CNN – DENSENET201

Lego Minifigure Classification

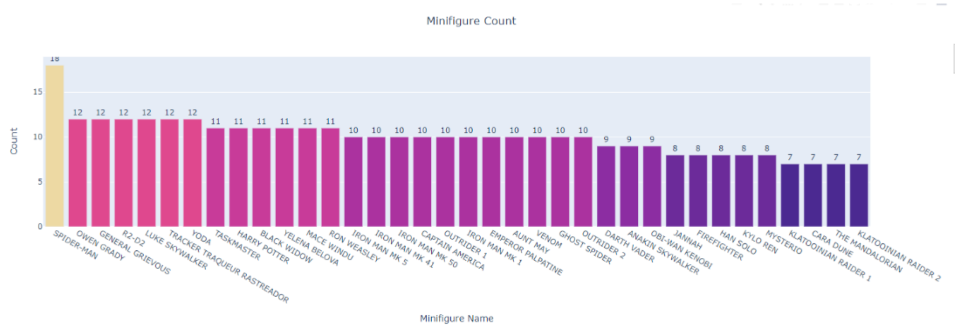
Arnas Juravicius - 18257305
Oisin McNamara - 18237398

The data set

The data set we used contained 360 images. Our CNN aims to recognize the character of the Lego minifigure provided in each image. As this dataset is small additional data augmentation was needed to increase the dataset. The dataset contains 5 folders, Harry Potter, Jurassic World, Star Wars, Marvel, and a test set. The CNN architecture we implemented was DenseNet201.

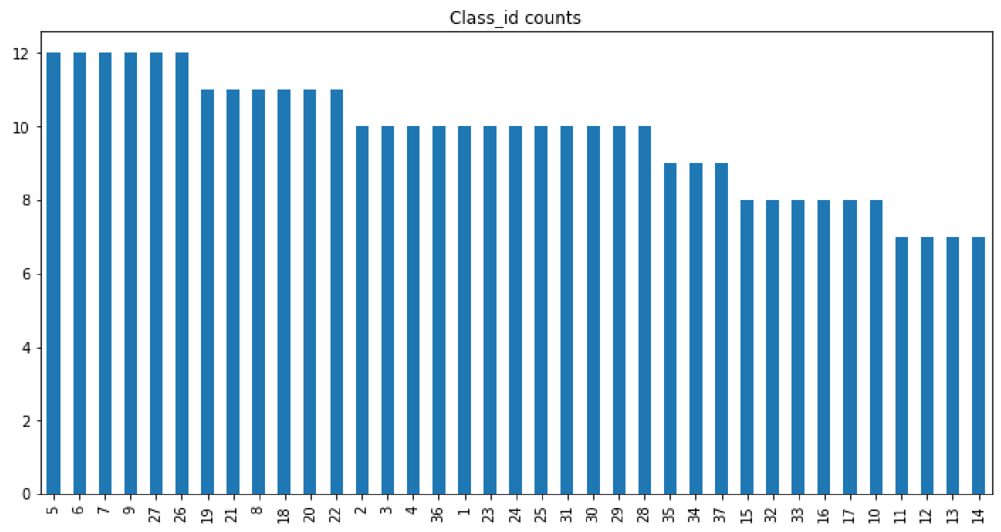
Visualizations

The first visualization was visualizing the number of minifigure names and how many times they appear.

