# AIML421, Final Project – Image Classification

Corvin Idler, ID 300598312, idlercorv@myvuw.ac.nz

## 1. Introduction

Recognising objects in images (e.g. dogs or cats) usually doesn't pose much of a problem to human beings, but for computers object recognition or image classification has historically been a hard to solve problem. Through the advent of deep learning and convolutional neural networks in particular, much progress has been achieved in this area. Deep convolutional neural networks (CNNs) are now able to achieve competitive or even superior performance to human beings on hard vision problems. The project underlying this report aims to understand how to construct CNNs for image classification purposes and how to best tune them and make them work properly.

### Scope and objectives of this project

• understand the basic image Classification pipeline and properly use available data for model training and hyper-parameter tuning.
• implement CNNs, train them, save and load trained CNNs and test them on unseen test data.
• understand the influences of different design choices and hyperparameters on training CNNs.
• understand different techniques to improve the CNN performance

### Data Set

The consists of a total of 6,000 jpeg compressed RGB images across 3 classes, with 2,000 images per class. The 3 classes are tomato, cherry and strawberry. The three classes of images have been manually made to be completely mutually exclusive, i.e. each image belongs to only one class.

*Figure 1: Example images: images in the three rows belong to the tomato, cherry, and strawberry*

### Tools

The experiments of this project were conducted on Saturn Cloud with the PyTorch deep-learning library. Most of the training was performed on the followng GPU: NVIDIA Tesla T4 (15GB Memory) with NVIDIA-SMI version 460.73.01 and CUDA Version: 11.2.

## 2. Problem investigation

### Image dimensions

I analysed the distribution of input image dimensions (c.f. Figure 2), with the majority being 300x300 pixel. One can resize these images with torchvision transforms to a common dimension, but I decided to investigate the outliers in more detail and found that they also constitute visual outliers in the sense that very often other items might dominate the picture or that they depict more a stylized version (cf. Figure 3) of the 3 classes (tomato, cherry and strawberry). I therefore decided to exclude these pictures from any further training. The class distribution of images that ended up

being included in the training set is as follows: {tomato: 1386, strawberry: 1380, cherry: 1375} and the class distribution for those that were excluded is {cherry: 19, strawberry: 18, tomato: 17}

| | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| (300,300) | 4141 | (201,251) | 3 | (284,177) | 1 | (366,138) | 1 | (870,900) | 1 | (178,283) | 1 |
| (219,230) | 1 | (275,183) | 3 | (221,228) | 1 | (310,163) | 1 | (800,957) | 1 | (202,250) | 1 |
| (299,169) | 1 | (181,278) | 1 | (262,193) | 1 | (277,182) | 1 | (301,168) | 1 | (217,232) | 1 |
| (220,229) | 1 | (215,234) | 1 | (301,167) | 1 | (214,236) | 1 | (256,197) | 1 | | |
| (261,193) | 1 | (181,279) | 1 | (183,276) | 1 | (182,277) | 1 | (259,194) | 1 | | |
| (225,225) | 19 | (300,168) | 1 | (238,211) | 1 | (253,199) | 1 | (254,199) | 1 | | |

*Figure 2: Frequency table of image dimensions of training data set*


*Figure 3: Outlier pictures (dimensions and content)*

Image normalisation was performed on all pictures with the following customary channel means [0.485, 0.456, 0.406] and channel standard deviations [0.229, 0.224, 0.225]. The means and standard deviations weren't calculated on the dataset of this project but instead taken from prior research done by the image classification community on the ImageNet2012 dataset (c.f. https://github.com/pytorch/vision/pull/1965). I determined it would be best (particularly once transfer learning is used) to stay in line with common practice rather than determining the mean and standard deviation intrinsic to the training dataset.

No further pre-processing was performed, as image augmentation steps are described under the methodology part of this report. No explicit feature selection was performed upfront, as this is inherently done by the deep learning network.

The following random generators were set to fixed random seeds: np.random.seed(2342), torch.manual_seed(2342), poutyne.set_seeds(2342). Regardless, unexplained variability in the results for some neural networks between various "ceteris paribus" trials was still encountered. As a counter measure the state of a neural network was saved before any experiments and reloaded before each experiment to make sure that for hyperparameter optimization the networks were comparable (e.g. in terms of random weight initialization). This seems to have led to better reproducibility of results and more consistent results across experiments.

