maybe colors or edges.



In terms of the size of a kernal, it is usually 1x1, 3x3, or 5x5. I believe this is just simply to avoid a common ML problem of having too much noise in your model if the kernal becomes too big, capturing too much information. In terms of the values/weights in a kernal, this depends on the scenerio and will depend on what features you are trying to capture. This is why kernals are often also termed as 'filters.'

And of course, that is only for one single layer. CNN will have multiple convolution layers with multiple different kernals, and the CNN will be trained to adjust the kernals used for each layer to be able to figure out all the right features of an image to classify it.

With that said, it is important to address a big obstacle here. Neural networks (more specifically, convoluted neural networks) have so many ways they can be trained with lots of different parameters. This offers lots of flexibility, but we don't really know on the best way to set up our model. Essentially, we are stuck with the bias variance tradeoff, where we could build a model that is well known s.t. model variance is low, but there may be violations to the assumptions of that model, causing it to be biased. Conversely, we could have a model that can be quite complicated (thus reducing bias), but since there are so many parameters, it would increase model variance.

For the sake of this project, there is a way we can ensure good baseline results, by using models that were pre-trained in the past for other applications. We choose the xception-network with the weights pre-trained on the ImageNet data set, a large data set used for object recognition. However, we will edit the final layer, since this is the output layer that classifies the images, and of course, we need to adjust it to train for our images of potential trash. This is done by setting the parameter `include_top` to FALSE.

Hide

```
mod_base <- application_xception(weights = 'imagenet', include_top = FALSE, input_shape = c(widt
h, height, 3))
```

```
WARNING:tensorflow:From C:\Users\kevin\OneDrive\DOCUME~1\.virtualenvs\r-tensorflow\lib\site-packages\keras\src\backend.py:1398: The name tf.executing_eagerly_outside_functions is deprecated. Please use tf.compat.v1.executing_eagerly_outside_functions instead.

2024-08-11 07:47:54.553793: I tensorflow/core/platform/cpu_feature_guard.cc:182] This TensorFlow binary is optimized to use available CPU instructions in performance-critical operations.
To enable the following instructions: SSE SSE2 SSE3 SSE4.1 SSE4.2 AVX2 AVX512F AVX512_VNNI FMA, in other operations, rebuild TensorFlow with the appropriate compiler flags.
WARNING:tensorflow:From C:\Users\kevin\OneDrive\DOCUME~1\.virtualenvs\r-tensorflow\lib\site-packages\keras\src\layers\normalization\batch_normalization.py:979: The name tf.nn.fused_batch_norm is deprecated. Please use tf.compat.v1.nn.fused_batch_norm instead.
```

Hide

```
freeze_weights(mod_base) # we freeze the weights so they are no longer trainable, since we want to keep them from the original network
```

Below, we define a function that will build that final layer that we left out from the `imagenet` model. I set some parameters to variables that we will use to fine-tune our model later.

Hide

```
model_function <- function(learningRate = 0.0001, dropoutrate=0.2,
                           n_dense=1024){

  k_clear_session() #resets all generated states, good for general practice before defining another model

  model <- keras_model_sequential() %>%
    mod_base %>%
    layer_global_average_pooling_2d() %>%
    layer_dense(units = n_dense) %>%
    layer_activation("relu") %>%
    layer_dropout(rate = dropoutrate, seed = 220) %>%
    layer_dense(units=output_n, activation="softmax")

  model %>% compile(
    loss = "categorical_crossentropy",
    optimizer = optimizer_adam(learning_rate = learningRate),
    metrics = "accuracy"
  )

  return(model)

}
```

Now, we let the model compile with those base values and inspect the architecture.

Hide

```
model <- model_function()
model
```

```
Model: "sequential"
_________________________________________________________________
 Layer (type)                Output Shape              Param #   Train
able
=================================================================
 xception (Functional)       (None, 7, 7, 2048)        20861480  N
 global_average_pooling2d (GlobalAvera  (None, 2048)           0         Y
 gePooling2D)
 dense_1 (Dense)             (None, 1024)              2098176   Y
 activation (Activation)     (None, 1024)              0         Y
 dropout (Dropout)           (None, 1024)              0         Y
 dense (Dense)               (None, 6)                 6150      Y
=================================================================
Total params: 22965806 (87.61 MB)
Trainable params: 2104326 (8.03 MB)
Non-trainable params: 20861480 (79.58 MB)
_________________________________________________________________
```

We can see that we have 20 million parameters from the xception model that are non-trainable, since we only train the top layer, which is a 1024 x 1 feature vector that is densely connected to the output classification layer. This is essentially 1024 variables, which take on certain values depending on if our image has certain properties, and all of this is trained during the convolution layers of the xception network (aka the ‘black box’).