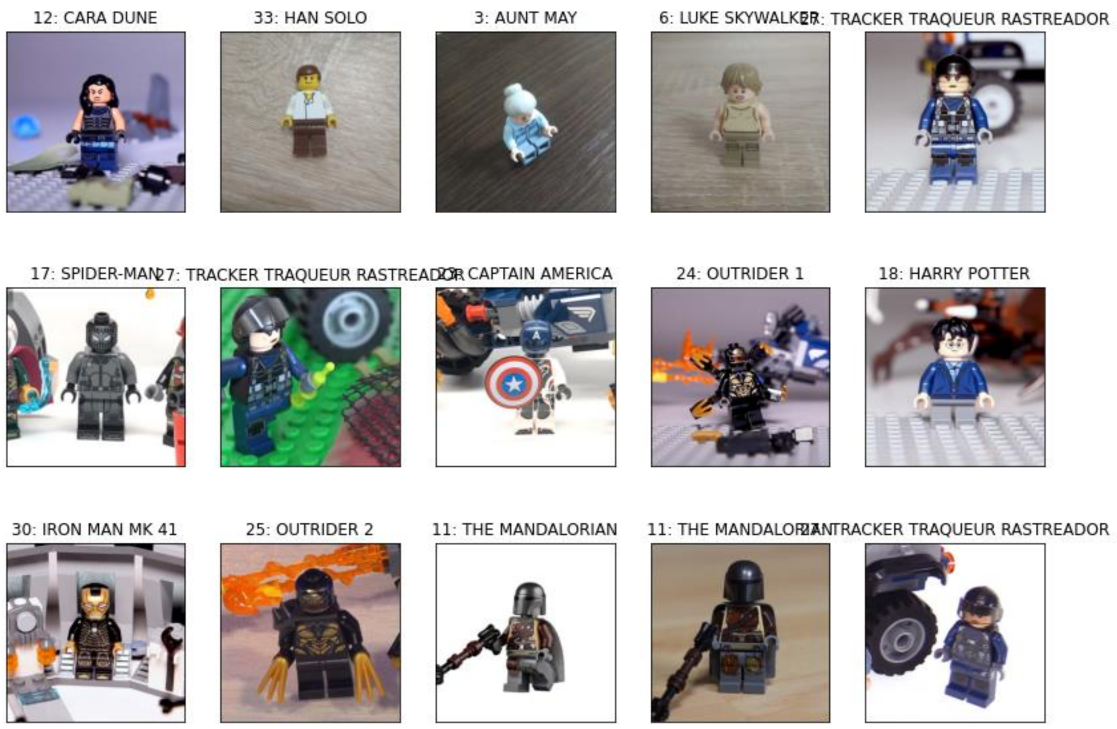


This conveyed that Spider Man minifigures appeared much more than the like of Cara Dune at 18 times compared to 7.

The next thing was to visualize how many different class ids there were and how many times they appeared. This information was found in the index.csv



We also sampled a random 15 images from the dataset. This gives us an idea of what the data looked like.



Preprocessing

We had to sort through our data and optimize it for our training model during the data preprocessing stage. The total size of the data is 361. So first, we split our data into train and test sets, the train has 324 images, and the test has 37. This is known as a two-fold validation split.

Next, we converted the pixels of image data to an array. Following this, we read in the images using "cv2" imread function. However, this function reads in the image as BGR format (blue, green, and red). This may cause issues for our model as some images may look distorted as the original format is RGB. So, following this, we used the "cv2" function COLOR_BGR2RGB. This converted the images back to RGB, sorting the issue.

Following this the images needed to be rescaled to fit the model. We fit our images to be 128x128 pixels, so the images are smaller in pixel size. This makes the resolution worse, but the processing time is faster. This was something we had to incorporate as our model took a long time to train the data. The next step is to normalize the pixel range between 0-1 to speed up learning time and accuracy.

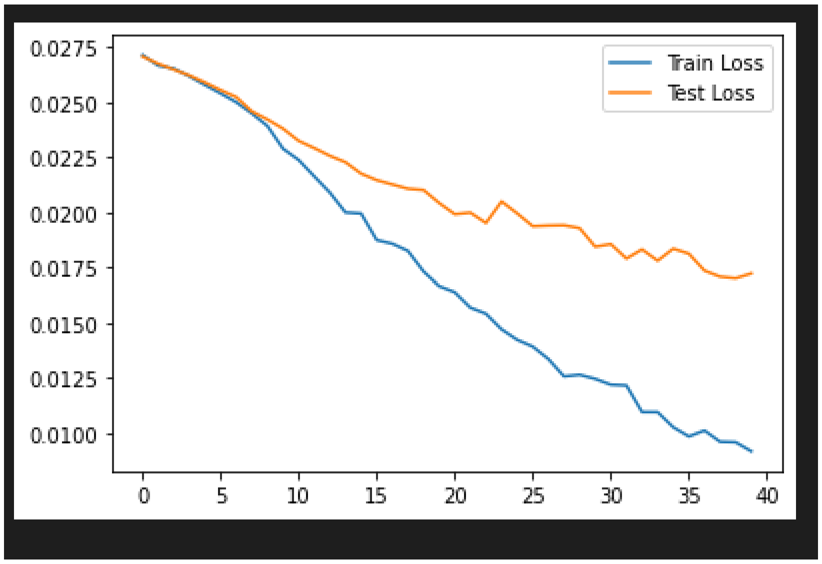
The network structure

As we mentioned above, we chose the DenseNet architecture for our project. Each layer in DenseNet receives extra input from all preceding levels and sends its feature maps to all future layers. The term "concatenation" is used. Each layer gets "collective knowledge" from the levels above it. Since each one of these layers receives feature maps from all layers after it, the network can be more compact. This allows DenseNet to have higher computational and memory efficiency.

DenseNet also utilizes a concept called residual learning. Residual learning is a type of neural network also known as ResNet. Residual networks skip connections and jump over some layers. This builds a construct known as a pyramid. Typically, ResNet models are implemented with double or triple layer skips that contain nonlinearities. However, DenseNet has the capability of several parallel skips. Due to this feature, DenseNet is more efficient in terms of computation and accuracy than typical ResNet networks.

