GLASGOW CALEDONIAN UNIVERSITY

Image Processing and Machine Vision

MMH623545-24-B-GLAS

UK Coins Classification using Convolution Neural Networks

word cound: 2045

by Alexandru Belea
Student ID: S1919206

Date: April 5, 2025

Contents

List of Figures

Abstract

This report presents a study on transfer learning for Convolutional Neural Networks (CNNs), where a model trained on Brazilian coins is adapted to classify UK coins by using Trensfer learning. The approach demonstrates how knowledge from one domain can be effectively transferred to another related domain with limited training data, resulting in improved model performance. The report then makes an attempt to use a larger pre-trained model, the InceptionV3 by Google, which is trained on a much larger dataset. This model is used again for transfer training with the UK coins dataset and results are discussed.

1. Introduction

Convolutional Neural Networks (CNNs) have become the cornerstone of computer vision applications, and transfer learning has emerged as a powerful technique to leverage pre-trained models for specialized tasks. This project explores the application of transfer learning using Google's InceptionV3 model for the classification of UK coins. By utilizing the pre-trained InceptionV3 architecture, we can benefit from feature extractors that have already learned to recognize patterns and shapes from millions of images, allowing us to achieve higher accuracy with our relatively small dataset of UK coins. This approach is particularly valuable when working with limited training data, as it significantly reduces the need for extensive computational resources and training time. For this implementation, we're using TensorFlow, Google's open-source machine learning framework, along with the high-level Keras API. Together, these tools provide a streamlined workflow for building, training, and evaluating our coin classification model. The project demonstrates how deep learning techniques can be effectively applied to numismatic classification tasks, potentially supporting applications in currency recognition systems, automated coin sorting, or educational tools.

2. Method

2.1 Brazilian Dataset CNN Creation and training

The method used to classify UK coins when the available training dataset contains only 309 images is to pretrain the neural network on a larger dataset. Figure 2.1 shows the complete process of transfer learning from Brazilian to UK coin classification. The flowchart in 2.1 illustrates how knowledge from one dataset (Brazilian coins) can be transferred to improve learning on a related but different dataset (UK coins) with limited samples.

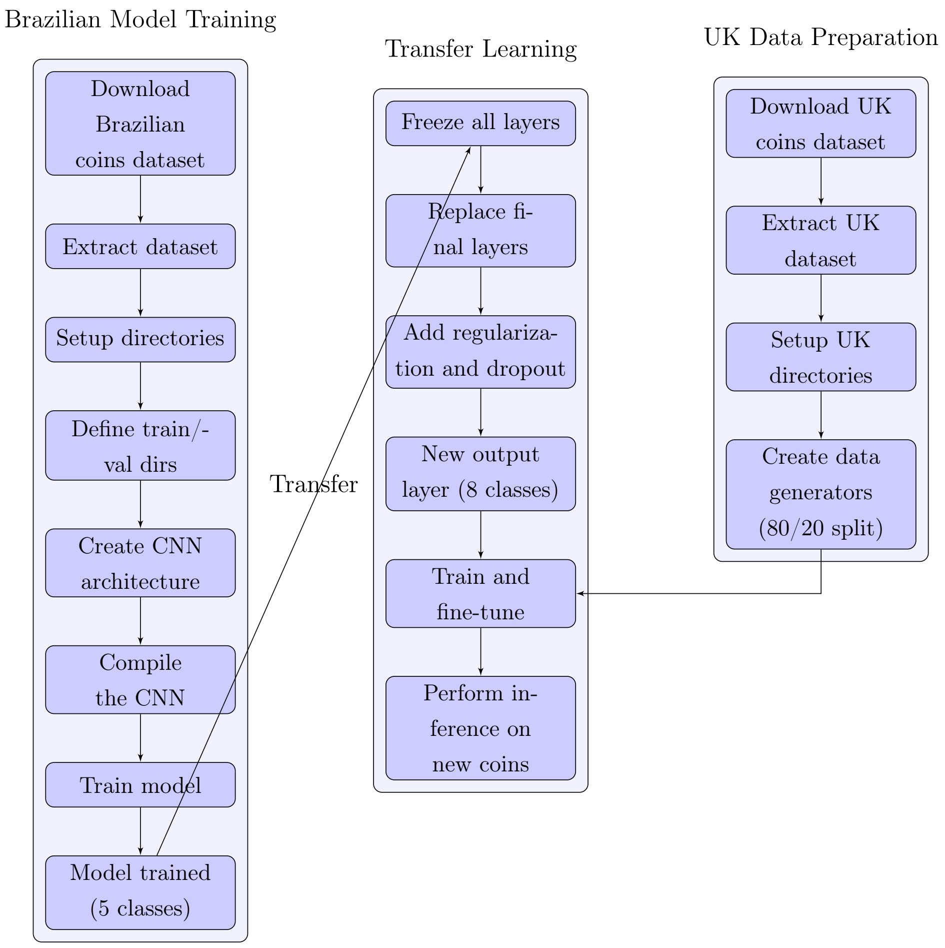


Figure 2.1: CNN Transfer Learning Flowchart: Brazilian to UK Coins

2.1.1 Creating the Model

As shown in Figure 2.2 the model trained on the Brazilian Coins is made out of 3 hidden convolutional layers, with respective pooling layers before being flattened.

```python
# Input Layer
img_input = layers.Input(shape=(100, 100, 3))
# first layer takes image in, finds features (16 filters/kernels)
x = layers.Conv2D(16, (5, 5), activation='relu')(img_input)
x = layers.MaxPooling2D((2, 2))(x)
# second layer looks at the output of first layer (32 filters)
x = layers.Conv2D(32, (3, 3), activation='relu')(x)
x = layers.MaxPooling2D((2, 2))(x)
# 3rd layer as above 64 filters
x = layers.Conv2D(64, (3, 3), activation='relu')(x)
x = layers.MaxPooling2D((2, 2))(x)

# Flatten and add output layer
x = layers.Flatten()(x)
x = layers.Dense(128, activation='relu')(x) # reduced number of neurons
    due to small network
output = layers.Dense(5, activation='softmax')(x)

# Create model:
model = Model(img_input, output)
```

Listing 2.1: Defining layers for Brazilian Dataset

The code in Figure 2.1 first creates an input layer for the NN with the expected shape of images 100 x 100 and 3 channels for RGB. The output is then fed as input to the first training layer, created using Conv2D, which creates a 2D convolutional layer with 16 feature maps with kernel size of 5 by 5 and a default stride of 1. This results in an output size as follows:

$$\text{OutputSize} = \left\lfloor \frac{\text{InputSize} - \text{KernelSize} + 1}{\text{Stride}} \right\rfloor$$

which results in an output (both Heights and Width) floored:

$$\text{OutputHeight} = \left\lfloor \frac{100 - 5 + 1}{1} \right\rfloor = \left\lfloor \frac{96}{1} \right\rfloor = 96$$

which gives 147,456 neurons:

$$TotalNeurons = OutputHeight \times OutputWidth \times NumFilters = 96 \times 96 \times 16 = 147,456$$

and results in 1216 trainable parameters:

$$Numberofparameters = (KernelHeight \times KernelWidth \times InputChannels + 1) \times Nroffilters$$

$$Numberofparameters = (5 \times 5 \times 3 + 1) \times 16 = 76 \times 16 = 1216$$