With that said, let is train the model with our training data now!

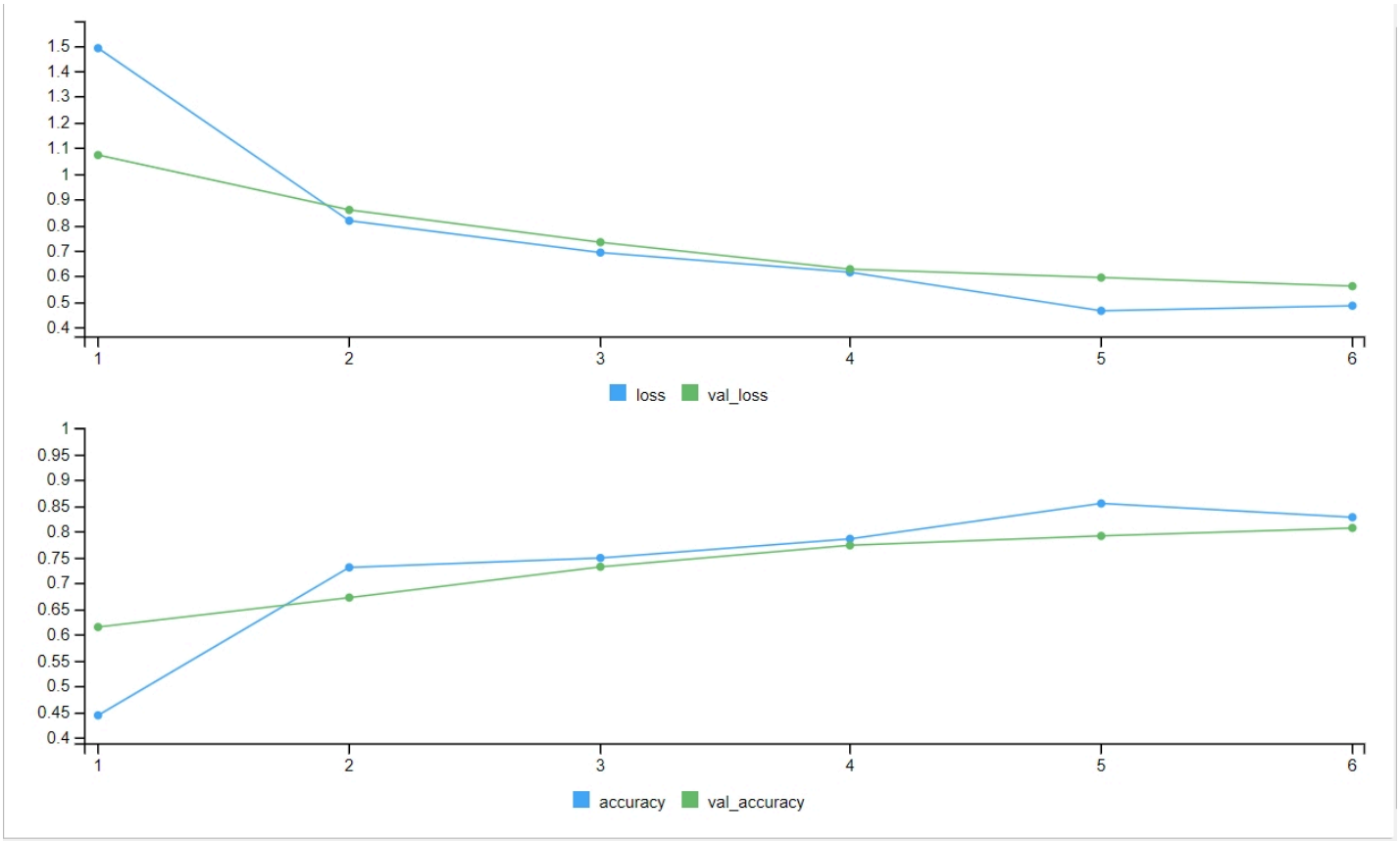
Hide

```
batch_size <- 32 # number of training samples fed to our neural network at once
epochs <- 6 # number of times our entire training data set is passed through the neural network

hist <- model %>% fit_generator(
  train_images,
  steps_per_epoch = train_images$n %/% batch_size,
  epochs = epochs,
  validation_data = validation_images,
  validation_steps = validation_images$n %/% batch_size,
  verbose = 2
)
```

```
Epoch 1/6
```

```
WARNING:tensorflow:From C:\Users\kevin\OneDrive\DOCUME~1\.virtualenvs\r-tensorflow\lib\site-packages\keras\src\utils\tf_utils.py:492: The name tf.ragged.RaggedTensorValue is deprecated. Please use tf.compat.v1.ragged.RaggedTensorValue instead.

WARNING:tensorflow:From C:\Users\kevin\OneDrive\DOCUME~1\.virtualenvs\r-tensorflow\lib\site-packages\keras\src\engine\base_layer_utils.py:384: The name tf.executing_eagerly_outside_functions is deprecated. Please use tf.compat.v1.executing_eagerly_outside_functions instead.
```

```
50/50 - 223s - loss: 1.4893 - accuracy: 0.4429 - val_loss: 1.0724 - val_accuracy: 0.6146 - 223s/epoch - 4s/step
Epoch 2/6
50/50 - 209s - loss: 0.8163 - accuracy: 0.7306 - val_loss: 0.8583 - val_accuracy: 0.6719 - 209s/epoch - 4s/step
Epoch 3/6
50/50 - 220s - loss: 0.6921 - accuracy: 0.7489 - val_loss: 0.7325 - val_accuracy: 0.7318 - 220s/epoch - 4s/step
Epoch 4/6
50/50 - 211s - loss: 0.6149 - accuracy: 0.7861 - val_loss: 0.6273 - val_accuracy: 0.7734 - 211s/epoch - 4s/step
Epoch 5/6
50/50 - 195s - loss: 0.4648 - accuracy: 0.8549 - val_loss: 0.5943 - val_accuracy: 0.7917 - 195s/epoch - 4s/step
Epoch 6/6
50/50 - 194s - loss: 0.4847 - accuracy: 0.8281 - val_loss: 0.5609 - val_accuracy: 0.8073 - 194s/epoch - 4s/step
```