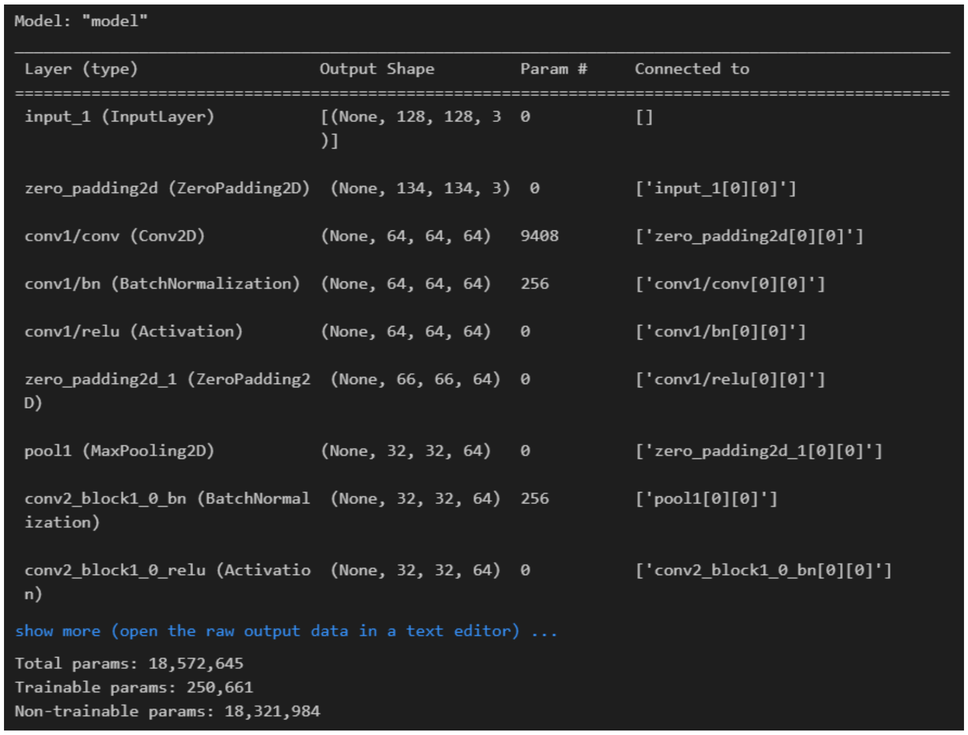
For weight initialization, we used ImageNet. Resnet is a model pre-trained on ImageNet using ImageNet's weight initialization. Batch normalization was applied to our pre-trained denseNet201 model. Batch normalization on various layers maintains that the mean output is close to 0, and the output standard deviation is close to 1. This allowed our layers in the model to learn more independently of one another.

We utilized the MSE (Mean squared error) loss function, which reduced our loss number. Mean squared error is calculated as the average of the squared differences between the predicted and actual values. This gives the result back as always being positive regardless of the sign of the predicted and actual values. Squaring means that the bigger mistakes result in more errors than the smaller ones. This punishes the model for making bigger mistakes rather than focusing too much on the small ones



For regularization, we only used pooling. We used the average pooling technique, and it takes the average value of size m*m. It uses the average value of the features.

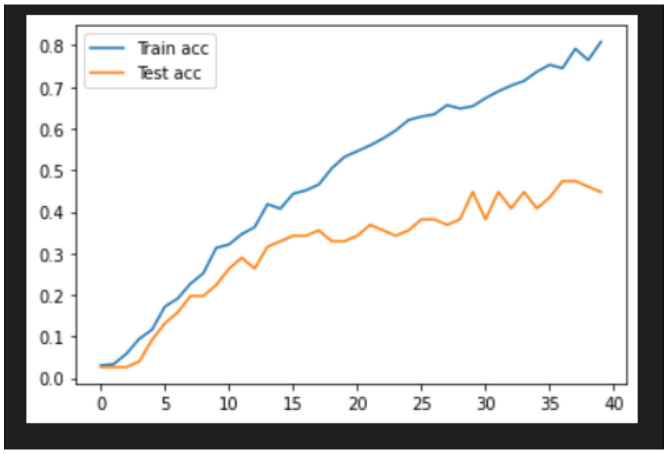
Below is a summary of our model. Things like pooling and batch normalization can be seen.