This results in a large number of parameters, which

- Hierarchy of features:
  - First layer detects simple, low-level features (edges, textures)
  - Middle layer combines these to form parts (corners, shapes)
  - Last hidden layer detects high-level, complex features (entire coin)[1]

After each hidden layer, the function max_pooling2d is called with previous layer out as an input - this downsamples the input, with the output of the first layer as follows:

$$\text{output\_shape} = \lfloor \frac{\text{input\_shape} - \text{pool\_size}}{\text{strides}} \rfloor$$

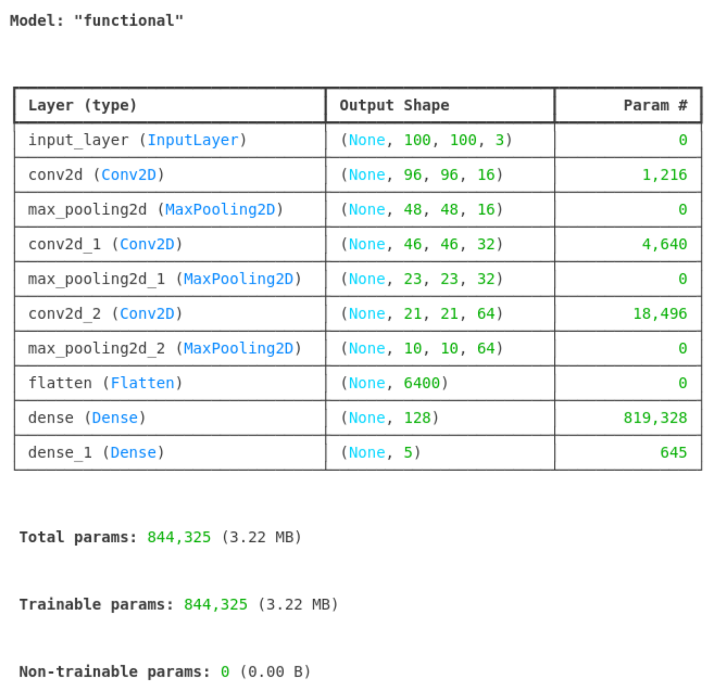
$$\text{output\_shape} = \lfloor \frac{(96, 96, 3) - (2, 2)}{(2, 2)} \rfloor = (48, 48, 16)$$

Therefore, after the first convolution layer, the output is 16 separate 2D arrays (feature maps) of size (49,49), each representing the response of one of the learned filters across the spatial dimensions of the input. This calculation was verified by looking at Figure 2.2 Layers conv2d and max_pooling2d under the Output Shape column. This is then repeated for the second hidden layer and third. After each layer the output is run through MaxPooling2D, which reduces the number of parameters and computation in the network, it reduces the spacial dimensions while retaining the most important features detected by the convolutions.

The model specifications were done using the Keras compile() method, with RMSprop optimizer and a low rate of learning of 0.001. RMSprop adapts learning rates on a per-parameter basis based on the history of gradients, but it still uses the global learning rate you specify as a starting point, more on this below[2].

```python
from tensorflow.keras.optimizers import RMSprop
# use categorical crossentropy losss
model.compile(loss='categorical_crossentropy', # Multi-class classification
              optimizer=RMSprop(learning_rate=0.001),
              metrics=['acc'])
```

Listing 2.2: Brazilian Layer specifications



Model: "functional"

Layer (type)	Output Shape	Param #
input_layer (InputLayer)	(None, 100, 100, 3)	0
conv2d (Conv2D)	(None, 96, 96, 16)	1,216
max_pooling2d (MaxPooling2D)	(None, 48, 48, 16)	0
conv2d_1 (Conv2D)	(None, 46, 46, 32)	4,640
max_pooling2d_1 (MaxPooling2D)	(None, 23, 23, 32)	0
conv2d_2 (Conv2D)	(None, 21, 21, 64)	18,496
max_pooling2d_2 (MaxPooling2D)	(None, 10, 10, 64)	0
flatten (Flatten)	(None, 6400)	0
dense (Dense)	(None, 128)	819,328
dense_1 (Dense)	(None, 5)	645

```
Total params: 844,325 (3.22 MB)

Trainable params: 844,325 (3.22 MB)

Non-trainable params: 0 (0.00 B)
```

Figure 2.2: Brazilian coins Dataset Model Summary

Finally the newly designed neural network is compiled as shown in Figure 2.2. The arguments given for compilation are loss 'categorical_crossentropy' which calculates the difference between the predicted probability distribution and the actual distribution [3].

2.1.2 Preparing the dataset

An important part of machine learning is having a correctly labeled, preferably large, dataset as well as optimally preprocessed. The Listing 2.3 shows the part of code that preprocesses the data. The Brazilian Coins dataset is already split in separate folders for training and validation. The function ImageDataGenerator() is used to set the values of the training data and validation data. For the training data, several modifications are performed to artificially increase the dataset by modifying the training data. The arguments are commented on in the Listing 2.3

```
from tensorflow.keras.preprocessing.image import ImageDataGenerator
# All images will be rescaled by 1./255 and augmented
train_datagen = ImageDataGenerator(
      rescale=1./255,            # convert to floating point 0.0-1.0
      rotation_range=40,         # rotate coins to train NN to detect
    coins at angles
      width_shift_range=0.2,     # This helps the model learn to
    recognize coins even if they are not perfectly centered in the image.
      height_shift_range=0.2,    # As above but vertically
      zoom_range=0.2,
      fill_mode='nearest')      # fill empty pixels created after the shifts above with a copy of nearest pixel
val_datagen = ImageDataGenerator(rescale=1./255) #do not augment validation
B = 15 #Batch size
# Flow training images in batches of B using train_datagen generator
train_generator = train_datagen.flow_from_directory(
        train_dir,  # This is the source directory for training images
        target_size = (100, 100),  # All images will be resized to 100x100
        batch_size=B,
        # To use categorical_crossentropy loss, we need categorical labels
        class_mode='categorical')
# Flow validation images in batches of 20 using val_datagen generator
validation_generator = val_datagen.flow_from_directory(
        validation_dir,
        target_size=(100, 100),
        batch_size=B,
        class_mode='categorical')
```

Listing 2.3: Preprocessing the Dataset

The validation data should not be augmented, as it is the ground truth on which to test the NN with the newly trained parameters, therefore only the rescale argument is called. The data is then loaded by calling the flow_from_directory method with target size of 100 by 100 pixels which will ensure all loaded images have the size expected by the input layer. Similar steps are taken when loading the UK dataset later, with the exception of using the argument validation_split=0.2 which splits the dataset intio 80% training and 20% validation, as it was not pre-split. A flowchart of the data augmentation step can be seen below in Figure 2.3.