Above is a corresponding plot that RStudio shows of the accuracy over the 6 epochs (I just have a screenshot since this does not appear in a notebook file). We can see that our accuracy is around 80%, which isn't bad, but there is certainly room for improvement.

Testing our model

First, let's now use the images we set aside for testing in our model, to see how it performs, with the same parameters as before when using our training images.

Hide

```
path_test <- "test/" # setting path to our testing data

test_data_gen <- image_data_generator(rescale = 1/255)

test_images <- flow_images_from_directory(path_test,
                                          test_data_gen,
                                          target_size = target_size,
                                          class_mode = "categorical",
                                          classes = label_list,
                                          shuffle = F,
                                          seed = 220)
```

```
Found 507 images belonging to 6 classes.
```

Hide

```
model %>% evaluate_generator(test_images,
                             steps = test_images$n)
```

```
  1/507 [..............................] - ETA: 37:09 - loss: 0.1385 - accuracy: 0.937
5
  2/507 [..............................] - ETA: 30:36 - loss: 0.2489 - accuracy: 0.875
0
  3/507 [..............................] - ETA: 30:06 - loss: 0.3294 - accuracy: 0.833
3
  4/507 [..............................] - ETA: 31:16 - loss: 0.3192 - accuracy: 0.851
6
  5/507 [..............................] - ETA: 31:57 - loss: 0.3227 - accuracy: 0.856
2
  6/507 [..............................] - ETA: 31:35 - loss: 0.3461 - accuracy: 0.854
2
  7/507 [..............................] - ETA: 31:30 - loss: 0.4071 - accuracy: 0.830
4
  8/507 [..............................] - ETA: 31:20 - loss: 0.4254 - accuracy: 0.828
1
  9/507 [..............................] - ETA: 30:44 - loss: 0.4280 - accuracy: 0.833
3
 10/507 [..............................] - ETA: 30:19 - loss: 0.4187 - accuracy: 0.843
8
 11/507 [..............................] - ETA: 30:07 - loss: 0.4069 - accuracy: 0.852
3
 12/507 [..............................] - ETA: 29:46 - loss: 0.4332 - accuracy: 0.843
8
 13/507 [..............................] - ETA: 29:21 - loss: 0.4271 - accuracy: 0.846
2
 14/507 [..............................] - ETA: 29:06 - loss: 0.4391 - accuracy: 0.839
3
 15/507 [..............................] - ETA: 28:45 - loss: 0.5479 - accuracy: 0.810
4
 16/507 [..............................] - ETA: 28:14 - loss: 0.6458 - accuracy: 0.7732
```

```
WARNING:tensorflow:Your input ran out of data; interrupting training. Make sure that your datase
t or generator can generate at least `steps_per_epoch * epochs` batches (in this case, 507 batch
es). You may need to use the repeat() function when building your dataset.
```

```
507/507 [==============================] - 56s 103ms/step - loss: 0.6458 - accuracy: 0.7732
     loss  accuracy
0.6457799 0.7731755
```