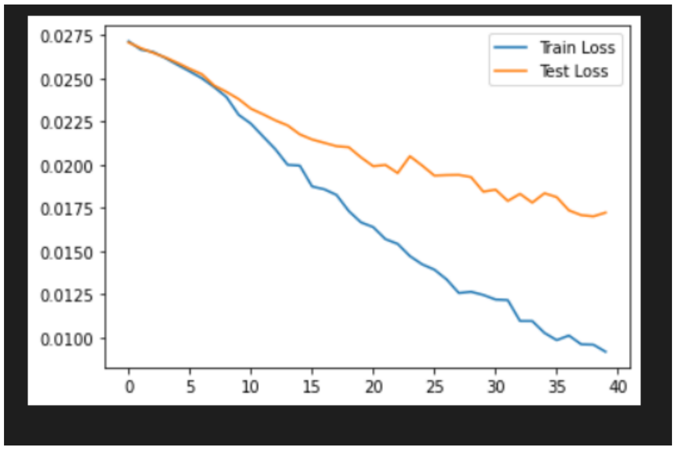


```
Model: "model"
__________________________________________________________________________________________________
 Layer (type)                   Output Shape         Param #     Connected to                     
==================================================================================================
 input_1 (InputLayer)           [(None, 128, 128, 3  0           []                               
                                )]                                                                
                                                                                                  
 zero_padding2d (ZeroPadding2D)  (None, 134, 134, 3)  0          ['input_1[0][0]']                
                                                                                                  
 conv1/conv (Conv2D)            (None, 64, 64, 64)   9408        ['zero_padding2d[0][0]']         
                                                                                                  
 conv1/bn (BatchNormalization)  (None, 64, 64, 64)   256         ['conv1/conv[0][0]']             
                                                                                                  
 conv1/relu (Activation)        (None, 64, 64, 64)   0           ['conv1/bn[0][0]']               
                                                                                                  
 zero_padding2d_1 (ZeroPadding2  (None, 66, 66, 64)  0           ['conv1/relu[0][0]']             
 D)                                                                                               
                                                                                                  
 pool1 (MaxPooling2D)           (None, 32, 32, 64)   0           ['zero_padding2d_1[0][0]']       
                                                                                                  
 conv2_block1_0_bn (BatchNormal  (None, 32, 32, 64)  256         ['pool1[0][0]']                  
 ization)                                                                                         
                                                                                                  
 conv2_block1_0_relu (Activatio  (None, 32, 32, 64)  0           ['conv2_block1_0_bn[0][0]']      
 n)                                                                                               
                                                                                                  
show more (open the raw output data in a text editor) ...
Total params: 18,572,645
Trainable params: 250,661
Non-trainable params: 18,321,984
```

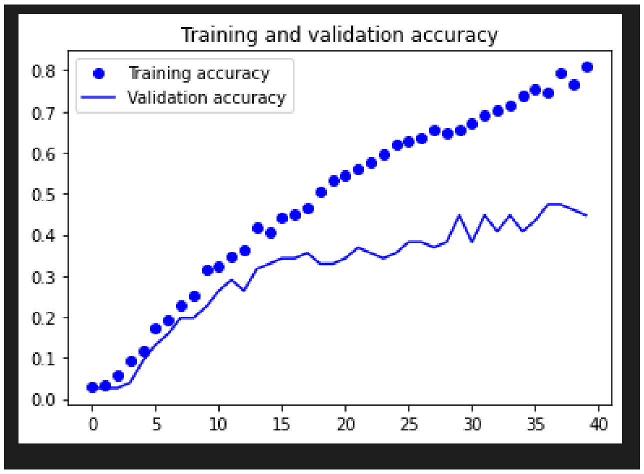
Results

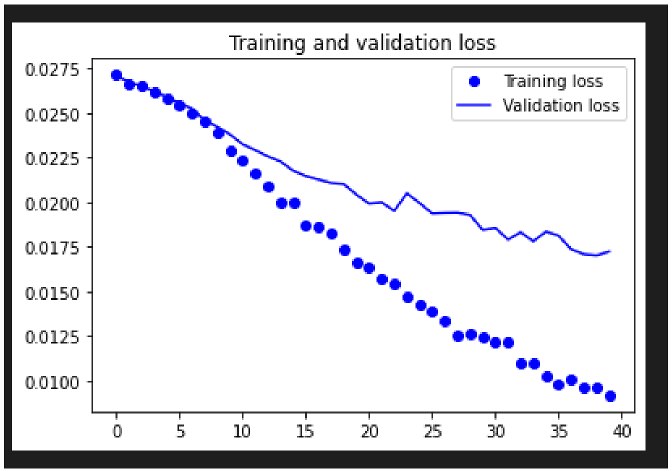
We firstly used 50 epochs, but we found the model started overfitting. For the results, we plotted the training accuracy and loss against the test accuracy and loss. With 50 epochs, the accuracy started spiking the test accuracy, so we believed it started overfitting. We lowered the epochs to 40 and found that the scores are much more correlated between the training and the test. Therefore, there may be slight underfitting/overfitting in the model. This can be seen in the test loss. Although it is decreasing, it does not decrease as suddenly as the train loss, so the model may be slightly overfitting. Lowering the epochs may fix this problem.