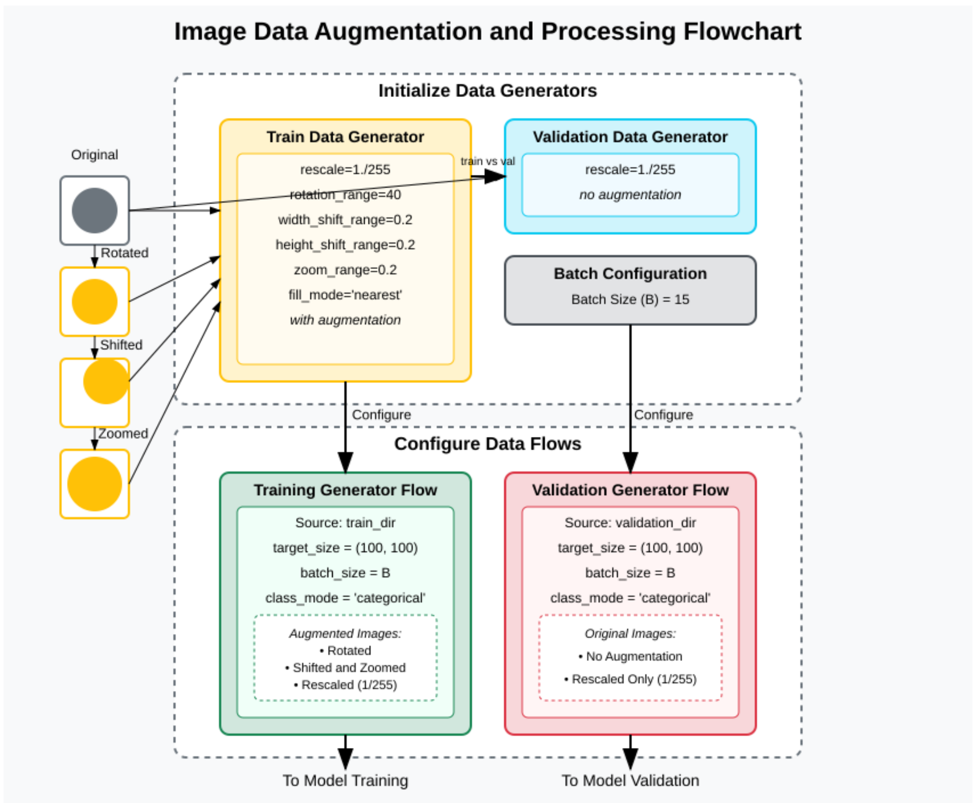


Figure 2.3: Data Augmentation and Preprocessing

2.1.3 Model Training

Training is initiated as per Listing 2.5 below, with a flowchart depicting the algorithm available in Figure 2.4. Two methods were used to improve training: early_stopping, which monitors val_acc during training and stops the training if it does not improve, with the added option of restoring the parameters to the best Epoch. As well as reduce_lr, which reduces the learning rate by half when it detects the training is not improving, this is done per Epoch as opposed to RMSprop which slows down training per parameter, while maintaining the overall training speed set at compilation[2].

2.2 Transfer Learning with smaller UK Coins Dataset

For the Transfer Learning to the UK dataset, the original layers were first frozen and a new model using the Dense function was produced. Figure 2.7 shows the new model now has an extra layer. The Dense function creates a fully connected layer as seen in Figure 2.9 that branches out from the flattened layer - in effect replacing the final two layers: one dense layer that is trained on the features of the Brazilian coins and the final output layer. A dropout layer is used as a regularization technique to prevent overfitting, this is necessary due to the small UK dataset. By randomly freezing parameters it makes the training less reliant on any one feature and able to generalize [4].