(I am going to ignore the error since we should have enough data; I’m not sure why it pops up)

So with our testing data, we also get around 80% accuracy (give or take a % or two). This could be improved, but shows that our model was not overfit to our training data.

For fun, I will use a candy wrapper that was not in our training data to test out how our model predicts it. Again, with how we setup the model beforehand, there is no need for me to “set up” the image beforehand - the model will do that for us.

Hide

```
test_image <- image_load("testImage.jpg",
                         target_size = target_size)

x <- image_to_array(test_image)
x <- array_reshape(x, c(1, dim(x)))
x <- x/255
pred <- model %>% predict(x)
```

```
1/1 [==============================] - 3s 3s/step
□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□
1/1 [==============================] - 3s 3s/step
```

Hide

```
pred <- data.frame("Type of Waste" = label_list, "Probability" = t(pred))
pred <- pred[order(pred$Probability, decreasing=T),][1:6,]
pred$Probability <- paste(format(100*pred$Probability,2),"%")
pred
```

	Type.of.Waste <chr>	Probability <chr>
5	plastic	89.044970 %
6	trash	3.587638 %
3	metal	2.536827 %
4	paper	1.787657 %
2	glass	1.582717 %
1	cardboard	1.460196 %

6 rows

Looks like our model predicted correctly, since this is a plastic wrapper.

Now I raise a question. Is there a particular type of material in our classification that our model does not accurately predict (which would be the main source of our lower accuracy)? The code block below will show us just that.

We will generate a matrix of all predictions for each image in our testing data for every material type. We label the highest prediction as the predicted class and then compare it with the actual correct class to get the % of correct classifications.

