Indoor Scene Recognition using CNN & DNN

Clementine Surya

2023-04-25

```
# Load necessary package
library(knitr)
library(jpeg)
library(keras)
library(tensorflow)
library(tidyverse)
library(imager)
library(caret)
```

Exploratory Data Analysis for Original Data and Aggregated Data

Justification on why I do data aggregation will be in Task 2 part

In this section, my aim is to perform an exploratory data analysis. To begin with, I attempt to extract the file name of each image. To achieve this, I locate the folder for each target class. The code snippet below demonstrates how to obtain the folder name within the “train” folder:

```
# Original data
folder_list <- list.files("data_indoor/train/")
folder_list
```

```
##  [1] "bathroom"      "bedroom"       "children_room" "closet"
##  [5] "corridor"      "dining_room"   "garage"        "kitchen"
##  [9] "living_room"   "stairs"
```

```
# Aggregated data
folder_list_agg <- list.files("data_indoor_aggregation/train/")
folder_list_agg
```

```
## [1] "living_spaces"  "private_spaces" "service_spaces" "utility_spaces"
```

I combine the folder name to the path or directory of the train folder to access the contents of each folder.

```
# Original data
folder_path <- paste0("data_indoor/train/", folder_list, "/")
folder_path
```

```
##  [1] "data_indoor/train/bathroom/"      "data_indoor/train/bedroom/"
##  [3] "data_indoor/train/children_room/" "data_indoor/train/closet/"
##  [5] "data_indoor/train/corridor/"      "data_indoor/train/dining_room/"
##  [7] "data_indoor/train/garage/"        "data_indoor/train/kitchen/"
##  [9] "data_indoor/train/living_room/"   "data_indoor/train/stairs/"
```

```
# Aggregated data
folder_path_agg <- paste0("data_indoor_aggregation/train/", folder_list_agg , "/")
folder_path_agg
```

```
## [1] "data_indoor_aggregation/train/living_spaces/"
## [2] "data_indoor_aggregation/train/private_spaces/"
## [3] "data_indoor_aggregation/train/service_spaces/"
## [4] "data_indoor_aggregation/train/utility_spaces/"
```

I use the map() function to loop or iterate and collect the file name for each folder. The map() will return a list to combine the file name from 10 different folders simply use the unlist() function.

```
# Original data
# get file name
file_name <- map(folder_path,
                 function(x) paste0(x, list.files(x))
                 ) %>%
  unlist()

# check how many images in the train folder
length(file_name)
```

```
## [1] 1713
```

```
# Aggregated data
# get file name
file_name_agg <- map(folder_path_agg,
                 function(x) paste0(x, list.files(x))
                 ) %>%
  unlist()

# check how many images in the train folder
length(file_name_agg)
```

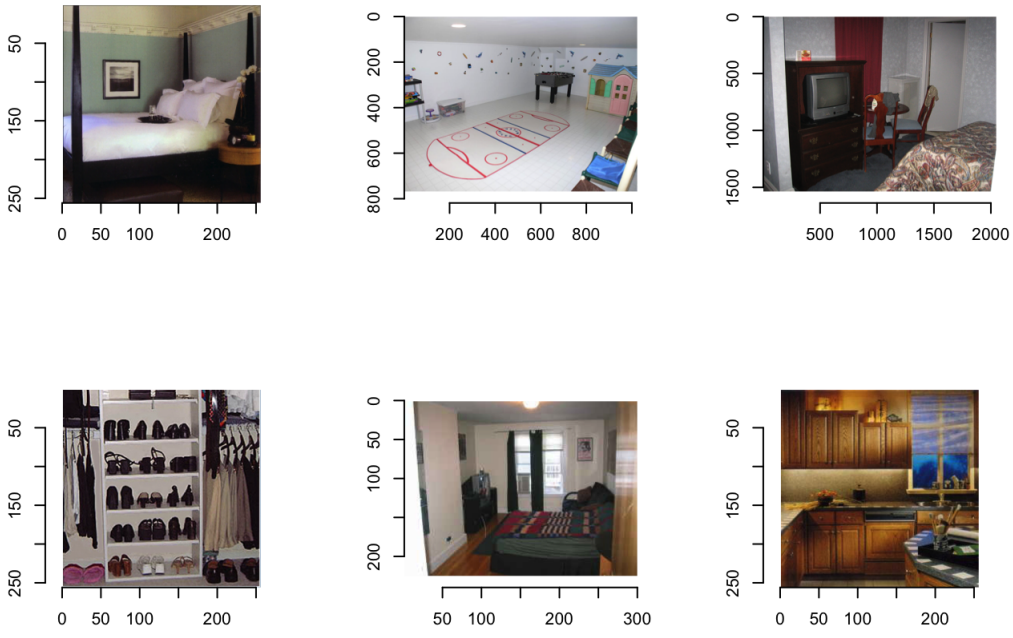
```
## [1] 1713
```

To check the content of the file, I use the load.image() function from the imager package. For example, I randomly visualize 6 images from the data.

```
# Randomly select image
set.seed(123)
sample_image <- sample(file_name, 6)

# Load image into R
img <- map(sample_image, load.image)

# Plot image
par(mfrow = c(2, 3)) # Create 2 x 3 image grid
map(img, plot)
```



```
## [[1]]
## Image. Width: 256 pix Height: 256 pix Depth: 1 Colour channels: 3
##
## [[2]]
## Image. Width: 1024 pix Height: 768 pix Depth: 1 Colour channels: 3
##
## [[3]]
## Image. Width: 2048 pix Height: 1536 pix Depth: 1 Colour channels: 3
##
## [[4]]
## Image. Width: 256 pix Height: 256 pix Depth: 1 Colour channels: 3
##
## [[5]]
## Image. Width: 300 pix Height: 225 pix Depth: 1 Colour channels: 3
##
## [[6]]
## Image. Width: 256 pix Height: 256 pix Depth: 1 Colour channels: 3
```

Check Image Dimension

Here, I analyze the distribution of image dimensions. On the following code, I create a function that will instantly get the height and width of an image and convert it into a data.frame.

```
# Function for acquiring width and height of an image
get_dim <- function(x){
  img <- load.image(x)

  df_img <- data.frame(height = height(img),
                       width = width(img),
                       filename = x
                       )

  return(df_img)
}
```

I sample 100 images from the file name and get the height and width of the image. I use sampling here because it will take a quite long time to load all images.

```
# Randomly get 100 sample images
set.seed(123)
sample_file <- sample(file_name, 100)
# Run the get_dim() function for each image
file_dim <- map_df(sample_file, get_dim)
```

Now, let's get the statistics for the image dimensions.

```
summary(file_dim)
```

```
##      height           width          filename
##  Min.   : 225.0   Min.   : 225.0   Length:100
##  1st Qu.: 256.0   1st Qu.: 256.0   Class :character
##  Median : 256.0   Median : 256.0   Mode  :character
##  Mean   : 456.6   Mean   : 481.3
##  3rd Qu.: 456.8   3rd Qu.: 560.5
##  Max.   :2592.0   Max.   :2592.0
```

The image data has a great variation in the dimension. Some images has less than 225 pixels in height and width respectively while others has up to 2592 pixels.

Data Augmentation

I transform all image into 64 x 64 pixels as per instructions. Bigger dimensions will have more features but will also take longer time to train. However, if the image size is too small, we will lose a lot of information from the data. So balancing this trade-off is the art of data preprocessing in image classification.