# 3. Methodology

## 3.1 Additional training data / Image augmentation

After having filterd out "invalid" training examples as described in the previous step, the next step was to increase the number of training samples. Instead of downloading more images from the internet (e.g. ImangeNet or other sources) I decided to use image augmentation techniques. Transformations I included in my tests were random scaling between 0.8 and 1.0, random rotations up to 15 degrees as well as horizontal and vertical flips. I tried as well to include ColorJitter, but that didn't prove to be beneficial.

It seems common to only use random transformations on a training data set, which changes a defined proportion of the image dataset each epoch during the training phase. This way the neural

network gets exposure to a certain amount of new (augmented) images each epoch and can try to generalize better instead of just "memorizing" the training dataset.

In experiments I found it was more beneficial to create one large, augmented dataset up front (deterministically flipping images horizontally, vertically and both and then adding each of these images sets to the original dataset). I then subjected this data set to random transformations (scaling, rotation) and added the results again to the previous data set. This led to an eightfold increase in training data.

### 3.2 Cross-validation

I decided to use one single validation data set consisting of 20% of the images for the purpose stopping the training of a neural net once the classification accuracy on the validation data set drops or doesn't increase for three epochs in a row ("early stopping" with patience). After early stopping I would then restore the state of the network that had the best accuracy score of all epochs.

Initially I made the mistake to split the image data set only after the image augmentation step. This can lead to a heavy "contamination" of the validation data set with images similar (slightly scaled or slightly rotated) to those in the training dataset. I therefore ensured programmatically that original images and their transformations are tied together when entering the training dataset vs. validation dataset. I also decided not to include any transformed versions of original images in the validation dataset, to make it as close as possible to something likely to be encountered in an unseen test dataset. Furthermore, I realized that in my original approach class balance wasn't maintained (particularly in the validation data set). I therefore used stratified sampling (without replacement) to ensure programmatically that class balance is maintained. I ended up with 26496 training images and 829 validation images.

### 3.3 Hyperparameter settings – batch size

I experimented with batch sizes of 8, 16, 32, 64 and 256. For big networks during my transfer learning experiments batch sizes above 32 exceeded the memory of the GPU and were therefore not even an option. On my initial CNN network (without transfer learning) I used 8, 64 and 256 as hyperparameters for the batch size for experiments (everything else being equal).

| | 8 | 64 | 256 |
|---|---|---|---|
| Max training accuracy | 92.089 | 85.134 | 73.596 |
| Max validation accuracy | 74.548 | 71.653 | 65.983 |
| Training time | 21.68 | 19.79 | 19.63 |

The results are rather interesting. Larger batch sizes are supposed to lead to a speed up in training by making better use of the GPU, but from the results we can see that the time savings are miniscule. Despite early stopping being used during the trials, all 3 experiments made full use of 10 training epochs and particularly for batch sizes 64 and 256 the validation accuracy didn't plateau, so it is possible that these settings could have achieved similar accuracy levels as batch size 8 if granted more than 10 episodes. The additional time required for using more episodes clearly would have overpowered any time savings from using larger batch sizes, which defeats the purpose of their use. Any further experiments were therefore performed with batch size 8.

## 3.4 Hyperparameter settings – learning rate

Hyperparameter experiments for the learning rate of the ADAM optimizer were done on my initial CNN. The following learning rates were trialled: 0.00001,0.0001, 0.001 and 0.01

It is evident, that high learning rates considerably speed up the training time due to the early stopping rule kicking in after very few epochs. A profound negative effect on both the training and validation accuracy can be noted on the downside of things. This is probably due to missing good local or global minima by overshooting. As can be seen in the confusion matrix, all instances in the validation data set were classified as tomato when the learning rate was set to 0.01. This suggests a very deviant neural network. That said, a too low training rate is also problematic (in our case due to excessively slow convergence). The training efforts "timed out" after 10 epochs without the validation accuracy having plateaued. In the light of the above findings, the training rate of 0.0001 was used for all other experiments.

| | 0.00001 | 0.0001 | 0.001 | 0.01 |
|---|---|---|---|---|
| Max training accuracy | 69.75 | 92.09 | 86.36 | 33.53 |
| Max validation accuracy | 63.21 | 74.55 | 67.19 | 33.53 |
| Training time | 21.65 | 21.68 | 10.80 | 8.66 |