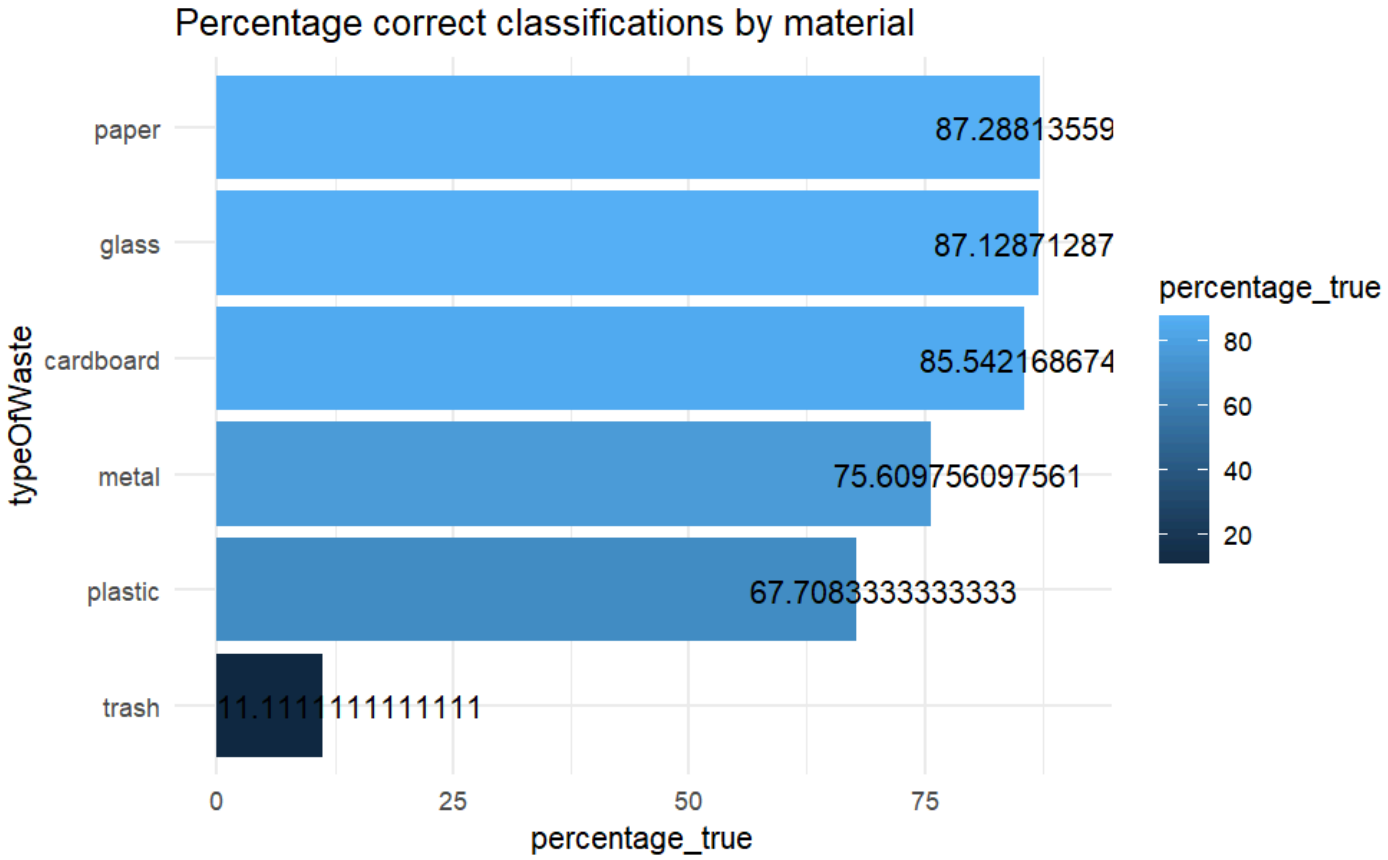
Hide

```
predictions <- model %>%
  predict_generator(
    generator = test_images,
    steps = test_images$n
  ) %>% as.data.frame
```

```
WARNING:tensorflow:Your input ran out of data; interrupting training. Make sure that your datase
t or generator can generate at least `steps_per_epoch * epochs` batches (in this case, 507 batch
es). You may need to use the repeat() function when building your dataset.
```

Hide

```
names(predictions) <- paste0("Class",0:5)

predictions$predicted_class <-
  paste0("Class",apply(predictions,1,which.max)-1)
predictions$true_class <- paste0("Class",test_images$classes)

predictions %>% group_by(true_class) %>%
  summarise(percentage_true = 100*sum(predicted_class ==
                                        true_class)/n()) %>%
  left_join(data.frame(typeOfWaste = names(test_images$class_indices),
                       true_class = paste0("Class",0:5)), by="true_class") %>%
  select(typeOfWaste, percentage_true) %>%
  mutate(typeOfWaste = fct_reorder(typeOfWaste,percentage_true)) %>%
  ggplot(aes(x=typeOfWaste,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 material")
```



We can see that the trash category does not appear to get accurate classifications. This makes sense as well, since this is a broad category with many images of different things that are not extremely similar with one another.

Refining our model

So there are multiple ways to try and solve our problem with our model being poor accuracy. We could just try and gather more image data for the trash category, but given this is more of a broad category, this may not be that easy. So instead, we will try and tune the model we made. There are multiple ways, and for simplicity, I go with a

very homemade approach, which is similar to `grid_search` in python. We will define a grid with values for some parameters that our model uses. We will just loop through every combination of parameters and fit the model, save the model combination, and final accuracy. We of course, want the parameters that result in the highest validation accuracy.

Note that on my computer, this whole process takes a while with our set of images. In the future, I may come back to this project and try more efficient methods to achieve this task, or test more sets of parameters, but since it is quite computationally intensive, I only test a very small grid of parameters.