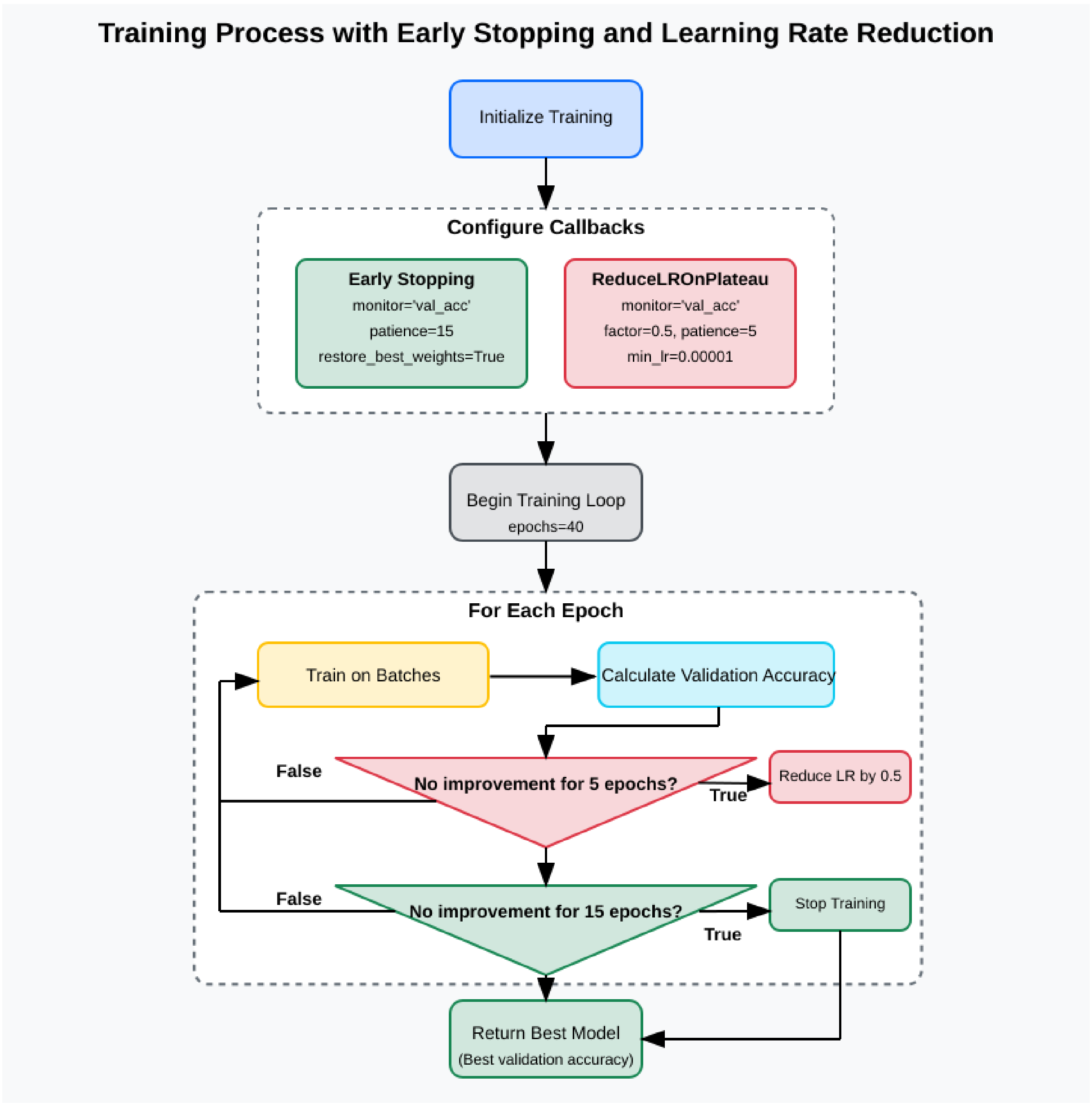


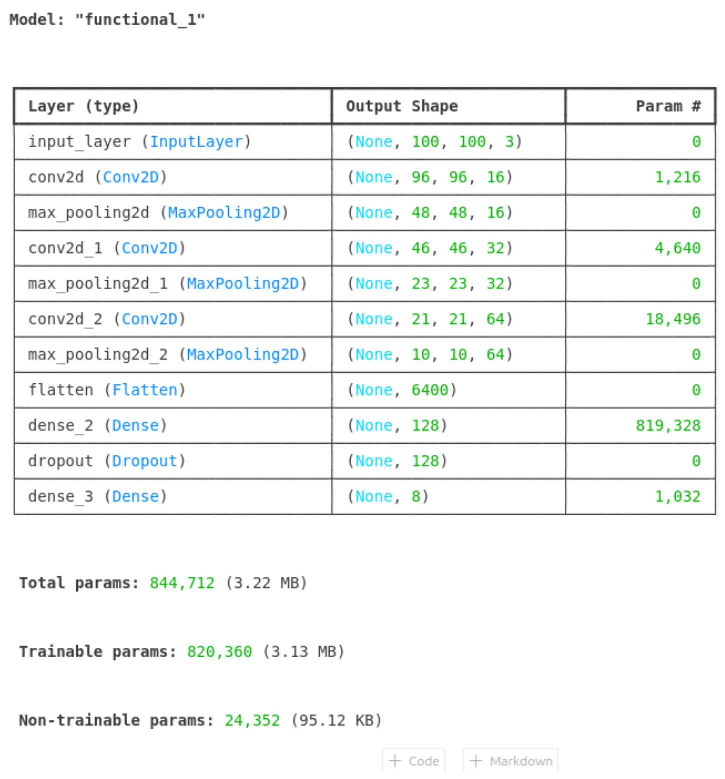
Figure 2.4: Learning Algorithm, values for Brazilian dataset NN

```
replace_point = model.get_layer('flatten')
x = replace_point.output
x = layers.Dense(128, activation='relu', kernel_regularizer=regularizers
    .l2(0.0005))(x)
x = layers.Dropout(0.4)(x)                    #  prevent overfitting with
    a dropout
UKoutput = layers.Dense(8, activation='softmax')(x)
UKmodel = Model(img_input, UKoutput)
```

Figure 2.6: Transfer Training CNN design

```python
from tensorflow.keras.callbacks import EarlyStopping

early_stopping = EarlyStopping(
    monitor='val_loss',  # 'val_loss' to restore best epoch parameters
    patience=18,  # number of epochs with no improvement before stopping
    restore_best_weights=True  # Restore to parameters providing best val_acc
)

# Learning rate reduction when improvement plateaus
reduce_lr = ReduceLROnPlateau(
    monitor='val_acc',  # monitor validation accuracy
    factor=0.5,         # multiply learning rate by 0.5 when triggered
    patience=8,         # wait 5 epochs with no improvement before reducing
    min_lr=0.00001,     # don't reduce learning rate below this value
    verbose=1           # print message when reducing learning rate
)

history = model.fit(
      train_generator,
      steps_per_epoch=765//B,  # 765 images = batch_size * steps
      epochs=40,
      validation_data=validation_generator,
      callbacks=[early_stopping, reduce_lr], # used for restoring best epoch parameters.
      validation_steps=300//B,  # 300 images = batch_size * steps
      verbose=2)
```