I also set the batch size for the data so the model will be updated every time it finished training on a single batch. Here, I set the batch size to 20.

```
# Desired height and width of images
target_size <- c(64, 64)

# Batch size for training the model
batch_size <- 20
```

As our training set is relatively small, I do data augmentation with below settings. By rotating the images, shifting them horizontally and vertically, shearing them, zooming in and out, and flipping them horizontally, I introduce variations that are common in real-world images of rooms and scenes. These variations can help the model learn to recognize the important features of the different types of rooms/scenes, regardless of their orientation or position within the image.

```
# Image Generator
# for training data
train_datagen <- image_data_generator(rescale = 1/255,
                                      rotation_range = 40,
                                      width_shift_range = 0.2,
                                      height_shift_range = 0.2,
                                      shear_range = 0.2,
                                      zoom_range = 0.2,
                                      horizontal_flip = TRUE)
# for validation data
validation_datagen <- image_data_generator(rescale = 1/255)

# for test data
test_datagen <- image_data_generator(rescale = 1/255)
```

Now I insert our image data into the generator using the flow_images_from_directory(). From this process, I get the augmented image both training, validation, and test data.

```
# Original data
set.seed(123)
# Train data generator with data augmentation
train_generator <- flow_images_from_directory(directory = "data_indoor/train/", # Folder of the data
                                                target_size = target_size, # target of the image dimensi
on (64 x 64)
                                                color_mode = "rgb", # use RGB color
                                                batch_size = batch_size ,
                                                seed = 123,  # set random seed
                                                generator = train_datagen
                                                )

# Validation data generator with data augmentation
validation_generator <- flow_images_from_directory(directory = "data_indoor/validation/",
                                                   target_size = target_size,
                                                   color_mode = "rgb",
                                                   batch_size = batch_size ,
                                                   seed = 123,
                                                   generator = validation_datagen
                                                   )

test_generator <- flow_images_from_directory(directory = "data_indoor/test/",
                                                   target_size = target_size,
                                                   color_mode = "rgb",
                                                   batch_size = batch_size ,
                                                   seed = 123,
                                                   generator = test_datagen
                                                   )
```

```
# Aggregated data
set.seed(123)
# Train data generator with data augmentation
train_generator_agg <- flow_images_from_directory(directory = "data_indoor_aggregation/train/", # Folder of
the data
                                                  target_size = target_size, # target of the image dimensi
on (64 x 64)
                                                  color_mode = "rgb", # use RGB color
                                                  batch_size = batch_size ,
                                                  seed = 123,  # set random seed
                                                  generator = train_datagen
                                                  )

# Validation data generator with data augmentation
validation_generator_agg <- flow_images_from_directory(directory = "data_indoor_aggregation/validation/",
                                                target_size = target_size,
                                                color_mode = "rgb",
                                                batch_size = batch_size ,
                                                seed = 123,
                                                generator = validation_datagen
                                                )

test_generator_agg <- flow_images_from_directory(directory = "data_indoor_aggregation/test/",
                                                target_size = target_size,
                                                color_mode = "rgb",
                                                batch_size = batch_size ,
                                                seed = 123,
                                                generator = test_datagen
                                                )
```

Here, I collect some information from the generator and check the class proportion of the train dataset. The index correspond to each labels of the target variable and ordered alphabetically (bathroom, bedroom, children room, closet, corridor, dining room, garage, kitchen, living room, stairs).

```
# Original data
# Number of training samples
train_samples <- train_generator$n

# Number of validation samples
valid_samples <- validation_generator$n

# Number of target classes/categories
output_n <- n_distinct(train_generator$classes)

# Get the class proportion
table("\nFrequency" = factor(train_generator$classes)) %>%
  prop.table()
```

```
##
## Frequency
##          0          1          2          3          4          5          6
## 0.05720957 0.19322825 0.03269119 0.03969644 0.10099241 0.07997665 0.03035610
##          7          8          9
## 0.21424402 0.20607122 0.04553415
```

Based on the frequency above, it appears that the classes for predicting indoor scenes are imbalanced. This class imbalance can pose several challenges and disadvantages for this task. It can affect the overall performance of the predictive model, as the model may become biased towards the majority classes and fail to accurately predict the minority classes. This can lead to lower accuracy and recall scores for the minority classes, making it difficult to achieve overall high performance.

```
# Aggregated data
# Number of training samples
train_samples_agg <- train_generator_agg$n

# Number of validation samples
valid_samples_agg <- validation_generator_agg$n

# Number of target classes/categories
output_n_agg <- n_distinct(train_generator_agg$classes)

# Get the class proportion
table("\nFrequency" = factor(train_generator_agg$classes)) %>%
  prop.table()
```

```
##
## Frequency
##         0         1         2         3
## 0.2860479 0.2259194 0.2714536 0.2165791
```

The frequency distribution in aggregated dataset provided seems to indicate a relatively balanced distribution across the four classes, with each class accounting for approximately 22-28% of the total data. This balance could be viewed as an improvement from the original dataset, and may lead to more accurate model performance.

Task

The task is to build a predictive model to predict the type of indoor scene from the image data.

Task 1

In this section, I develop a total of six deep learning systems, where each one possesses distinct configurations, hyperparameters, and training settings. Among these, **four systems will be built using two DNN models and two CNN models, utilizing the Original Data with ten classes**. Meanwhile, **the remaining two systems will involve two CNN models trained with Aggregated Data, featuring four classes**.

Model 1

- Input layer: layer_flatten layer which takes a 3-dimensional input tensor of size 64 x 64 x 3 and flattens it into a 1-dimensional tensor.
- Hidden layers: Three layer_dense layers with 256, 128, and 64 units, respectively, and ReLU activation function. The kernel_regularizer is set to L2 regularization with a coefficient of 0.001 to help prevent overfitting.
- Output layer: layer_dense layer with units = 10 and activation = softmax which produces a probability distribution over the output classes.
- Compilation: The model is compiled with the categorical_crossentropy loss function, optimizer_sgd optimizer with a learning rate of 0.001 and momentum of 0.9, and accuracy metric for evaluation.
- The “fit” code block contains a early stopping regularization that halts the model training when it starts to overfit the training data. The value of 5 for “early_stopping_5” indicates that the training process will stop if the validation loss does not decrease for five consecutive epochs.

```
set.seed(123)
tensorflow::tf$random$set_seed(123)

# Deploy model
model_dnn_1 <- keras_model_sequential(name = "model_dnn_1") %>%
  # Flatten the input
  layer_flatten(input_shape = c(target_size, 3)) %>%
  # Dense Layers
  layer_dense(units = 256, activation = "relu", kernel_regularizer = regularizer_l2(0.001)) %>%
  layer_dense(units = 128, activation = "relu", kernel_regularizer = regularizer_l2(0.001)) %>%
  layer_dense(units = 64, activation = "relu", kernel_regularizer = regularizer_l2(0.001)) %>%
  # Output Layer
  layer_dense(units = output_n, activation = "softmax")

# Compile
model_dnn_1 %>%
  compile(
    loss = "categorical_crossentropy",
    optimizer = optimizer_sgd(learning_rate = 0.001, momentum = 0.9),
    metrics = "accuracy"
  )

# Define the early stopping callback
early_stopping_5 <- callback_early_stopping(monitor = "val_loss", patience = 5)

# Train with data augmentation
fit_dnn_1 <- model_dnn_1 %>%
  fit(
    # training data
    train_generator,
    # training epochs
    steps_per_epoch = as.integer(train_samples / batch_size),
    epochs = 100,
    # validation data
    validation_data = validation_generator,
    validation_steps = as.integer(valid_samples / batch_size),
    # early stopping callback
    callbacks = list(early_stopping_5)
  )
```