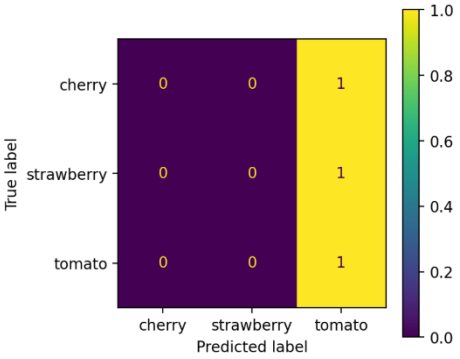



*Figure 4: Initial CNN design with ADAM training rate of 0.01*

## 3.5 Regularisation strategy

As described under section 3.2, an early stopping rule was used to prevent overfitting. If accuracy on the validation data set drops or doesn't increase for three epochs in a row (early stopping with patience), the training will stop and the model state from the epoch that achieved the best accuracy on the validation dataset will be restored. Some experiments were performed for the use of two drop out layers in my initial CNN. The CNN without dropout layers exhibited a steeper improvement of the accuracy on the validation dataset, which also plateaued earlier than in the case of the CNN with drop out layers (c.f. figure 5 and 6). The accuracy improvement curve was also smoother when no dropout layers were used. In terms of end results after 10 epochs there was not much of a difference (neither for training accuracy nor for validation accuracy). Further hyperparameter optimisation could have been attempted to optimize the architecture design and parametrisation of the dropout layers, but this wasn't pursued further, due to the dominant results of the transfer learning experiments in comparison to the initially designed network.

| | No dropout | dropout |
|---|---|---|
| Max training accuracy | 90.399 | 92.09 |
| Max validation accuracy | 75.513 | 74.55 |
| Training time | 21.62 | 21.68 |

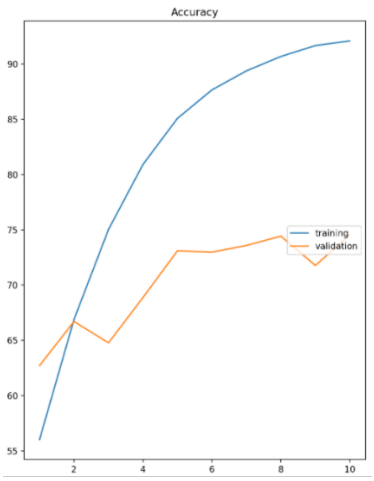



*Figure 5: Training trajectory with drop out layers*

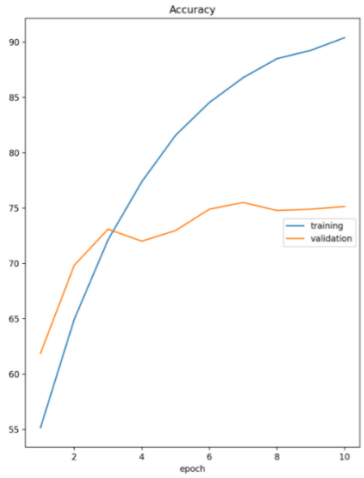



*Figure 6: Training trajectory without drop out layers*

### 3.6 Transfer learning

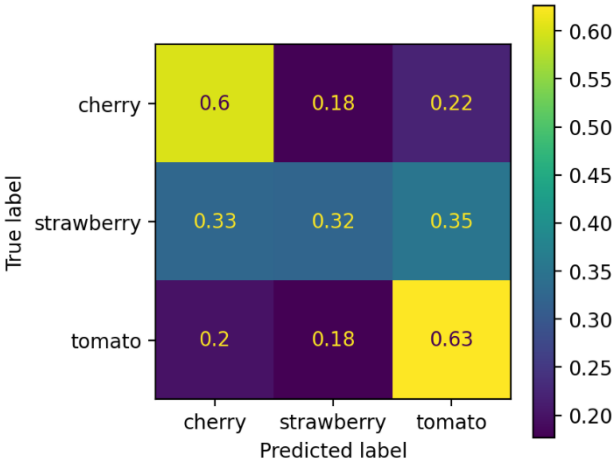
The use of existing pre-trained models provided a quantum leap in terms of performance. I evaluated the VGG-16 [1], ResNet-50 [2], EfficientNet-b2 [3] model, all of which were pretrained on the 1000-class ImageNet dataset. For transfer learning all layers of a pretrained network have their weights frozen and only the last two layers are modified and retrained to account for a different number in classes and also for the different objects per class compared to the original data set. The results will be discussed in more detail in the following section 4.