Figure 2.5: Brazilian CNN Training



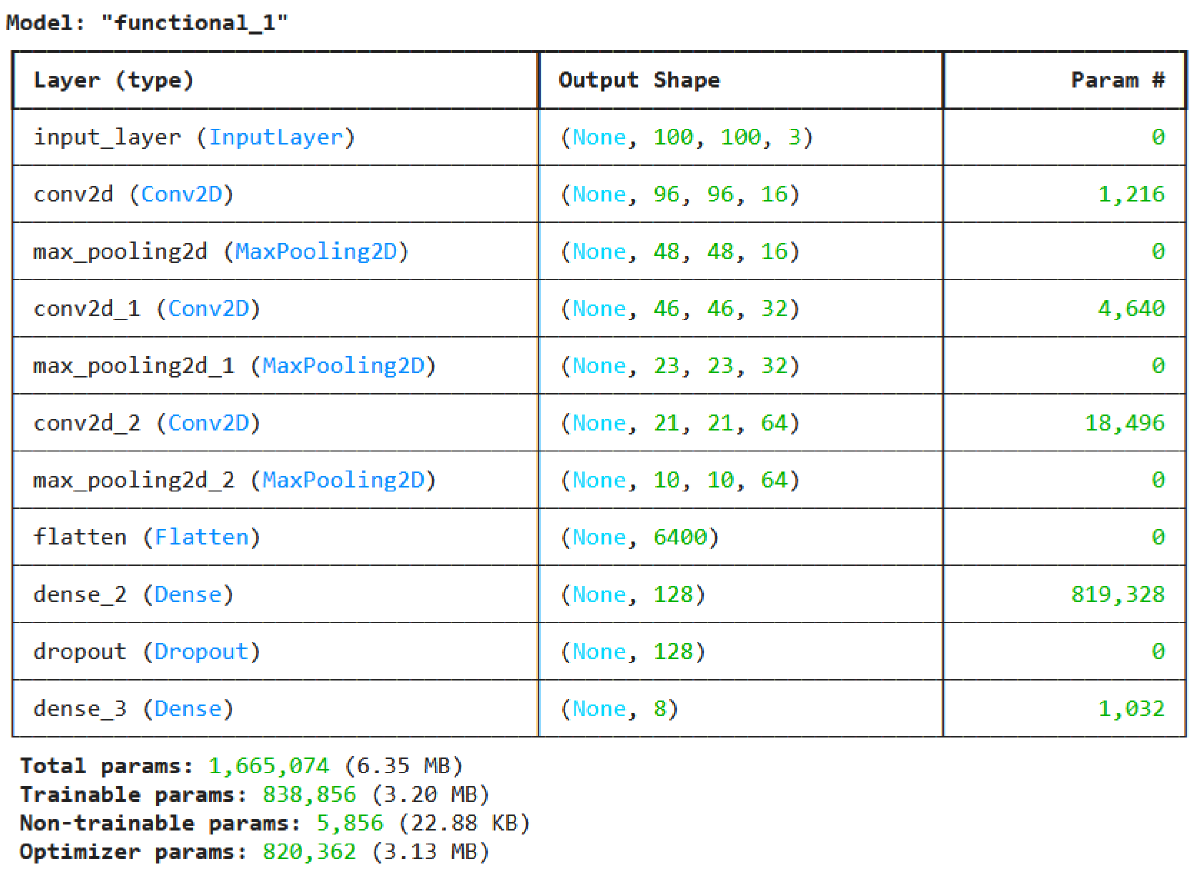
```
Model: "functional_1"

┏━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━┳━━━━━━━━━━━━━━━━━━━━━━━━┳━━━━━━━━━━━━━━━┓
┃ Layer (type)                    ┃ Output Shape           ┃       Param # ┃
┡━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━╇━━━━━━━━━━━━━━━━━━━━━━━━╇━━━━━━━━━━━━━━━┩
│ input_layer (InputLayer)        │ (None, 100, 100, 3)    │             0 │
├─────────────────────────────────┼────────────────────────┼───────────────┤
│ conv2d (Conv2D)                 │ (None, 96, 96, 16)     │         1,216 │
├─────────────────────────────────┼────────────────────────┼───────────────┤
│ max_pooling2d (MaxPooling2D)    │ (None, 48, 48, 16)     │             0 │
├─────────────────────────────────┼────────────────────────┼───────────────┤
│ conv2d_1 (Conv2D)               │ (None, 46, 46, 32)     │         4,640 │
├─────────────────────────────────┼────────────────────────┼───────────────┤
│ max_pooling2d_1 (MaxPooling2D)  │ (None, 23, 23, 32)     │             0 │
├─────────────────────────────────┼────────────────────────┼───────────────┤
│ conv2d_2 (Conv2D)               │ (None, 21, 21, 64)     │        18,496 │
├─────────────────────────────────┼────────────────────────┼───────────────┤
│ max_pooling2d_2 (MaxPooling2D)  │ (None, 10, 10, 64)     │             0 │
├─────────────────────────────────┼────────────────────────┼───────────────┤
│ flatten (Flatten)               │ (None, 6400)           │             0 │
├─────────────────────────────────┼────────────────────────┼───────────────┤
│ dense_2 (Dense)                 │ (None, 128)            │       819,328 │
├─────────────────────────────────┼────────────────────────┼───────────────┤
│ dropout (Dropout)               │ (None, 128)            │             0 │
├─────────────────────────────────┼────────────────────────┼───────────────┤
│ dense_3 (Dense)                 │ (None, 8)              │         1,032 │
└─────────────────────────────────┴────────────────────────┴───────────────┘

 Total params: 844,712 (3.22 MB)

 Trainable params: 820,360 (3.13 MB)

 Non-trainable params: 24,352 (95.12 KB)
```

Figure 2.7: UK model Summary - all Brazilian layers frozen

Finally an output fully connected layer is created with 8 units as the new dataset has 8 classes. This can be seen in the shape of the UK model summary in 2.7. The model is then trained again using similar training method as seen in 2.5, however with 80 epochs and early_stopping patience of 40.

2.2.1 Unfreezing one Brazilian layer

To improve the results of the Transfer Training and reduce overfitting, the last convolutional layer of the original model was unfrozen. This allows training backpropagation to modify the parameters in the unfrozen layer of the original model. The new Model summary in Figure 2.8 As can be seen, the model now has 838,856 trainable parameters as opposed to 820,360 in Figure 2.7. This change should allow the new model to not overfit and be better at validating new datasets (unseen during training).



```
Model: "functional_1"
┏━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━┳━━━━━━━━━━━━━━━━━━━━━━━━┳━━━━━━━━━━━━━━━┓
┃ Layer (type)                    ┃ Output Shape           ┃       Param # ┃
┡━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━╇━━━━━━━━━━━━━━━━━━━━━━━━╇━━━━━━━━━━━━━━━┩
│ input_layer (InputLayer)        │ (None, 100, 100, 3)    │             0 │
├─────────────────────────────────┼────────────────────────┼───────────────┤
│ conv2d (Conv2D)                 │ (None, 96, 96, 16)     │         1,216 │
├─────────────────────────────────┼────────────────────────┼───────────────┤
│ max_pooling2d (MaxPooling2D)    │ (None, 48, 48, 16)     │             0 │
├─────────────────────────────────┼────────────────────────┼───────────────┤
│ conv2d_1 (Conv2D)               │ (None, 46, 46, 32)     │         4,640 │
├─────────────────────────────────┼────────────────────────┼───────────────┤
│ max_pooling2d_1 (MaxPooling2D)  │ (None, 23, 23, 32)     │             0 │
├─────────────────────────────────┼────────────────────────┼───────────────┤
│ conv2d_2 (Conv2D)               │ (None, 21, 21, 64)     │        18,496 │
├─────────────────────────────────┼────────────────────────┼───────────────┤
│ max_pooling2d_2 (MaxPooling2D)  │ (None, 10, 10, 64)     │             0 │
├─────────────────────────────────┼────────────────────────┼───────────────┤
│ flatten (Flatten)               │ (None, 6400)           │             0 │
├─────────────────────────────────┼────────────────────────┼───────────────┤
│ dense_2 (Dense)                 │ (None, 128)            │       819,328 │
├─────────────────────────────────┼────────────────────────┼───────────────┤
│ dropout (Dropout)               │ (None, 128)            │             0 │
├─────────────────────────────────┼────────────────────────┼───────────────┤
│ dense_3 (Dense)                 │ (None, 8)              │         1,032 │
└─────────────────────────────────┴────────────────────────┴───────────────┘
 Total params: 1,665,074 (6.35 MB)
 Trainable params: 838,856 (3.20 MB)
 Non-trainable params: 5,856 (22.88 KB)
 Optimizer params: 820,362 (3.13 MB)
```

Figure 2.8: UK model Summary - one Brazilian layers unfrozen

2.3 Transfer Training from pretrained Google InceptionV3 model

An attempt was made to use the InceptionV3 Model which is made available by Google[5], an open source model pre-trained on the ImageNet library spanning hundreds of thousands of images[6]. The model contains multiple convolutional blocks and mixed layers. The model is forked at layer "mixed7" which gives a good balance between keeping the features of the original model while allowing training on new data. The custom classification for our case was created by first flattening the last layer of the InceptionV3 model and a fully connected layer added that should capture the features of the small UK coins dataset. A relatively high droprate had to be used of 0.8 to ensure the new model does not overfit on such a small dataset. The output layer is similar to the previous model from the Brazilian dataset.