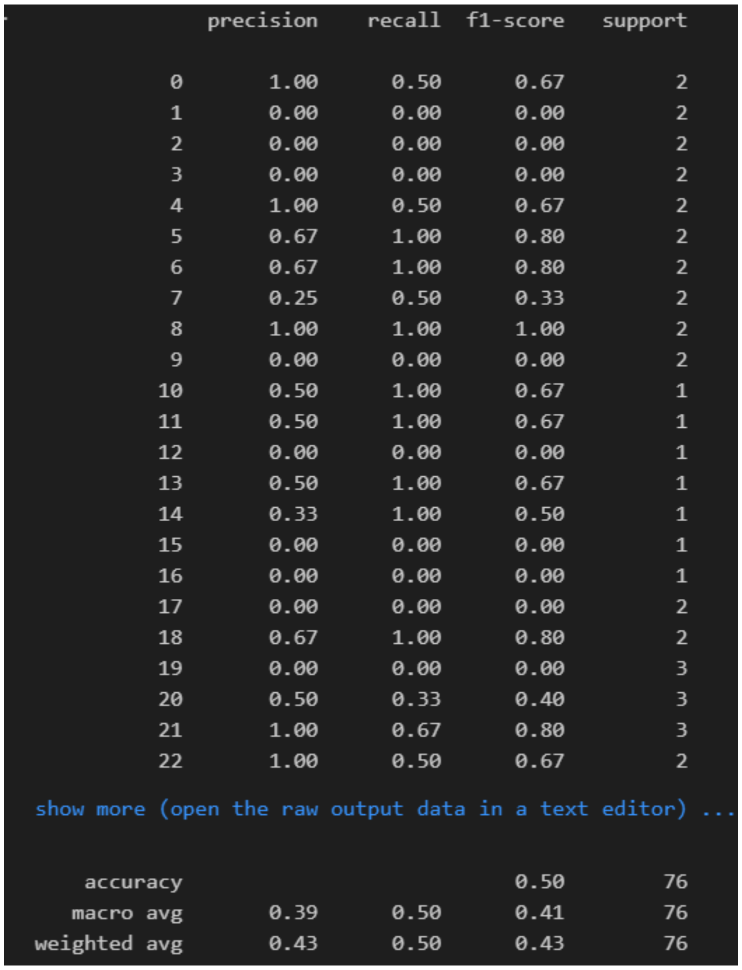


Below shows us again that the model may be overfitting slightly.





We then plotted the precision, recall and F1 score for each image in the test set. Below are some of the scores that we got.



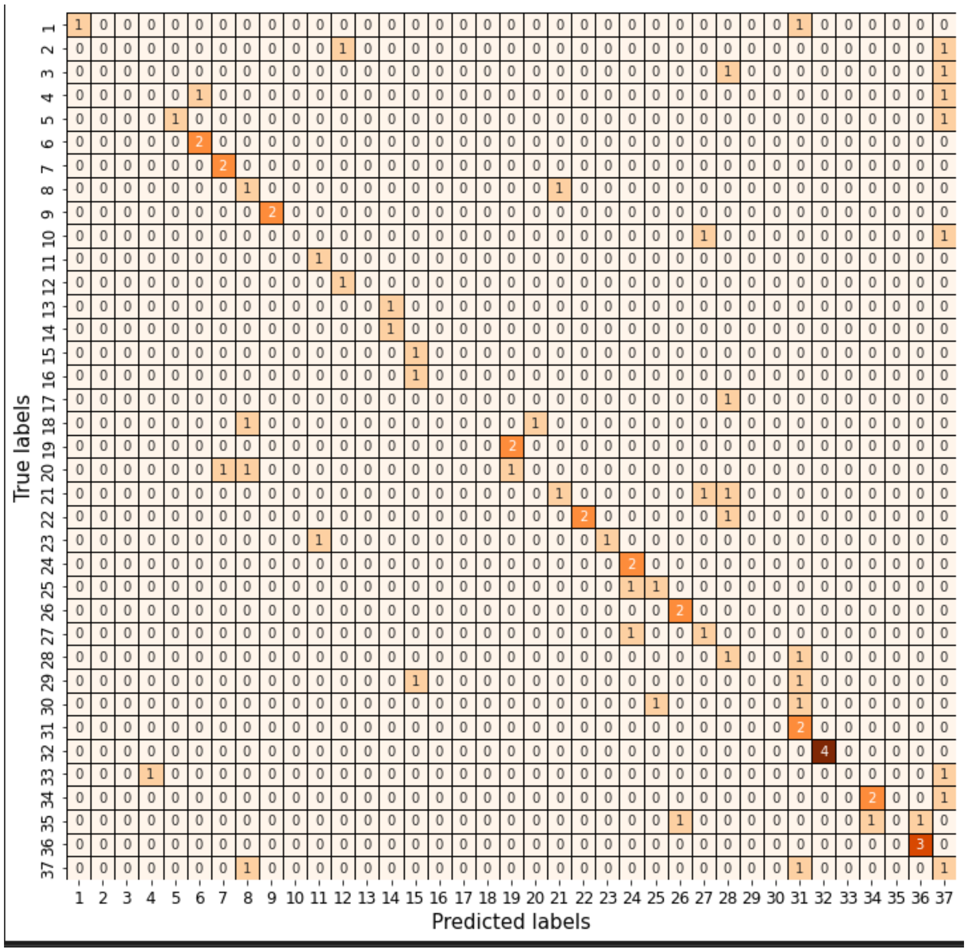
```
              precision    recall  f1-score   support

           0       1.00      0.50      0.67         2
           1       0.00      0.00      0.00         2
           2       0.00      0.00      0.00         2
           3       0.00      0.00      0.00         2
           4       1.00      0.50      0.67         2
           5       0.67      1.00      0.80         2
           6       0.67      1.00      0.80         2
           7       0.25      0.50      0.33         2
           8       1.00      1.00      1.00         2
           9       0.00      0.00      0.00         2
          10       0.50      1.00      0.67         1
          11       0.50      1.00      0.67         1
          12       0.00      0.00      0.00         1
          13       0.50      1.00      0.67         1
          14       0.33      1.00      0.50         1
          15       0.00      0.00      0.00         1
          16       0.00      0.00      0.00         1
          17       0.00      0.00      0.00         2
          18       0.67      1.00      0.80         2
          19       0.00      0.00      0.00         3
          20       0.50      0.33      0.40         3
          21       1.00      0.67      0.80         3
          22       1.00      0.50      0.67         2

  show more (open the raw output data in a text editor) ...

    accuracy                           0.50        76
   macro avg       0.39      0.50      0.41        76
weighted avg       0.43      0.50      0.43        76
```