Model 2

- Input layer: layer_flatten layer which takes a 3-dimensional input tensor of size 64 x 64 x 3 and flattens it into a 1-dimensional tensor.
- Hidden layers: Five layer_dense layers with 256, 128, 64, 32, and 16 units, respectively, and ReLU activation function. Each dense layer is followed by a layer_dropout layer with a rate of 0.3, which randomly drops out some of the activations to help prevent overfitting. The use of five layer_dense layers allows the model to learn multiple levels of abstraction from the input data, potentially improving its ability to capture complex relationships and patterns.
- Output layer: layer_dense layer with units = 10 and activation = softmax which produces a probability distribution over the output classes.
- Compilation: The model is compiled with the categorical_crossentropy loss function, optimizer_rmsprop optimizer with a learning rate of 0.001, and accuracy metric for evaluation.
- The “fit” code block contains a early stopping regularization that halts the model training when it starts to overfit the training data. The value of 5 for “early_stopping_5” indicates that the training process will stop if the validation loss does not decrease for five consecutive epochs.

```
set.seed(123)
tensorflow::tf$random$set_seed(123)

# Deploy model
model_dnn_2 <- keras_model_sequential(name = "model_dnn_2") %>%
  # Flatten the input
  layer_flatten(input_shape = c(target_size, 3)) %>%
  # Dense Layers
  layer_dense(units = 256, activation = "relu") %>%
  layer_dropout(rate = 0.3) %>%
  layer_dense(units = 128, activation = "relu") %>%
  layer_dropout(rate = 0.3) %>%
  layer_dense(units = 64, activation = "relu") %>%
  layer_dropout(rate = 0.3) %>%
  layer_dense(units = 32, activation = "relu") %>%
  layer_dropout(rate = 0.3) %>%
  layer_dense(units = 16, activation = "relu") %>%
  layer_dropout(rate = 0.3) %>%
  # Output Layer
  layer_dense(units = output_n, activation = "softmax")

# Compile
model_dnn_2 %>%
  compile(
    loss = "categorical_crossentropy",
    optimizer = optimizer_rmsprop(learning_rate = 0.001),
    metrics = "accuracy"
  )

# Define the early stopping callback
early_stopping_5 <- callback_early_stopping(monitor = "val_loss", patience = 5)

# Train with data augmentation
fit_dnn_2 <- model_dnn_2 %>%
  fit(
    # training data
    train_generator,
    # training epochs
    steps_per_epoch = as.integer(train_samples / batch_size),
    epochs = 100,
    # validation data
    validation_data = validation_generator,
    validation_steps = as.integer(valid_samples / batch_size),
    # early stopping callback
    callbacks = list(early_stopping_5)
  )
```

Model 3

- Convolutional Layers: This model consists of two convolution layers, each followed by a max pooling layer, to extract features from the input image. The filters increase in number as we move deeper into the network (64 and 128), allowing the model to learn more complex features in the higher layers. The kernel sizes used are 3 x 3. The activation function used is ReLU. The input shape is 64 x 64 x 3.
- Fully Connected Layers: After the convolution layers, the output is flattened and fed into three fully connected layers. The first fully connected layer takes the flattened vector as input and has 64 hidden units. The ReLU activation function is used again, and a regularization term is added to the weights of this layer to prevent overfitting. The second fully connected layer has 32 hidden units and uses the ReLU activation function. Like the previous layer, a regularization term is added to the weights. The output layer, which has 10 hidden units and uses the softmax activation function to produce the probability distribution over the classes.
- Compilation: This compiles the model and sets the loss function to categorical cross-entropy, the optimizer to Stochastic Gradient Descent (SGD) with a learning rate of 0.001 and momentum of 0.9, and the metric to be used during training and evaluation to accuracy.
- The “fit” code block contains a early stopping regularization that halts the model training when it starts to overfit the training data. The value of 10 for “early_stopping_10” indicates that the training process will stop if the validation loss does not decrease for ten consecutive epochs.

```
set.seed(123)
tensorflow::tf$random$set_seed(123)

# Deploy model
model_cnn_1 <- keras_model_sequential(name = "model_cnn_1") %>%
  # Convolution Layers
  layer_conv_2d(filters = 64, kernel_size = c(3,3), activation = "relu",
                input_shape = c(target_size, 3)) %>%
  layer_max_pooling_2d(pool_size = c(2,2)) %>%
  layer_conv_2d(filters = 128, kernel_size = c(3,3), activation = "relu") %>%
  layer_max_pooling_2d(pool_size = c(2,2)) %>%
  # Flatten the input
  layer_flatten() %>%
  # Dense Layers
  layer_dense(units = 64, activation = "relu", kernel_regularizer = regularizer_l2(0.01)) %>%
  layer_dense(units = 32, activation = "relu", kernel_regularizer = regularizer_l2(0.01)) %>%
  # Output Layer
  layer_dense(units = output_n, activation = "softmax", name = "Output")

# Compile
model_cnn_1 %>%
  compile(
    loss = "categorical_crossentropy",
    optimizer = optimizer_sgd(learning_rate = 0.001, momentum = 0.9),
    metrics = "accuracy"
  )

# Define the early stopping callback
early_stopping_10 <- callback_early_stopping(monitor = "val_loss", patience = 10)

# Train with data augmentation
fit_cnn_1 <- model_cnn_1  %>%
  fit(
    # training data
    train_generator,
    # training epochs
    steps_per_epoch = as.integer(train_samples / batch_size),
    epochs = 100,
    # validation data
    validation_data = validation_generator,
    validation_steps = as.integer(valid_samples / batch_size),
    # early stopping callback
    callbacks = list(early_stopping_10)
  )
```

Model 4

- Convolutional Layers: This model consists of four convolutional layers, each followed by a max pooling layer, to extract features from the input image. The filters increase in number as we move deeper into the network (16, 32, 64, 128), allowing the model to learn more complex features in the higher layers. The kernel sizes used are 3 x 3. The activation function used is ReLU. The input shape is 64 x 64 x 3. Additionally, batch normalization is applied after each convolutional layer to help with convergence and generalization.
- Fully Connected Layers: After the convolutional layers, the output is flattened and fed into two fully connected layers. The first fully connected layer has 64 units and uses the ReLU activation function. This layer allows the model to learn high-level representations of the features extracted by the convolutional layers. A dropout layer with a rate of 0.25 is applied after this layer to prevent overfitting. The second fully connected layer has 10 units, which is the number of classes in the dataset, and uses the softmax activation function to generate the final class probabilities.
- Compilation: The model is compiled using the categorical cross-entropy loss function, which is commonly used for multi-class classification problems. The Adam optimizer with a learning rate of 0.001 is used, which has been shown to perform well in many deep learning applications. Finally, the accuracy metric is used to evaluate the model's performance during training and validation.

```
set.seed(123)
tf$random$set_seed(123)

# Deploy model
model_cnn_2 <- keras_model_sequential(name = "model_cnn_2") %>%
  # Convolution Layers
  layer_conv_2d(filters = 16, kernel_size = c(3,3), activation = "relu",
                input_shape = c(target_size, 3)) %>%
  layer_max_pooling_2d(pool_size = c(2,2)) %>%
  layer_batch_normalization() %>%
  layer_conv_2d(filters = 32, kernel_size = c(3,3), activation = "relu") %>%
  layer_max_pooling_2d(pool_size = c(2,2)) %>%
  layer_batch_normalization() %>%
  layer_conv_2d(filters = 64, kernel_size = c(3,3), activation = "relu") %>%
  layer_max_pooling_2d(pool_size = c(2,2)) %>%
  layer_batch_normalization() %>%
  layer_conv_2d(filters = 128, kernel_size = c(3,3), activation = "relu") %>%
  layer_max_pooling_2d(pool_size = c(2,2)) %>%
  layer_batch_normalization() %>%
  # Flatten the input
  layer_flatten() %>%
  layer_batch_normalization() %>%
  layer_dense(units = 64, activation = "relu") %>%
  layer_dropout(0.25) %>%
  layer_dense(units = output_n, activation = "softmax")

# Compile
model_cnn_2  %>%
  compile(
    loss = "categorical_crossentropy",
    optimizer = optimizer_adam(learning_rate = 0.001),
    metrics = "accuracy"
  )

# Train with data augmentation
fit_cnn_2 <- model_cnn_2 %>%
  fit(
  # training data
  train_generator,
  # training epochs
  steps_per_epoch = as.integer(train_samples / batch_size),
  epochs = 100,
  # validation data
  validation_data = validation_generator,
  validation_steps = as.integer(valid_samples / batch_size),
)
```