As previously the model used Categorical Cross-Entropy but label smoothing of 0.1 was added to help with the overfitting issue, which improves generalization by introducing a small amount of uncertainty into labels [7].

The UK dataset was imported converting the images to a resolution similar to the

Table 2.1: Summary of InceptionV3 Transfer Learning Model for UK Coin Classification

Parameter	**Description**
Base Architecture	InceptionV3 (pre-trained, frozen layers)
Input Shape	$(150, 150, 3)$ RGB images
Output Shape	$(None, 8)$ UK coin denominations
Feature Extraction	Until 'mixed7' layer with output shape $(7, 7, 768)$
Custom Classification Head	• Flatten: $(7, 7, 768) \rightarrow (37632)$ • Dense: 256 neurons, ReLU, L2 regularization (0.001) • Dropout: rate 0.8 • Output: 8 neurons, softmax activation
Training Configuration	• Loss: Categorical Cross-Entropy with label smoothing (0.1) • Optimizer: RMSprop (lr: 0.0001) • Metric: Accuracy
Total Parameters	18,611,368 (∼71.00 MB)
Trainable Parameters	9,636,104 (∼36.76 MB)
Non-trainable Parameters	8,975,264 (∼34.24 MB)

```python
from tensorflow.keras.optimizers import RMSprop
from tensorflow.keras import regularizers
from keras.losses import CategoricalCrossentropy

# Flatten the output layer to 1 dimension
x = layers.Flatten()(last_output)
# Add a fully connected layer with 1,024 hidden units and ReLU
    activation
x = layers.Dense(256, activation='relu', #256
                 kernel_regularizer=regularizers.l2(0.001))(x)  # Added
    L2 regularization

# Dropout rate
x = layers.Dropout(0.8)(x)
# Final Relu function for non-binary classification
x = layers.Dense(8, activation='softmax')(x)

# Configure and compile
model = Model(pre_trained_model.input, x)
model.compile(loss=CategoricalCrossentropy(label_smoothing=0.1),   # '
    categorical_crossentropy',
              optimizer=RMSprop(learning_rate=0.0001),
              metrics=['acc'])
```

Figure 2.9: Custom Model from InceptionV3

model chosen input shape of (150,150) - this was chosen as an average between the dataset of closer to 100 by 100 and the InceptionV3 original training of three or four hundred pixels. The idea was to give the highest chance for the model to pick up similar features. The model summary was too long but it is summarized in Table 2.1. The training then was initialized in the same way as in Figure 2.5. While a larger value of epochs was chosen, 160,and a higher patience for EarlyStopping, the model training stopped at around 80 epochs each time.

3. Testing and Results

3.1 Results of Training with Brazilian dataset

The training results for the Brazilian dataset were relatively good for a dataset of under one thousand images. The val_acc output - which is the validation accuracy when testing the network on the validation dataset, plateaued somewhere around 0.93.

```
>Epoch 38/40
>51/51 - 2s - 37ms/step - acc: 0.9307 - loss: 0.2271 - val_acc: 0.9367 -
    val_loss: 0.2074 - learning_rate: 6.2500e-05
>Epoch 39/40
>
>Epoch 39: ReduceLROnPlateau reducing learning rate to 3.125000148429535
   e-05.
>51/51 - 2s - 37ms/step - acc: 0.9281 - loss: 0.2234 - val_acc: 0.9367 -
    val_loss: 0.2125 - learning_rate: 6.2500e-05
>Epoch 40/40
>51/51 - 2s - 37ms/step - acc: 0.9386 - loss: 0.1968 - val_acc: 0.9467 -
    val_loss: 0.2073 - learning_rate: 3.1250e-05
```

Figure 3.1: Brazilian CNN Training result

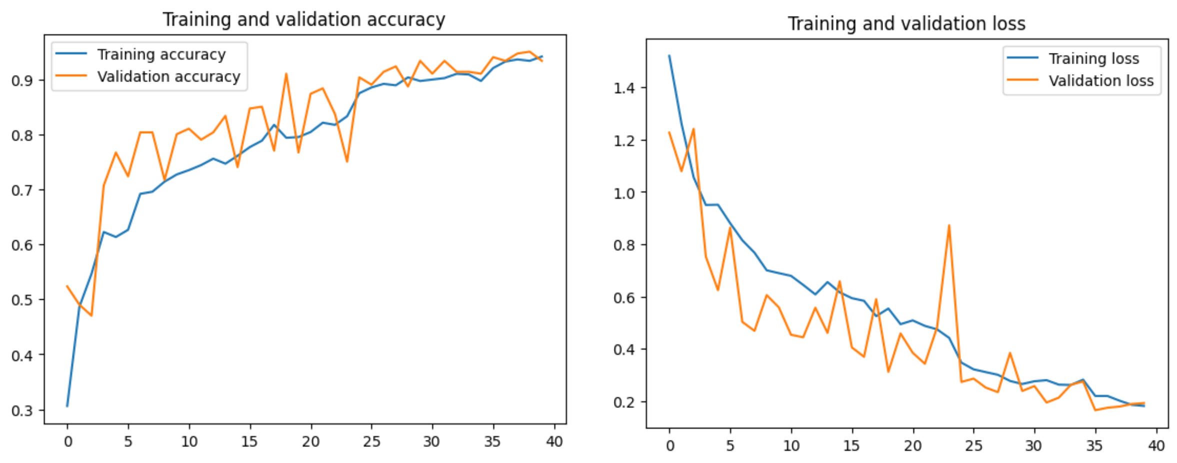


Figure 3.2: Brazil Model Accuracy Chart

Figure 3.2 shows that the Training Accuracy of the Brazilian dataset is just above 90% with validation accuracy starting to creep just towards the end below Training accuracy, this can signify overfitting. However, the restore_best_weights was set to True which means the best parameters will be restored upon termination of training.

3.2 Results of Transfer Training with UK dataset

3.2.1 Training with all Brazilian Layers frozen

The model in Figure 2.7 was used to do Transfer Training on the smaller UK dataset. The results can be seen in Figure 3.3 and 3.4. Due to maintaining all previous convolutional layers frozen, the results show the new model suffers from overfitting, with validation loss higher than training loss. This indicates that unfreezing some of the deeper convolutional layers and fine-tuning them with a very low learning rate might improve performance by allowing the feature extractors to adapt to the specific characteristics of UK coins.