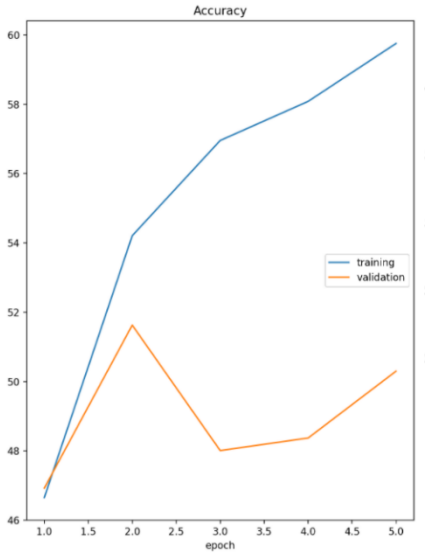
## 4. Results

### 4.1 Baseline MLP

For the baseline MLP I flattened the image as a first step, leading to a 3*300*300 vector input vector. The first layer consisted of 255 neurons to which the 270000 inputs connected. This leads to roughly 69 Million parameters (weights) to be estimated in the first layer. The second layer consist of 3 neurons to perform the final classification function, leading to 768 parameters for this layer. For the amount of training data available this network is hopelessly overparametrized in my opinion. Due to the lack of convolutional layers any spatial information is also lost or at least a lot harder to capture. One can see that the training accuracy keeps improving over the course of all epochs, but the validation accuracy is hardly improving at all. This is a strong sign of the network failing to generalize as it gets better and better at basically just memorizing the training data. We can see that the network struggled most with correctly identifying strawberries for which the network basically failed to exceed the performance of random guessing. The training time was “moderate” in comparison to the time taken for some transfer learning training tasks, which is partly due to the early stopping which ended the training phase after 5 epochs.

```
----------------------------------------------------------------
        Layer (type)               Output Shape         Param #
================================================================
            Linear-1                [8, 255]      68,850,255
            Linear-2                  [8, 3]             768
================================================================
Total params: 68,851,023
Trainable params: 68,851,023
Non-trainable params: 0
----------------------------------------------------------------
Input size (MB): 8.24
Forward/backward pass size (MB): 0.02
Params size (MB): 262.65
Estimated Total Size (MB): 270.90
-
```