Following on from this, we plotted a confusion matrix for each class id.

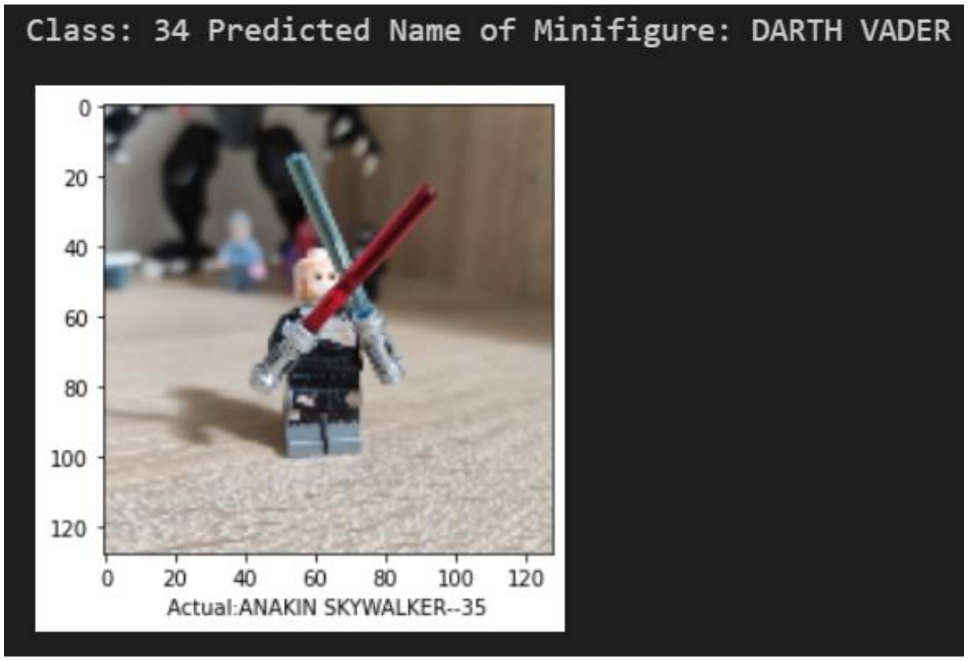


As we can see there is a medium to strong correlation between the true and predicted labels, so we felt our model is fairly accurate.

Evaluation of results

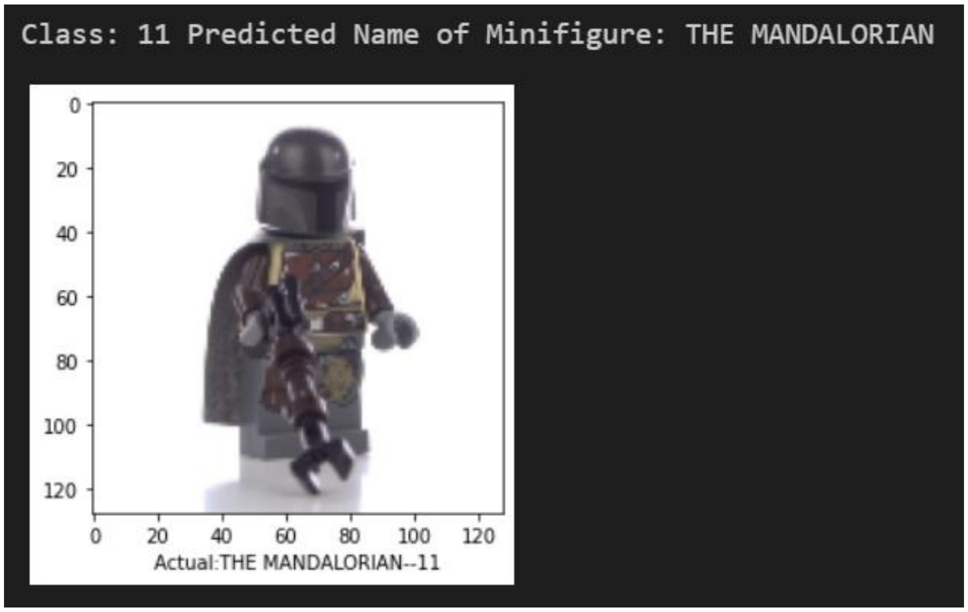
Since our model got 50-60% accuracy, hypothetically thinking our model should get around 50% accuracy when predicting the images. To evaluate the results, we took a sample of 20 images from the test set. We then passed these 20 images into the model and predicted an answer.