Model 5

The configuration for model 5 is identical to that of model 3, except that it will be trained on aggregated data instead of the original dataset. Consequently, the output layer of model 5 will consist of four classes instead of ten classes.

```
set.seed(123)
tensorflow::tf$random$set_seed(123)

# Deploy model
model_cnn_agg_1 <- keras_model_sequential(name = "model_cnn_agg_1") %>%
  # Convolution Layers
  layer_conv_2d(filters = 64, kernel_size = c(3,3), activation = "relu",
                input_shape = c(target_size, 3)) %>%
  layer_max_pooling_2d(pool_size = c(2,2)) %>%
  layer_conv_2d(filters = 128, kernel_size = c(3,3), activation = "relu") %>%
  layer_max_pooling_2d(pool_size = c(2,2)) %>%
  # Flatten the input
  layer_flatten() %>%
  # Dense Layers
  layer_dense(units = 64, activation = "relu", kernel_regularizer = regularizer_l2(0.01)) %>%
  layer_dense(units = 32, activation = "relu", kernel_regularizer = regularizer_l2(0.01)) %>%
  # Output Layer
  layer_dense(units = output_n_agg, activation = "softmax", name = "Output")

# Compile
model_cnn_agg_1 %>%
  compile(
    loss = "categorical_crossentropy",
    optimizer = optimizer_sgd(learning_rate = 0.001, momentum = 0.9),
    metrics = "accuracy"
  )

# define the early stopping callback
early_stopping_10 <- callback_early_stopping(monitor = "val_loss", patience = 10)

# Train with data augmentation
fit_cnn_agg_1 <- model_cnn_agg_1  %>%
  fit(
    # training data
    train_generator_agg,
    # training epochs
    steps_per_epoch = as.integer(train_samples_agg / batch_size),
    epochs = 100,
     # validation data
    validation_data = validation_generator_agg,
    validation_steps = as.integer(valid_samples_agg / batch_size),
    # early stopping callback
    callbacks = list(early_stopping_10)
  )
```

Model 6

The configuration for model 5 is identical to that of model 4, except that it will be trained on aggregated data instead of the original dataset. Consequently, the output layer of model 5 will consist of four classes instead of ten classes.

```
set.seed(123)
tf$random$set_seed(123)

# Deploy model
model_cnn_agg_2 <- keras_model_sequential(name = "model_cnn_agg_2") %>%
  # Convolution Layers
  layer_conv_2d(filters = 16, kernel_size = c(3,3), activation = "relu",
                input_shape = c(target_size, 3)) %>%
  layer_max_pooling_2d(pool_size = c(2,2)) %>%
  layer_batch_normalization() %>%
  layer_conv_2d(filters = 32, kernel_size = c(3,3), activation = "relu") %>%
  layer_max_pooling_2d(pool_size = c(2,2)) %>%
  layer_batch_normalization() %>%
  layer_conv_2d(filters = 64, kernel_size = c(3,3), activation = "relu") %>%
  layer_max_pooling_2d(pool_size = c(2,2)) %>%
  layer_batch_normalization() %>%
  layer_conv_2d(filters = 128, kernel_size = c(3,3), activation = "relu") %>%
  layer_max_pooling_2d(pool_size = c(2,2)) %>%
  layer_batch_normalization() %>%
  # Flatten the input
  layer_flatten() %>%
  layer_batch_normalization() %>%
  # Dense Layers
  layer_dense(units = 64, activation = "relu") %>%
  layer_dropout(0.25) %>%
  # Output Layer
  layer_dense(units = output_n_agg, activation = "softmax")

# Compile
model_cnn_agg_2  %>%
  compile(
    loss = "categorical_crossentropy",
    optimizer = optimizer_adam(learning_rate = 0.001),
    metrics = "accuracy"
  )

# Train with data augmentation
fit_cnn_agg_2 <- model_cnn_agg_2 %>%
  fit(
  # training data
  train_generator_agg,
  # training epochs
  steps_per_epoch = as.integer(train_samples_agg / batch_size),
  epochs = 100,
  # validation data
  validation_data = validation_generator_agg,
  validation_steps = as.integer(valid_samples_agg / batch_size),
)
```

Task 2

Here, I conduct a comparison of the six deep learning systems developed earlier. This comparison will entail an evaluation of their relative strengths and weaknesses in terms of training and predictive performance. By conducting this assessment, I aim to identify the best-performing model for predicting the type of indoor scene from the data.

Model 1 - Performance Assessment

```
model_dnn_1 %>%
  evaluate(train_generator, step = as.integer(train_samples / batch_size))
```

```
##      loss  accuracy
## 2.5595877 0.2805882
```

```
model_dnn_1 %>%
  evaluate(validation_generator, step = as.integer(valid_samples / batch_size))
```

```
##      loss  accuracy
## 2.5986924 0.2738095
```