Accuracy training: min: 46.65, max: 59.76x
validation: min: 46.92 max: 51.63

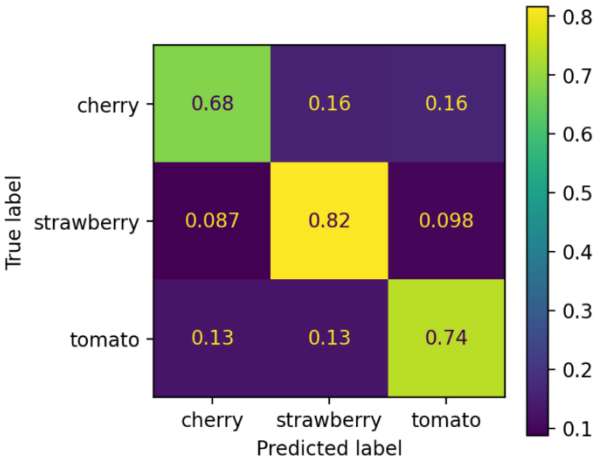
training time in minutes: 16.8

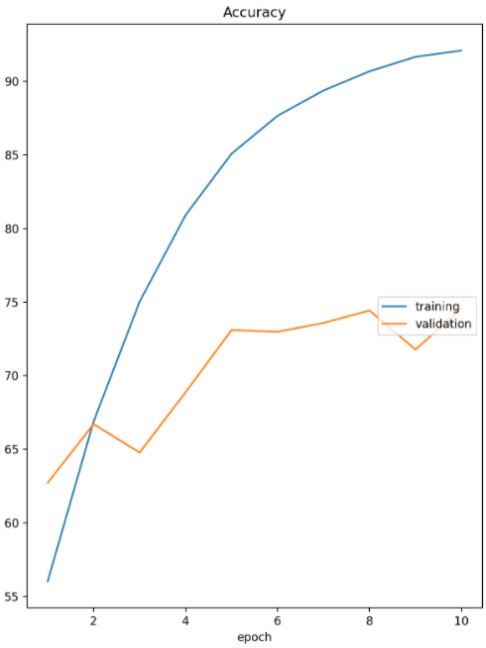
## 4.2 Initial CNN

The second network I trialled was based on the CNN from the Pytorch CIFAR10 tutorial[1] from the Pytorch documentation. This network was the main working horse for hyperparameter experimentation and image augmentation experiments. The network contains two convolutional layers (with MaxPooling) and three linear layers. The below results are the best that were achieved during these experiments. The number of free parameters is by a factor 7 smaller compared to the above baseline MLP. Ironically the network excels at classifying strawberries in contrast to the baseline MLP. The network reached an impressive (compared to the MLP) training accuracy of 92% and slowly started plateauing at that level. The validation accuracy was around 74% and also started to flatten out after 10 epochs. The training time was a bit longer than for the MLP, but the network was also trained for 10 instead of only 5 epochs. Interestingly the validation error of the first epoch is above the training error of this epoch. This is most likely due to the effect that the training error of the epoch is the cumulative error of all batches including the error of the first hundred or even thousand batches on an almost untrained network, while the validation error is based on a network that was already trained once on all batches.

[1] https://pytorch.org/tutorials/beginner/blitz/cifar10_tutorial.html

```
----------------------------------------------------------------
        Layer (type)               Output Shape         Param #
================================================================
            Conv2d-1          [8, 6, 296, 296]             456
         MaxPool2d-2          [8, 6, 148, 148]               0
            Conv2d-3         [8, 16, 144, 144]           2,416
         MaxPool2d-4          [8, 16, 72, 72]               0
           Dropout-5          [8, 16, 72, 72]               0
            Linear-6                [8, 120]       9,953,400
            Linear-7                 [8, 84]          10,164
           Dropout-8                 [8, 84]               0
            Linear-9                  [8, 3]             255
================================================================
Total params: 9,966,691
Trainable params: 9,966,691
Non-trainable params: 0
----------------------------------------------------------------
Input size (MB): 8.24
Forward/backward pass size (MB): 70.50
Params size (MB): 38.02
Estimated Total Size (MB): 116.76
----------------------------------------------------------------
```




Accuracy training: min: 56.031, max: 92.089

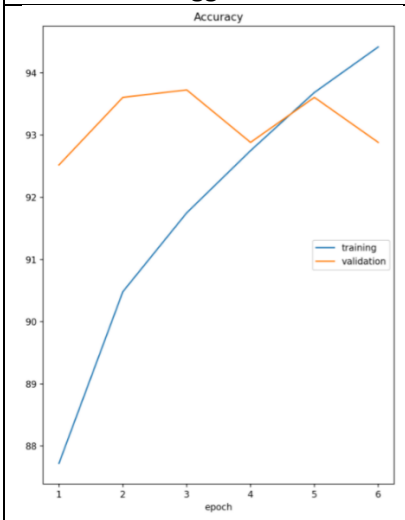
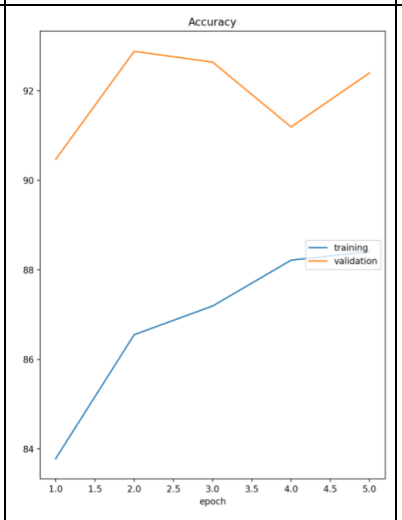
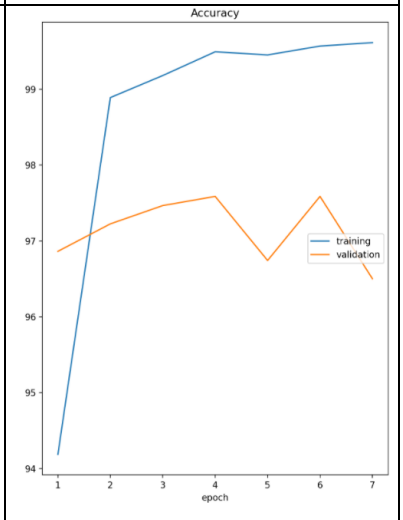
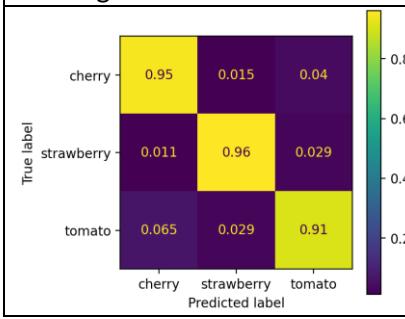
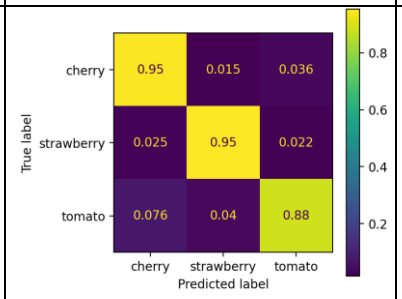
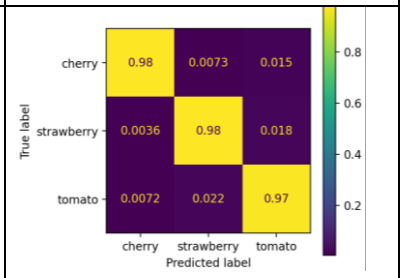
validation: min: 62.726, max: 74.548