Hide

```
tune_grid <- data.frame("dropoutrate" = c(0.3,0.2),
                        "n_dense" = c(1024,256))

tuning_results <- NULL

for (j in 1:length(tune_grid$dropoutrate)){
  for (k in 1:length(tune_grid$n_dense)){

      model <- model_function(
        # learning_rate = tune_grid$learning_rate[i],
        dropoutrate = tune_grid$dropoutrate[j],
        n_dense = tune_grid$n_dense[k]
      )

      hist <- model %>% fit_generator(
        train_images,
        steps_per_epoch = train_images$n %/% batch_size,
        epochs = epochs,
        validation_data = validation_images,
        validation_steps = validation_images$n %/%
          batch_size,
        verbose = 2
      )

      #Save model configurations
      tuning_results <- rbind(
        tuning_results,
        c("dropoutrate" = tune_grid$dropoutrate[j],
          "n_dense" = tune_grid$n_dense[k],
          "val_accuracy" = hist$metrics$val_accuracy))

  }
}
```

```
Epoch 1/6
50/50 - 217s - loss: 1.6334 - accuracy: 0.3937 - val_loss: 1.1427 - val_accuracy: 0.6432 - 217s/
epoch - 4s/step
Epoch 2/6
50/50 - 208s - loss: 0.8870 - accuracy: 0.7287 - val_loss: 0.7871 - val_accuracy: 0.7318 - 208s/
epoch - 4s/step
Epoch 3/6
50/50 - 199s - loss: 0.7441 - accuracy: 0.7407 - val_loss: 0.6832 - val_accuracy: 0.7917 - 199s/
epoch - 4s/step
Epoch 4/6
50/50 - 202s - loss: 0.5758 - accuracy: 0.8076 - val_loss: 0.6393 - val_accuracy: 0.7708 - 202s/
epoch - 4s/step
Epoch 5/6
50/50 - 198s - loss: 0.5447 - accuracy: 0.8145 - val_loss: 0.6565 - val_accuracy: 0.7604 - 198s/
epoch - 4s/step
Epoch 6/6
50/50 - 197s - loss: 0.5196 - accuracy: 0.8025 - val_loss: 0.5540 - val_accuracy: 0.7995 - 197s/
epoch - 4s/step
Epoch 1/6
50/50 - 200s - loss: 1.7356 - accuracy: 0.3142 - val_loss: 1.3443 - val_accuracy: 0.5677 - 200s/
epoch - 4s/step
Epoch 2/6
50/50 - 194s - loss: 1.2185 - accuracy: 0.5748 - val_loss: 1.0647 - val_accuracy: 0.6797 - 194s/
epoch - 4s/step
Epoch 3/6
50/50 - 194s - loss: 0.9875 - accuracy: 0.6700 - val_loss: 0.8810 - val_accuracy: 0.7057 - 194s/
epoch - 4s/step
Epoch 4/6
50/50 - 193s - loss: 0.8118 - accuracy: 0.7413 - val_loss: 0.8163 - val_accuracy: 0.7083 - 193s/
epoch - 4s/step
Epoch 5/6
50/50 - 193s - loss: 0.6986 - accuracy: 0.7729 - val_loss: 0.7712 - val_accuracy: 0.7266 - 193s/
epoch - 4s/step
Epoch 6/6
50/50 - 192s - loss: 0.6641 - accuracy: 0.7621 - val_loss: 0.6864 - val_accuracy: 0.7500 - 192s/
epoch - 4s/step
Epoch 1/6
50/50 - 204s - loss: 1.6332 - accuracy: 0.3874 - val_loss: 1.1616 - val_accuracy: 0.5677 - 204s/
epoch - 4s/step
Epoch 2/6
50/50 - 217s - loss: 0.9987 - accuracy: 0.6442 - val_loss: 0.7822 - val_accuracy: 0.7604 - 217s/
epoch - 4s/step
Epoch 3/6
50/50 - 197s - loss: 0.6886 - accuracy: 0.7767 - val_loss: 0.7218 - val_accuracy: 0.7188 - 197s/
epoch - 4s/step
Epoch 4/6
50/50 - 195s - loss: 0.6039 - accuracy: 0.7817 - val_loss: 0.6233 - val_accuracy: 0.7839 - 195s/
epoch - 4s/step
Epoch 5/6
50/50 - 194s - loss: 0.5196 - accuracy: 0.8215 - val_loss: 0.6034 - val_accuracy: 0.7943 - 194s/
epoch - 4s/step
Epoch 6/6
```

```
50/50 - 204s - loss: 0.4453 - accuracy: 0.8580 - val_loss: 0.5663 - val_accuracy: 0.8125 - 204s/
epoch - 4s/step
Epoch 1/6
50/50 - 197s - loss: 1.7092 - accuracy: 0.3382 - val_loss: 1.3430 - val_accuracy: 0.5964 - 197s/
epoch - 4s/step
Epoch 2/6
50/50 - 200s - loss: 1.1751 - accuracy: 0.6366 - val_loss: 1.0366 - val_accuracy: 0.6901 - 200s/
epoch - 4s/step
Epoch 3/6
50/50 - 190s - loss: 0.9145 - accuracy: 0.7199 - val_loss: 0.9333 - val_accuracy: 0.6328 - 190s/
epoch - 4s/step
Epoch 4/6
50/50 - 191s - loss: 0.7782 - accuracy: 0.7527 - val_loss: 0.7771 - val_accuracy: 0.7240 - 191s/
epoch - 4s/step
Epoch 5/6
50/50 - 189s - loss: 0.6998 - accuracy: 0.7584 - val_loss: 0.7034 - val_accuracy: 0.7630 - 189s/
epoch - 4s/step
Epoch 6/6
50/50 - 190s - loss: 0.6134 - accuracy: 0.7886 - val_loss: 0.7072 - val_accuracy: 0.7578 - 190s/
epoch - 4s/step
```