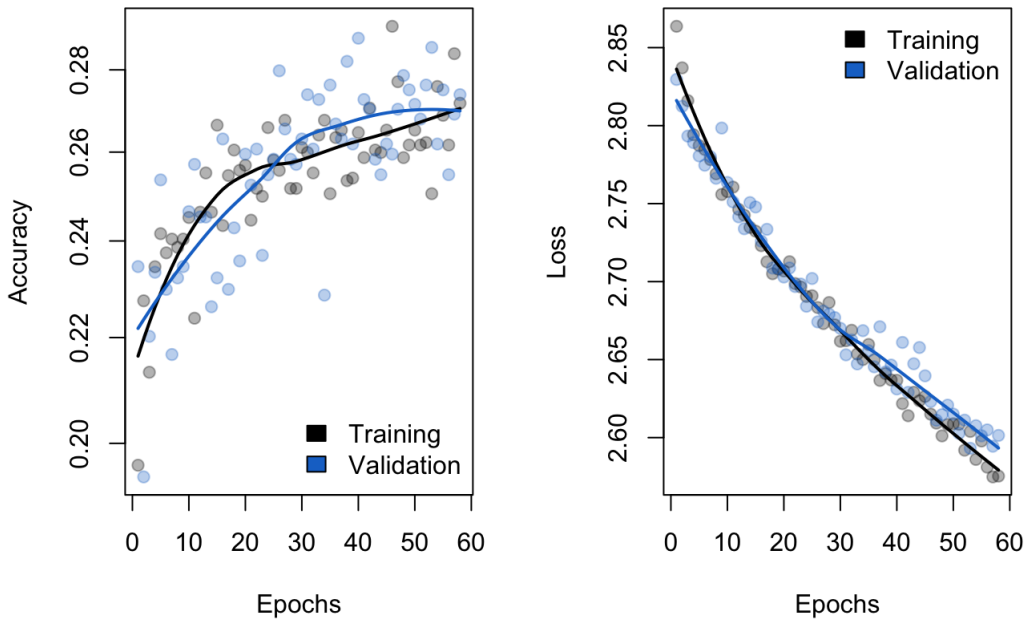
```
# to add a smooth line to points
smooth_line <- function(y) {
x <- 1:length(y)
out <- predict( loess(y ~ x) )
return(out)
}

# check learning curves
out_dnn_1 <- cbind(fit_dnn_1$metrics$accuracy,
fit_dnn_1$metrics$val_accuracy,
fit_dnn_1$metrics$loss,
fit_dnn_1$metrics$val_loss)

cols <- c("black", "dodgerblue3")
par(mfrow = c(1,2))

# accuracy
matplot(out_dnn_1[,1:2], pch = 19, ylab = "Accuracy", xlab = "Epochs",
col = adjustcolor(cols, 0.3),
log = "y")
matlines(apply(out_dnn_1[,1:2], 2, smooth_line), lty = 1, col = cols, lwd = 2)
legend("bottomright", legend = c("Training", "Validation"),
fill = cols, bty = "n")

# loss
matplot(out_dnn_1[,3:4], pch = 19, ylab = "Loss", xlab = "Epochs",
col = adjustcolor(cols, 0.3))
matlines(apply(out_dnn_1[,3:4], 2, smooth_line), lty = 1, col = cols, lwd = 2)
legend("topright", legend = c("Training", "Validation"),
fill = cols, bty = "n")
```



The accuracy assessment provided for Model 1 (model_dnn_1) shows that the model has a low accuracy both on the training and validation sets. The loss values indicates that the model is not overfitting as the training and validation loss values are similar. However, the low accuracy on both sets suggests that the model is not performing well in capturing the patterns and relationships in the data.

Model 2 - Performance Assessment

```
model_dnn_2 %>%
  evaluate(train_generator, step = as.integer(train_samples / batch_size))
```

```
##      loss  accuracy
## 2.0581560 0.2152941
```

```
model_dnn_2 %>%
  evaluate(validation_generator, step = as.integer(valid_samples / batch_size))
```

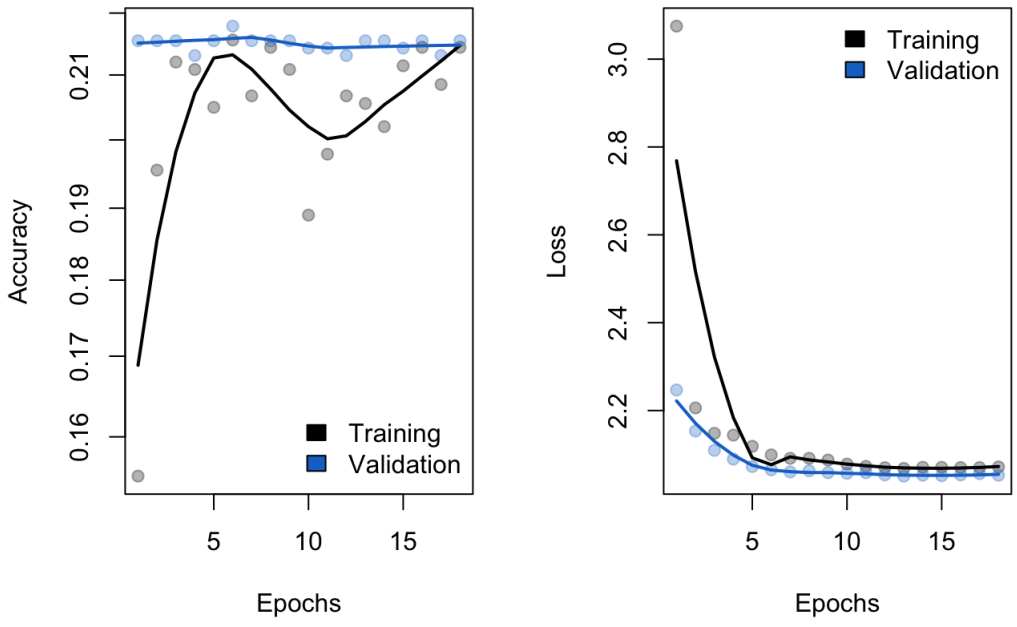
```
##      loss  accuracy
## 2.0517149 0.2166667
```

```
# check learning curves
out_dnn_2 <- cbind(fit_dnn_2$metrics$accuracy,
fit_dnn_2$metrics$val_accuracy,
fit_dnn_2$metrics$loss,
fit_dnn_2$metrics$val_loss)

cols <- c("black", "dodgerblue3")
par(mfrow = c(1,2))

# accuracy
matplot(out_dnn_2[,1:2], pch = 19, ylab = "Accuracy", xlab = "Epochs",
col = adjustcolor(cols, 0.3),
log = "y")
matlines(apply(out_dnn_2[,1:2], 2, smooth_line), lty = 1, col = cols, lwd = 2)
legend("bottomright", legend = c("Training", "Validation"),
fill = cols, bty = "n")

# loss
matplot(out_dnn_2[,3:4], pch = 19, ylab = "Loss", xlab = "Epochs",
col = adjustcolor(cols, 0.3))
matlines(apply(out_dnn_2[,3:4], 2, smooth_line), lty = 1, col = cols, lwd = 2)
legend("topright", legend = c("Training", "Validation"),
fill = cols, bty = "n")
```



The commentary for Model 2 (model_dnn_2) for the accuracy and loss performance are the same as Model 1. It shows low accuracy both on the training and validation sets. Hence, the model is not performing well in capturing the patterns and relationships in the data.

The two models appear to be performing similarly in terms of accuracy, with Model 1 having a slightly higher accuracy score on both the training and validation sets. Based on the provided performance assessments and settings, it can be inferred that Model 1 has a relatively simple architecture compared to Model 2, with only three dense layers and L2 regularization applied to the kernel weights. However, Model 1 achieved higher accuracy than Model 2 on both the training and validation sets. This suggests that in this specific task, a deeper and more complex DNN architecture may not necessarily lead to improved performance.