training time in minutes: 21.68

## 4.3 Transfer Learning CNN

After I struggled to increase the performance of my initial CNN beyond ca. 75% accuracy I decided to look into transfer learning as described in section 3.6

A comparison of the results of vgg16, resnet50 and eccifientnet-b2 can be found below. All networks achieved validation accuracy results above 90% and therefore outperform my initial CNN by a considerable margin. It is interesting to see that for resnet50 the validation accuracy was consistently above the training accuracy. This effect could also be witnessed for vgg16 for some episodes. One possible explanation is that the un-augmented images in the validation dataset are more similar to the types of images the network had initially been trained on, while the augmented images were too dissimilar from the images in the ImageNet data set and therefore require more training to reach good accuracy levels. The EfficienNet-B2 is the exception to this rule. This network also demonstrated an exceptionally steep accuracy improvement curve which more or less plateaued at around 99% training accuracy after only 3 episodes. Efficientnet-B2 performed best but also required the longest time for training. VGG16 performed second best and also exhibited the second longest training time. Resnet50 came in third which is slightly surprising as it outperformed the vgg16 net on the ImageNet 1000 dataset [4]. All three nets struggle most with classifying tomatoes. The effect is most pronounced for resnet50, slightly less for vgg16 and hardly noticeable for efficientnet-b2.

| vgg16 | resnet50 | efficientnet-b2 |
|---|---|---|
|  |  |  |
| Accuracy training:<br>min: 87.72, max: 94.41 | min: 83.79, max: 88.41 | min: 94.19, max: 99.62 |
| validation:<br>min: 92.52, max: 93.73 | min: 90.47, max: 92.88 | min: 97.59, max: 96.50 |
| training time in minutes: 39.77 | 20.48 | 73.52 |
|  |  |  |

Given the strong results of EfficientNet-B2 I chose it as my final network for submission.

## 5. Conclusions and future work

### 5.1 Conclusion

It is obvious from the results in this report that transfer learning is an incredibly powerful approach that can leverage the feature extraction capabilities of networks that were trained on very large data sets. One potential drawback might be, that one could end up with networks that are far larger than needed to a given task. For example, the networks in this report were all trained to distinguish 1000 classes of images/objects while the task at hand requires only the distinction of 3 classes. Due to the complexity of these networks, it might take a long time to re-train or “finetune” them even though only a small part of the network would have to be retrained. This is evident for example in the training time of the EfficientNet-B2

### 5.2 Future work

In terms of future work, it could be interesting to experiment more with image augmentation techniques like various jpeg compression levels (c.f. albumetations library) or the concept of image mix up. The use of Batch Normalisation could also have been and interesting addition to my initial CNN and might be interesting to explore in the future.

# References


[1] K. S. a. A. Zisserman, “Very Deep Convolutional Networks for Large-Scale Image Recognition,” in *3rd International Conference on Learning Representations*, San Diego, CA, USA, 2015.

[2] K. a. Z. X. a. R. S. a. S. J. He, “Deep Residual Learning for Image Recognition,” in *IEEE Conference on Computer Vision and Pattern Recognition*, Las Vegas, NV, USA, 2016.

[3] M. a. L. Q. Tan, “EfficientNet: Rethinking Model Scaling for Convolutional Neural Networks,” in *Proceedings of the 36th International Conference on Machine Learning*, Long Beach, California, USA, 2019.

[4] S. B. a. R. C. a. L. C. a. P. Napoletano, “Benchmark Analysis of Representative Deep Neural Network Architectures,” *IEEE Access,* vol. 6, pp. 64270--64277, 2018.