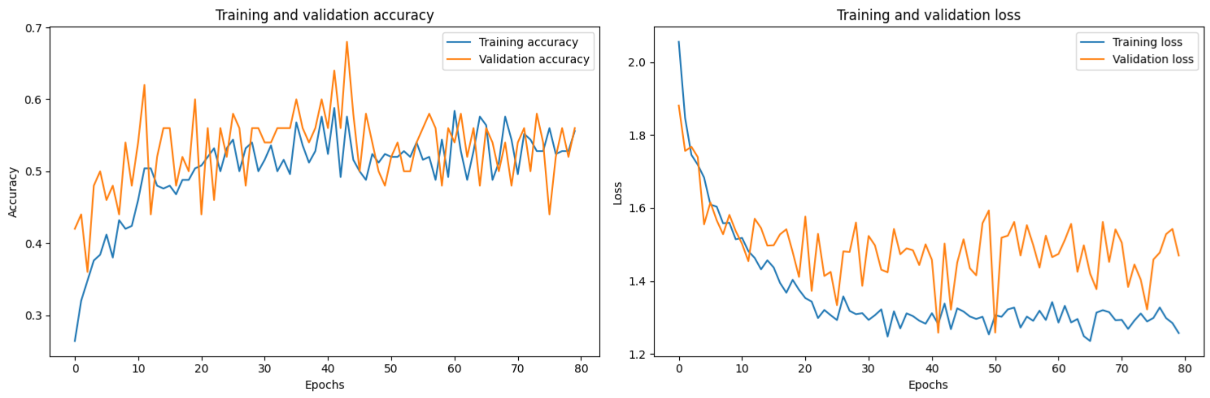


Figure 3.3: Accuracy Results - UK Model with all Brazilian layers frozen

```
25/25 - 1s - 39ms/step - acc: 0.5240 - loss: 1.3277 - val_acc: 0.5200 -
    val_loss: 1.4781 - learning_rate: 1.0000e-07
Epoch 78/80
25/25 - 1s - 50ms/step - acc: 0.5280 - loss: 1.2986 - val_acc: 0.5600 -
    val_loss: 1.5286 - learning_rate: 1.0000e-07
Epoch 79/80
25/25 - 1s - 39ms/step - acc: 0.5280 - loss: 1.2851 - val_acc: 0.5200 -
    val_loss: 1.5428 - learning_rate: 1.0000e-07
Epoch 80/80
25/25 - 1s - 49ms/step - acc: 0.5560 - loss: 1.2575 - val_acc: 0.5600 -
    val_loss: 1.4700 - learning_rate: 1.0000e-07
```

Figure 3.4: UK Transfer Training result - all Brazilian Layers frozen

3.2.2 Unfreezing the last Convolutional Brazilian layer

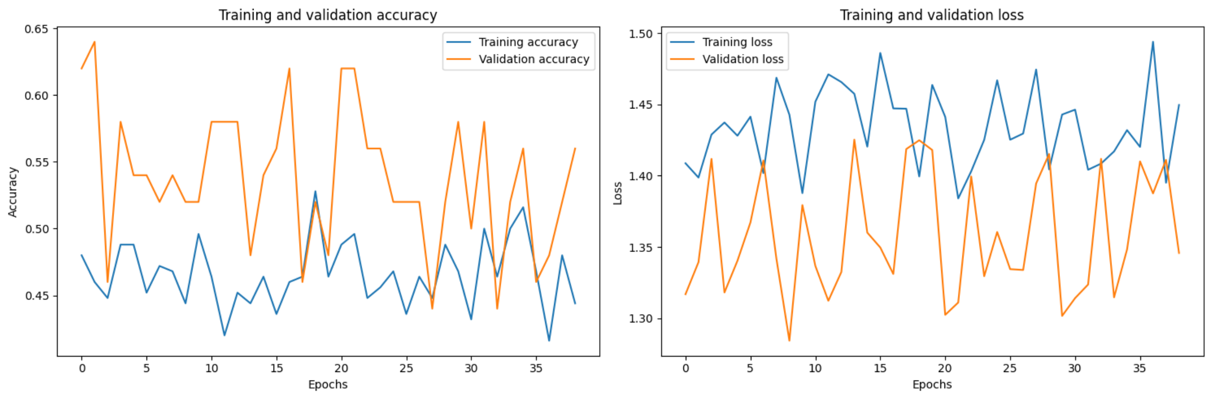


Figure 3.5: Accuracy Results - UK Model with one Brazilian layer unfrozen

The results of unfreezing the last convolutional layer of the Brazilian layer did not bring much improvement as Figure 3.5 shows. The training was stopped early and values were restored to best case of 0.64 Validation accuracy and 1.28 Validation loss.

```
>Epoch 37/80
>25/25 - 1s - 49ms/step - acc: 0.4160 - loss: 1.4941 - val_acc: 0.4800 -
    val_loss: 1.3876 - learning_rate: 1.2500e-06
>Epoch 38/80
>25/25 - 1s - 39ms/step - acc: 0.4800 - loss: 1.3951 - val_acc: 0.5200 -
    val_loss: 1.4112 - learning_rate: 1.2500e-06
>Epoch 39/80
>25/25 - 1s - 39ms/step - acc: 0.4440 - loss: 1.4497 - val_acc: 0.5600 -
    val_loss: 1.3457 - learning_rate: 1.2500e-06
```

Figure 3.6: UK Transfer Training result - last Brazilian Layer unfrozen

3.3 Results of Transfer Training from Google InceptionV3 model

The results when using a pre-trained model on the large dataset from Imagenet are shown in Figure 3.7.

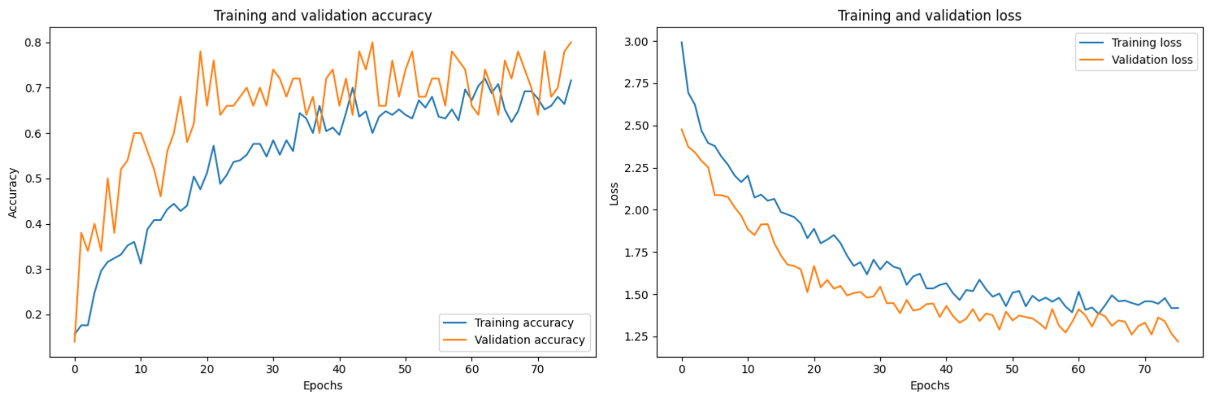


Figure 3.7: Accuracy Results - Transfer Training from InceptionV3

```
>25/25 - 1s - 57ms/step - acc: 0.3680 - loss: 2.1907 - val_acc: 0.5800 -
    val_loss: 1.8911 - learning_rate: 1.0000e-04
>Epoch 12/160
>25/25 - 1s - 50ms/step - acc: 0.4120 - loss: 2.1293 - val_acc: 0.4800 -
    val_loss: 1.9015 - learning_rate: 1.0000e-04
>Epoch 13/160
>25/25 - 1s - 50ms/step - acc: 0.4240 - loss: 2.1202 - val_acc: 0.4400 -
    val_loss: 2.0088 - learning_rate: 1.0000e-04
>...
>Epoch 137/160

>Epoch 137: ReduceLROnPlateau reducing learning rate to
   3.0517577354061 13e-09.
>25/25 - 1s - 50ms/step - acc: 0.7480 - loss: 1.2675 - val_acc: 0.8400 -
    val_loss: 1.1804 - learning_rate: 6.1035e-09
```