Model 3 - Performance Assessment

```
model_cnn_1 %>%
  evaluate(train_generator, step = as.integer(train_samples / batch_size))
```

```
##      loss  accuracy
## 1.8343490 0.3682353
```

```
model_cnn_1 %>%
  evaluate(validation_generator, step = as.integer(valid_samples / batch_size))
```

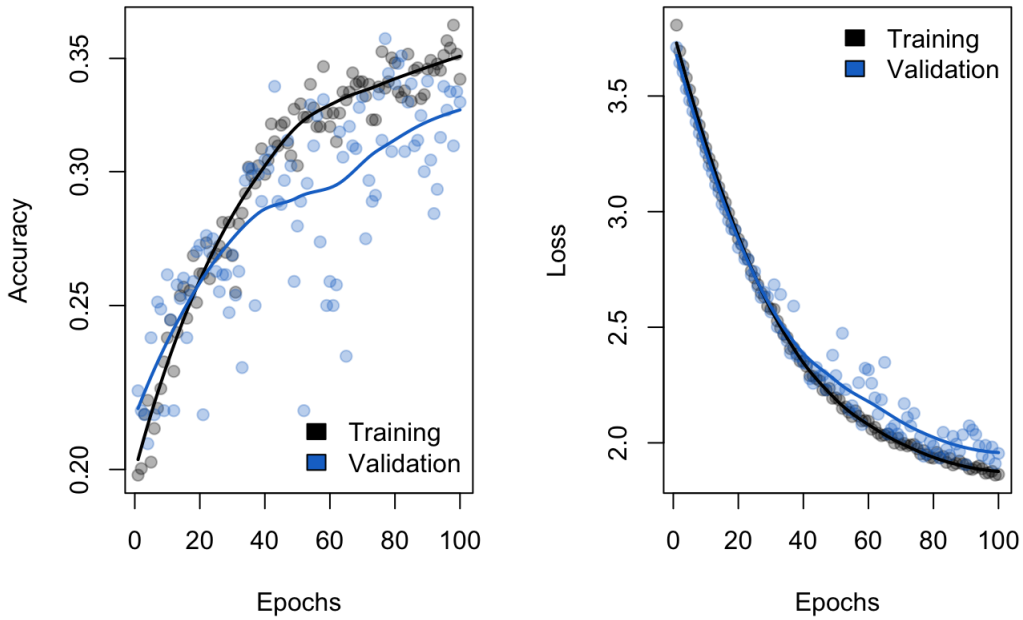
```
##     loss accuracy
## 1.956794 0.327381
```

```
# check learning curves
out_cnn_1 <- cbind(fit_cnn_1$metrics$accuracy,
fit_cnn_1$metrics$val_accuracy,
fit_cnn_1$metrics$loss,
fit_cnn_1$metrics$val_loss)

cols <- c("black", "dodgerblue3")
par(mfrow = c(1,2))

# accuracy
matplot(out_cnn_1[,1:2], pch = 19, ylab = "Accuracy", xlab = "Epochs",
col = adjustcolor(cols, 0.3),
log = "y")
matlines(apply(out_cnn_1[,1:2], 2, smooth_line), lty = 1, col = cols, lwd = 2)
legend("bottomright", legend = c("Training", "Validation"),
fill = cols, bty = "n")

# loss
matplot(out_cnn_1[,3:4], pch = 19, ylab = "Loss", xlab = "Epochs",
col = adjustcolor(cols, 0.3))
matlines(apply(out_cnn_1[,3:4], 2, smooth_line), lty = 1, col = cols, lwd = 2)
legend("topright", legend = c("Training", "Validation"),
fill = cols, bty = "n")
```



Model 4 - Performance Assessment

```
model_cnn_2 %>%
  evaluate(train_generator, step = as.integer(train_samples / batch_size))
```

```
##      loss  accuracy
## 1.1003159 0.5941176
```

```
model_cnn_2 %>%
  evaluate(validation_generator, step = as.integer(valid_samples / batch_size))
```

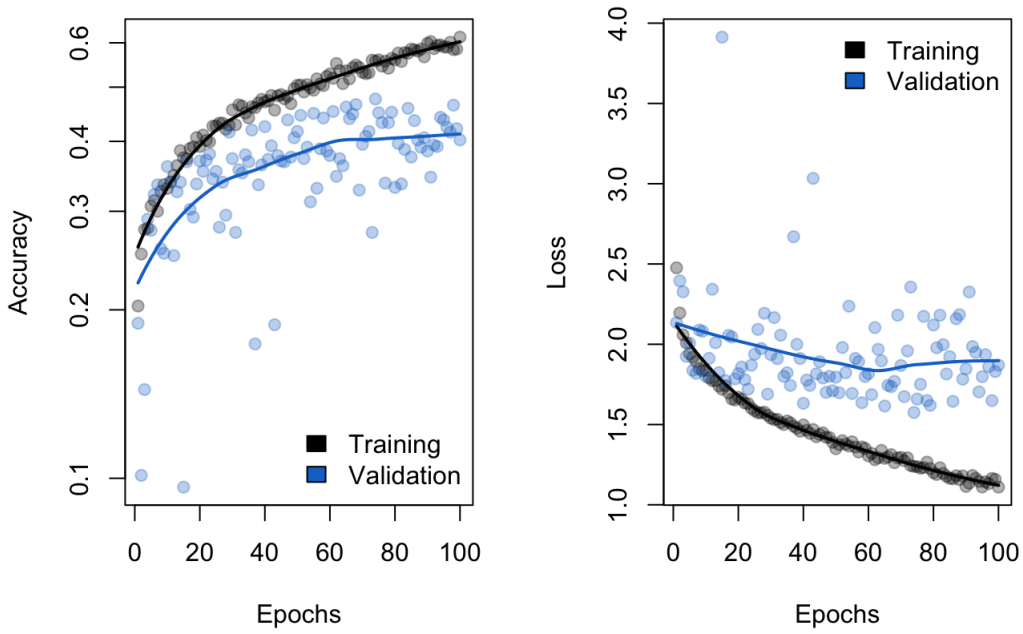
```
##      loss  accuracy
## 1.8635082 0.4047619
```

```
# check learning curves
out_cnn_2 <- cbind(fit_cnn_2$metrics$accuracy,
fit_cnn_2$metrics$val_accuracy,
fit_cnn_2$metrics$loss,
fit_cnn_2$metrics$val_loss)

cols <- c("black", "dodgerblue3")
par(mfrow = c(1,2))

# accuracy
matplot(out_cnn_2[,1:2], pch = 19, ylab = "Accuracy", xlab = "Epochs",
col = adjustcolor(cols, 0.3),
log = "y")
matlines(apply(out_cnn_2[,1:2], 2, smooth_line), lty = 1, col = cols, lwd = 2)
legend("bottomright", legend = c("Training", "Validation"),
fill = cols, bty = "n")

# loss
matplot(out_cnn_2[,3:4], pch = 19, ylab = "Loss", xlab = "Epochs",
col = adjustcolor(cols, 0.3))
matlines(apply(out_cnn_2[,3:4], 2, smooth_line), lty = 1, col = cols, lwd = 2)
legend("topright", legend = c("Training", "Validation"),
fill = cols, bty = "n")
```



Based on the evaluation results, it seems that Model 4 (model_cnn_2) performs better than Model 3 (model_cnn_1) in terms of accuracy and loss. Specifically, Model 4 achieved higher accuracy scores for both training and validation datasets, and it also had a lower loss compared to Model 3. This suggests that the more complex architecture used in Model 4, with additional convolution and pooling layers, may have allowed for better feature extraction and improved the model's ability to capture complex patterns and relationships in the data. However, we can also see that both models exhibit signs of overfitting, as the accuracy on the validation set is lower than the accuracy on the training set despite the regularization being implemented in the model.

From the results I obtain from the four models above, CNN tends to perform better than DNN in this indoor scene recognition task. This is because CNN are specifically designed to work with images, and they are able to automatically extract important features from the images by using convolution and pooling layers. On the other hand, DNNs are designed to work with general input data and do not have the specialized layers that are needed for image processing.

However, the validation accuracy that I obtained for the four models were low, approximately 20-40%. To gain a more structured and comprehensible understanding of the different rooms and spaces in a household and their relationships based on their primary function, I tried to implement the same architecture of model 3 and model 4 to the aggregated data. Thus, only the output units are changed.