Below you can see a predicted image. The actual name of the minifigure is Anakin Skywalker, and has a class id of 35. The predicted value of our model is class id of 34 and the predicted name was Darth Vader.



```
Class: 34 Predicted Name of Minifigure: DARTH VADER
```

Here is an image the model predicted correctly.



```
Class: 11 Predicted Name of Minifigure: THE MANDALORIAN
```

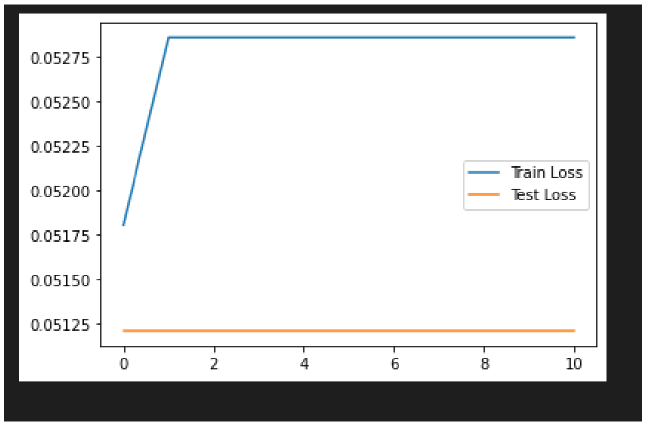
Our thinking is the model should get around 50% of images correct since the model's accuracy is around 50-60%. We manually counted the images to see how many it got right or wrong. Counting manually, 11 images were classified incorrectly, with 9 being correct. This is more or less 50%, so the results are accurate.

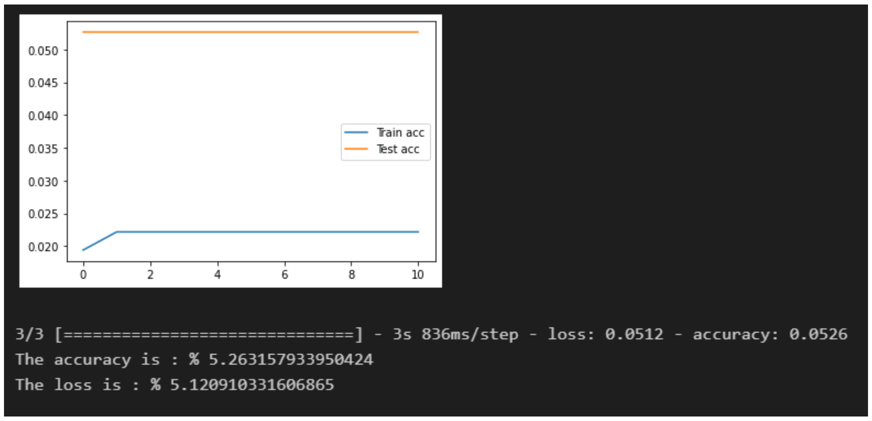
We also noticed that if the model predicted the wrong image, it would often predict from the same class id. For instance, if the image was from Star Wars the image was Darth Vader, the model often predicted a similar character such as Kylo Ren. This was an interesting observation we made.

Also, a note on the model accuracy. We reduced the images from 512x512 to 128x128. This was because it took over 40 minutes to train the model using 512x512 images on our machines. Since we are significantly reducing the image quality and resolution, the accuracy of the model will suffer. However, reducing the image in size significantly increased the model's performance by only taking 13 minutes to train the model now. Since this project has a due date, we found that reducing the image quality and size was something we had to do, to speed up the training time.