Figure 3.8: Training rezult Transfer from InceptionV3

The model did not overfit as the validation accuracy stayed above the training accuracy, as well as the validation loos being below training loss. When comparing the results in the UK model transfered from the Brazilian model - trained with under 1000 images, the power of pre-trained models on huge datasets is proven. However the accuracy did not go above 0.80 which means that while better than the previous Brazilian model, the Inception-based model will still struggle to get good inference results. To show this, inference was attempted using several images of coins, with better and lower level of detail that were not from the training dataset. Figure 3.9 shows the results of inference, which show that one pound coins are detected as two pound coins with quite high confidence -

in reality the diameter of these two coins differs, but the image dataset does not provide a size difference - with all coins taking over most of the image. This means the model does not have size information, and the £1 and £2 coins look otherwise quite similar with two gradients. One of the five pence coins is correctly identified, but the second 5p_2.png is incorrectly identified. The images 100_easy_*.jpg were added as a test: these images were in the training data and are correctly identified.

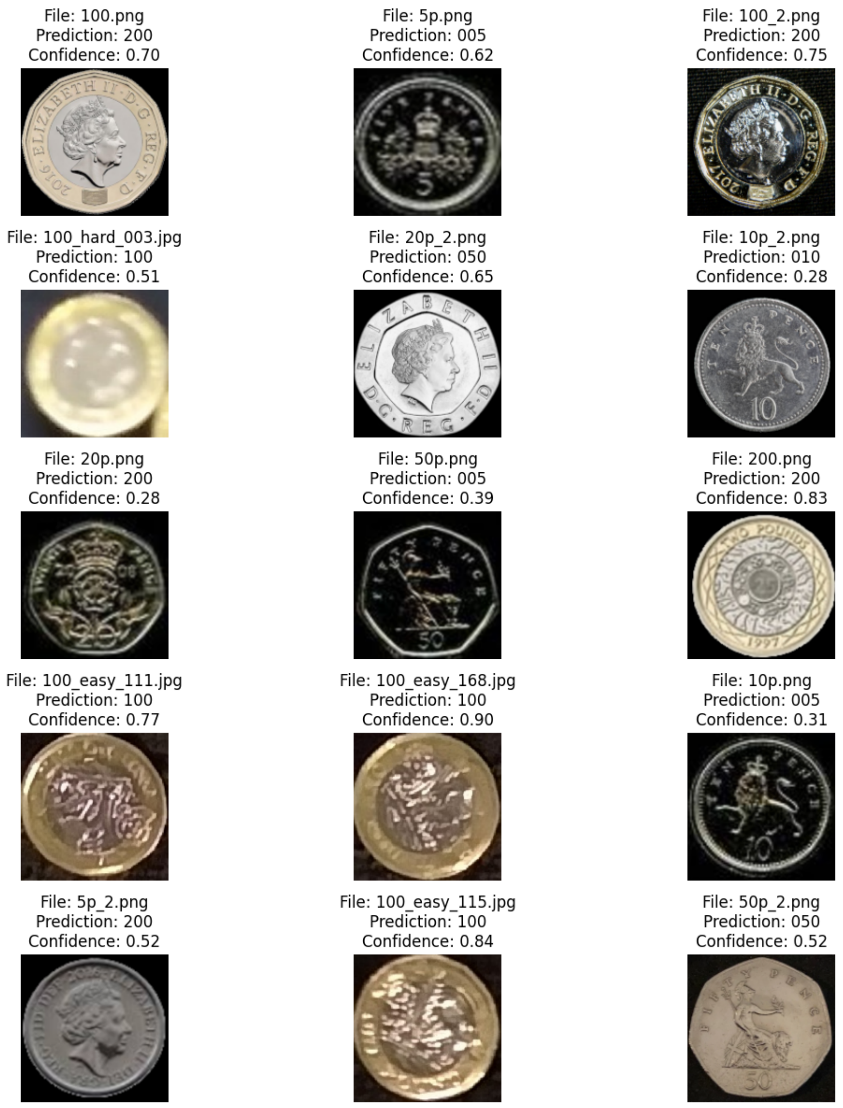


Figure 3.9: Inference results with model transfer trained from InceptionV3

4. Conclusions

The paper made an attempt at creating and training a couple of Convolution Neural Networks. Both training from scratch using small datasets of 768 images of coins from Brazil and 310 UK coins, as well as making use of a pre-existing network and weights trained by Google on the ImageNet large dataset of hundreds of thousands of coins. The paper showed that transfer training is a good method of making use of existing models instead of starting from scratch when dealing with small datasets. However, the results show that in both cases the UK dataset was just too small to make a valuable and trustworthy detection, as well as possible improvements in optimization that were not explored which might have helped. For example gradually unfreezing more layers of the Inception V3 model allowing parameters to be changed with very small learning rates might help, as well as using other existing models that might help.

Bibliography

[1] (2023, 3) Basic CNN architecture: 5 layers explained simply. Upgrad.com. [Online]. Available: https://www.upgrad.com/blog/basic-cnn-architecture/

[2] G. Hinton, N. Srivastava, and K. Swersky, "Neural networks for machine learning lecture 6a: Overview of mini-batch gradient descent," University of Toronto, Tech. Rep., 10 2012. [Online]. Available: https://www.cs.toronto.edu/~tijmen/csc321/slides/lecture_slides_lec6.pdf

[3] TensorFlow Team. (2024, 6) tf.keras.losses.categorical_crossentropy. [Online]. Available: https://www.tensorflow.org/api_docs/python/tf/keras/losses/categorical_crossentropy

[4] Keras Team. (2024) Keras documentation: Dropout layer. [Online]. Available: https://keras.io/api/layers/regularization_layers/dropout/

[5] ——. (2024) Keras documentation: InceptionV3. [Online]. Available: https://keras.io/api/applications/inceptionv3/

[6] (2021, 3) ImageNet. ImageNet. [Online]. Available: https://www.image-net.org/

[7] A. Rosebrock. (2019, 12) Label smoothing with Keras, TensorFlow, and deep learning. [Online]. Available: https://pyimagesearch.com/2019/12/30/label-smoothing-with-keras-tensorflow-and-deep-learning/