I classified these spaces into four groups: Living Spaces, Private Spaces, Service Spaces, and Utility Spaces. Living spaces are intended for socializing and relaxation, such as the living and dining room. Private spaces are intended for personal use and rest, such as bedrooms and children's rooms. Service spaces are intended for functional purposes like cooking and hygiene, such as bathrooms and kitchens. Finally, Utility spaces are intended for storage and circulation, such as closets, garages, corridors, and stairs. By categorizing these spaces based on their primary function, it becomes easier to understand how they fit into the overall design and purpose of a home or building.

Let's evaluate the validation performance of the model.

Model 5 - Performance Assessment

```
# Model 5 - Performance Assessment
model_cnn_agg_1 %>%
  evaluate(train_generator_agg, step = as.integer(train_samples_agg / batch_size))
```

```
##      loss  accuracy
## 1.2902460 0.4847059
```

```
model_cnn_agg_1 %>%
  evaluate(validation_generator_agg, step = as.integer(valid_samples_agg / batch_size))
```

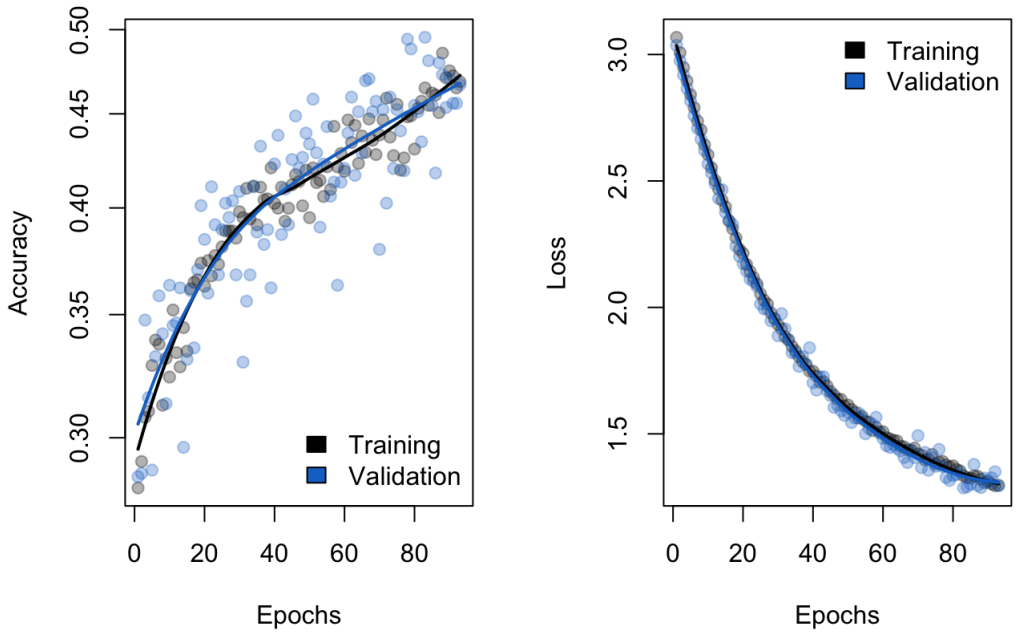
```
##      loss  accuracy
## 1.2954479 0.4654762
```

```
# check learning curves
out_cnn_agg_1 <- cbind(fit_cnn_agg_1$metrics$accuracy,
fit_cnn_agg_1$metrics$val_accuracy,
fit_cnn_agg_1$metrics$loss,
fit_cnn_agg_1$metrics$val_loss)

cols <- c("black", "dodgerblue3")
par(mfrow = c(1,2))

# accuracy
matplot(out_cnn_agg_1[,1:2], pch = 19, ylab = "Accuracy", xlab = "Epochs",
col = adjustcolor(cols, 0.3),
log = "y")
matlines(apply(out_cnn_agg_1[,1:2], 2, smooth_line), lty = 1, col = cols, lwd = 2)
legend("bottomright", legend = c("Training", "Validation"),
fill = cols, bty = "n")

# loss
matplot(out_cnn_agg_1[,3:4], pch = 19, ylab = "Loss", xlab = "Epochs",
col = adjustcolor(cols, 0.3))
matlines(apply(out_cnn_agg_1[,3:4], 2, smooth_line), lty = 1, col = cols, lwd = 2)
legend("topright", legend = c("Training", "Validation"),
fill = cols, bty = "n")
```



Model 6 - Performance Assessment

```
model_cnn_agg_2 %>%
  evaluate(train_generator_agg, step = as.integer(train_samples_agg / batch_size))
```

```
##      loss  accuracy
## 0.9490402 0.6294118
```

```
model_cnn_agg_2 %>%
  evaluate(validation_generator_agg, step = as.integer(valid_samples_agg / batch_size))
```

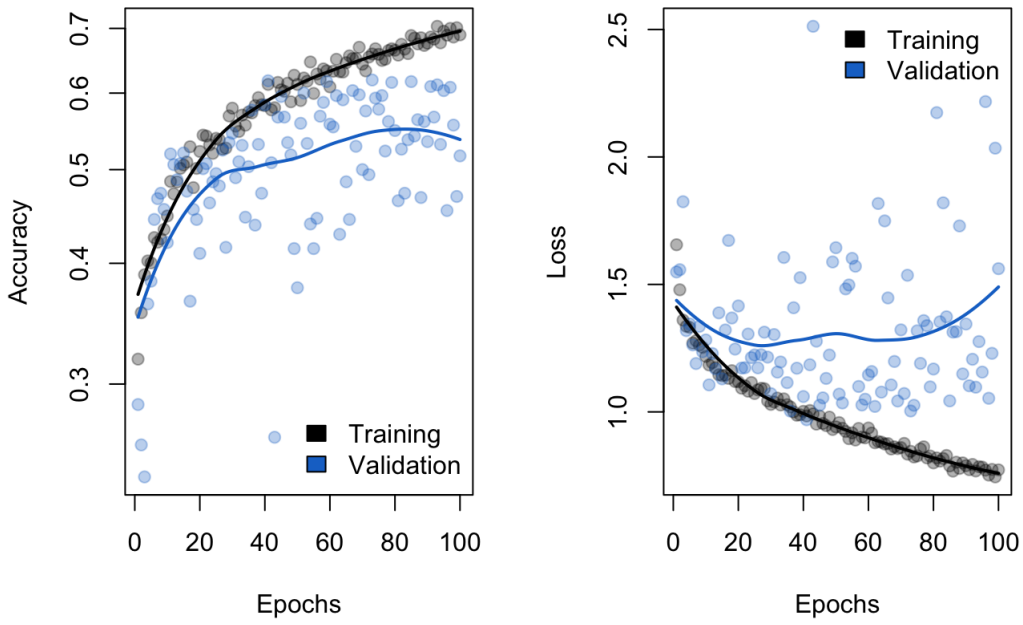
```
##      loss  accuracy
## 1.5726460 0.5142857
```

```
# check learning curves
out_cnn_agg_2 <- cbind(fit_cnn_agg_2$metrics$accuracy,
fit_cnn_agg_2$metrics$val_accuracy,
fit_cnn_agg_2$metrics$loss,
fit_cnn_agg_2$metrics$val_loss)

cols <- c("black", "dodgerblue3")
par(mfrow = c(1,2))

# accuracy
matplot(out_cnn_agg_2[,1:2], pch = 19, ylab = "Accuracy", xlab = "Epochs",
col = adjustcolor(cols, 0.3),
log = "y")
matlines(apply(out_cnn_agg_2[,1:2], 2, smooth_line), lty = 1, col = cols, lwd = 2)
legend("bottomright", legend = c("Training", "Validation"),
fill = cols, bty = "n")

# loss
matplot(out_cnn_agg_2[,3:4], pch = 19, ylab = "Loss", xlab = "Epochs",
col = adjustcolor(cols, 0.3))
matlines(apply(out_cnn_agg_2[,3:4], 2, smooth_line), lty = 1, col = cols, lwd = 2)
legend("topright", legend = c("Training", "Validation"),
fill = cols, bty = "n")
```



Based on the evaluation results provided, Model 6 (model_cnn_agg_2) is performing better than Model 5 (model_cnn_agg_1) in terms of predictive performance on both the training and validation datasets. However, the difference between the training and validation accuracy values indicates that Model 6 may be overfitting to the training data. In particular, the training accuracy value is higher than the validation accuracy value of for Model 6 and the validation loss value is higher than the training loss value, indicating that the model is not performing as well on the validation data as it is on the training data.