Hide

```
tuning_results
```

```
     dropoutrate n_dense val_accuracy1 val_accuracy2 val_accuracy3 val_accuracy4 val_accuracy5
[1,]         0.3    1024     0.6432292     0.7317708     0.7916667     0.7708333     0.7604167
[2,]         0.3     256     0.5677083     0.6796875     0.7057292     0.7083333     0.7265625
[3,]         0.2    1024     0.5677083     0.7604167     0.7187500     0.7838542     0.7942708
[4,]         0.2     256     0.5963542     0.6901042     0.6328125     0.7239583     0.7630208
     val_accuracy6
[1,]     0.7994792
[2,]     0.7500000
[3,]     0.8125000
[4,]     0.7578125
```

We can extract the best results:

Hide

```
best_results <- tuning_results[which(
  tuning_results[,ncol(tuning_results)] ==
    max(tuning_results[,ncol(tuning_results)])),]

best_results
```

```
  dropoutrate       n_dense val_accuracy1 val_accuracy2 val_accuracy3 val_accuracy4 val_accuracy
5
    0.2000000  1024.0000000     0.5677083     0.7604167     0.7187500     0.7838542     0.794270
8
val_accuracy6
    0.8125000
```

Finalizing our model

Now let us retrain our model with the best parameters we identified above. I also reduce the number of epochs to 5 to avoid potential over-fitting.

Hide

```
model <- model_function(
                        dropoutrate = best_results["dropoutrate"],
                        n_dense = best_results["n_dense"])

hist <- model %>% fit_generator(
  train_images,
  steps_per_epoch = train_images$n %/% batch_size,
  epochs = 5,
  validation_data = validation_images,
  validation_steps = validation_images$n %/% batch_size,
  verbose = 2
)
```

```
Warning in fit_generator(., train_images, steps_per_epoch = train_images$n%/%batch_size,  :
  `fit_generator` is deprecated. Use `fit` instead, it now accept generators.
```

```
Epoch 1/5
50/50 - 207s - loss: 1.4899 - accuracy: 0.4044 - val_loss: 1.0374 - val_accuracy: 0.7266 - 207s/
epoch - 4s/step
Epoch 2/5
50/50 - 199s - loss: 0.8948 - accuracy: 0.7091 - val_loss: 0.7885 - val_accuracy: 0.7370 - 199s/
epoch - 4s/step
Epoch 3/5
50/50 - 199s - loss: 0.7025 - accuracy: 0.7565 - val_loss: 0.8032 - val_accuracy: 0.6849 - 199s/
epoch - 4s/step
Epoch 4/5
50/50 - 198s - loss: 0.5855 - accuracy: 0.8025 - val_loss: 0.6324 - val_accuracy: 0.7865 - 198s/
epoch - 4s/step
Epoch 5/5
50/50 - 198s - loss: 0.5026 - accuracy: 0.8290 - val_loss: 0.5835 - val_accuracy: 0.7943 - 198s/
epoch - 4s/step
```

Finally, we will save our model, and call this model in the second part of this project, where I incorporate this model into a dashboard.

Hide

```
model %>% save_model_tf("recycleModel")
```