Impact of varying hyperparameters

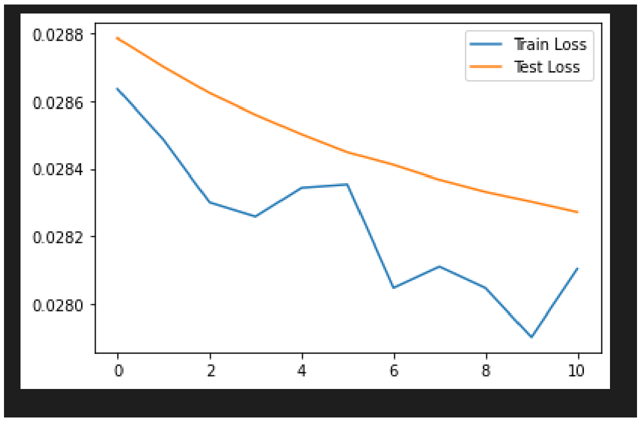
- Loss function: We attempted to use various loss functions for our model. We found using categorical cross-entropy the results were disappointing as it raised the loss by a significant amount. The loss was around 350% using this. We also attempted to use Sparse cross-entropy, but we received an error. Upon further research, we learned that Sparse cross-entropy is only used for when labels are expected to be integers. This did not make sense with our data. Finally, we implemented the MSE (mean squared error function); this gave us the best results for the loss with a loss of around 1%.
- Learning rate: During this project, we explored various learning rates' effects on the model. First, we observed a learning rate of 0.5, this caused our model to output a 5% loss and 5% accuracy. These results were not up to standard, so we continued to edit this parameter. We found using a learning rate of 0.0001 gave us the best results. This gave us a loss averaging 1% and an accuracy of 60%. Below is the learning rate at 0.5

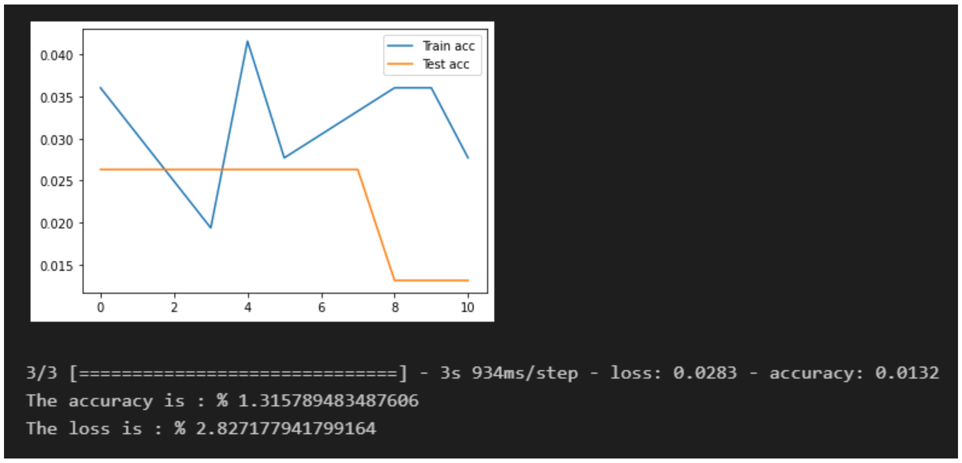




```
3/3 [==============================] - 3s 836ms/step - loss: 0.0512 - accuracy: 0.0526
The accuracy is : % 5.263157933950424
The loss is : % 5.120910331606865
```

- Optimizer: We first tried the optimizer Adagrad. Adagrad has various advantages as an optimizer, the learning changes at each parameter this automates the learning rate process. This feature would allow the optimizer to automatically set the learning rate. Adagrad is also good for training sparse data. As our data set was small, we thought this optimizer would be a good fit. However, Adagrad returned results that were not fully optimal. The accuracy dropped after 10 epochs; the model then stopped at this epoch to prevent over fitting. This led us to explore other options, we tried out the Adam optimizer as our next solution an found it gave back the best results with a 1% loss and 60% accuracy. Below is the Adagrad accuracy and loss.

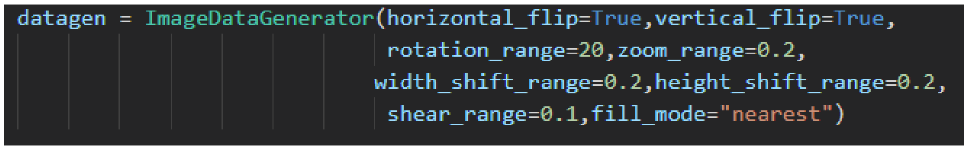




```
3/3 [==============================] - 3s 934ms/step - loss: 0.0283 - accuracy: 0.0132
The accuracy is : % 1.3157894834876O6
The loss is : % 2.827177941799164
```

Data Augmentation

Before we implemented data augmentation, our model would often get 100% accuracy. This was a result of the dataset being too small, only 361 images in size. For this we created an Image data generator as seen below.



```
datagen = ImageDataGenerator(horizontal_flip=True,vertical_flip=True,
                             rotation_range=20,zoom_range=0.2,
                            width_shift_range=0.2,height_shift_range=0.2,
                             shear_range=0.1,fill_mode="nearest")
```

This generator would generate a new image at runtime, it would rotate the image 20 degrees, zoom in, shear the image, and flip the image. This image would be generated at each epoch, so for 40 epochs, each epoch would have a different image of the same minifigure name and class id. This immensely helped reduce overfitting since our model would no longer produce 100% accuracy and now 50-60% accuracy. In addition, it increases the dataset size without the need for adding new images into the dataset. Data augmentation helped us in stopping overfitting since our dataset is generally small. We picked a small dataset since there may be complications when submitting the project if the dataset was gigabytes in size.