To sum up, having a model that does not overfit is important, as it can help ensure that the model's predictive performance remains consistent when applied to new data. However, it is also important to try to improve the accuracy of the model, and there are several techniques that can be employed to achieve this, such as increasing model complexity, using a different network architecture, or adjusting hyperparameters. It is important to balance the accuracy of the model with the risk of overfitting to ensure that the model is robust and performs well in the task.

After evaluating the predictive performance of all six models, I select Model 6 (model_cnn_agg_2) as the best model. This model was specifically designed for the aggregated data, and based on the evaluation results, it demonstrated the best predictive performance among all the models.

Task 3

Now that I have identified the best model based on the analysis of the validation data, it's time to evaluate its predictive performance using the test data.

Here, I prepare the test dataset for image classification by creating a data frame that contains the file names and corresponding class labels for each image in the test set.

```
# Create a data frame test_data which contains the file names and corresponding classes for the test datase
t.
test_data <- data.frame(file_name_agg = paste0("data_indoor_aggregation/test/", test_generator_agg$filename
s)) %>%
  mutate(class = str_extract(file_name_agg, "living_spaces|private_spaces|service_spaces|utility_spaces"))

# Display the first 10 rows of test_data
head(test_data, 10)
```

```
##                                                    file_name_agg
## 1   data_indoor_aggregation/test/living_spaces/dining_room_test1.jpg
## 2  data_indoor_aggregation/test/living_spaces/dining_room_test10.jpg
## 3  data_indoor_aggregation/test/living_spaces/dining_room_test11.jpg
## 4  data_indoor_aggregation/test/living_spaces/dining_room_test12.jpg
## 5  data_indoor_aggregation/test/living_spaces/dining_room_test13.jpg
## 6  data_indoor_aggregation/test/living_spaces/dining_room_test14.jpg
## 7  data_indoor_aggregation/test/living_spaces/dining_room_test15.jpg
## 8  data_indoor_aggregation/test/living_spaces/dining_room_test16.jpg
## 9  data_indoor_aggregation/test/living_spaces/dining_room_test17.jpg
## 10 data_indoor_aggregation/test/living_spaces/dining_room_test18.jpg
##            class
## 1  living_spaces
## 2  living_spaces
## 3  living_spaces
## 4  living_spaces
## 5  living_spaces
## 6  living_spaces
## 7  living_spaces
## 8  living_spaces
## 9  living_spaces
## 10 living_spaces
```

Below code is preparing image data for prediction. The image_prep() function converts the input images into arrays and rescales the pixel values.

```
# Function to convert image to array
image_prep <- function(x) {
  arrays <- lapply(x, function(path) {
    # Load image from file path
    img <- image_load(path, target_size = target_size,
                      grayscale = F
                      )
    # Convert image to array
    x <- image_to_array(img)
    # Reshape array to match model input shape
    x <- array_reshape(x, c(1, dim(x)))
    # Rescale image pixel values to be between 0 and 1
    x <- x/255 # rescale image pixel
  })
  # Combine arrays into a single tensor
  do.call(abind::abind, c(arrays, list(along = 1)))
}
```

The resulting arrays are then used for prediction Model 6 (model_cnn_agg_2). The predicted values are then returned as a vector using k_argmax() function.

```
# Convert test image data to array
test_x <- image_prep(test_data$file_name_agg)

# Use the CNN model to make predictions on the test image data
pred_test <- model_cnn_agg_2 %>% predict(test_x) %>% k_argmax()

# Show the first 10 predicted values
head(pred_test, 10)
```

```
## tf.Tensor([1 3 1 0 0 1 1 0 0 1], shape=(10), dtype=int64)
```

Decode function below is used to map predicted class indices to their corresponding labels. The sapply function is used to apply the decode function to each element in pred_test. This results in a character vector of predicted labels that correspond to the predicted class indices in pred_test.

```
# Convert encoding to label
decode <- function(x){
  case_when(x == 0 ~ "living_spaces",
            x == 1 ~ "private_spaces",
            x == 2 ~ "service_spaces",
            x == 3 ~ "utility_spaces",
            )
}

# Apply the decode function to the predicted test labels
pred_test <- sapply(pred_test, decode)

# SShow the first 10 predictions
head(pred_test, 10)
```

```
##  [1] "private_spaces" "utility_spaces" "private_spaces" "living_spaces"
##  [5] "living_spaces"  "private_spaces" "private_spaces" "living_spaces"
##  [9] "living_spaces"  "private_spaces"
```

Then, I use confusionMatrix function to create confusion matrix and related performance metrics to evaluate the performance of the model on the test set.

```
confusionMatrix(as.factor(pred_test),
                as.factor(test_data$class)
                )
```

```
## Confusion Matrix and Statistics
##
##                 Reference
## Prediction       living_spaces private_spaces service_spaces utility_spaces
##   living_spaces            127             30             57             21
##   private_spaces            89            151             81             52
##   service_spaces            13              5             57              9
##   utility_spaces            15              7             37            100
##
## Overall Statistics
##
##                Accuracy : 0.5112
##                  95% CI : (0.477, 0.5452)
##     No Information Rate : 0.2867
##     P-Value [Acc > NIR] : < 2.2e-16
##
##                   Kappa : 0.3521
##
##  Mcnemar's Test P-Value : < 2.2e-16
##
## Statistics by Class:
##
##                      Class: living_spaces Class: private_spaces
## Sensitivity                        0.5205                0.7824
## Specificity                        0.8221                0.6626
## Pos Pred Value                     0.5404                0.4048
## Neg Pred Value                     0.8101                0.9121
## Prevalence                         0.2867                0.2268
## Detection Rate                     0.1492                0.1774
## Detection Prevalence               0.2761                0.4383
## Balanced Accuracy                  0.6713                0.7225
##                      Class: service_spaces Class: utility_spaces
## Sensitivity                        0.24569                0.5495
## Specificity                        0.95638                0.9118
## Pos Pred Value                     0.67857                0.6289
## Neg Pred Value                     0.77184                0.8815
## Prevalence                         0.27262                0.2139
## Detection Rate                     0.06698                0.1175
## Detection Prevalence               0.09871                0.1868
## Balanced Accuracy                  0.60104                0.7306
```

According to the presented confusion matrix and statistics, the model's overall accuracy is moderate, ~50%. Furthermore, a class-wise analysis reveals that the sensitivity values for different classes ranging between 30-80%. On the other hand, the specificity values are generally higher and ranging from 70-90%.

Considering these results, further improvements are necessary to enhance the model's predictive performance.

References

RPubs - Image Classification with Convolutional Network. RPubs. https://rpubs.com/Argaadya/image_conv (https://rpubs.com/Argaadya/image_conv)