**Deep Learning**

**Graded Assignment 2:**

**CIFAR 100 (with 20 classes)**

**Group 08**

Bhawna Dixit (02014095)

Thomas Kok (02013791)

Emiel Maes (01308305)

**Table of Contents**

[**Problem description**](#_7b990umyo8op) **2**

[**Unregularized model**](#_meit9qlk26ik) **2**

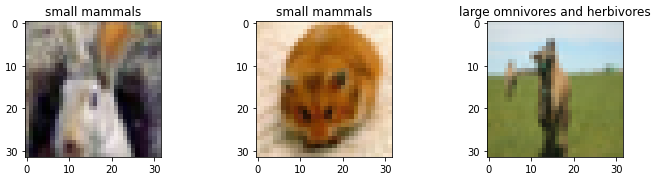
[**Regularization**](#_cmi0jrr46tfs) **19**

[**Augmentation**](#_evmnimhesh7v) **19**

[**Final model result**](#_4twnimjdq8dm) **22**

# Problem description

For the second assignment, the goal is to classify the CIFAR 100 dataset which consists of labeled pictures. The dataset has both ‘fine labels’ which results in 100 classes and ‘coarse labels’ which results in 20 classes. Here, the ‘coarse’ classification task is considered in which each picture has to be classified into 1 out of the 20 classes. Some examples are visualized in figure 1. The dataset has a training set of 50,000 32x32 colour images and a test set of 10,000 32x32 colour images. During our model training, the training set is split into 44,000 samples for training and 6,000 samples for validation. It must also be noted that each class has the same amount of samples which means that we are dealing with a balanced dataset.



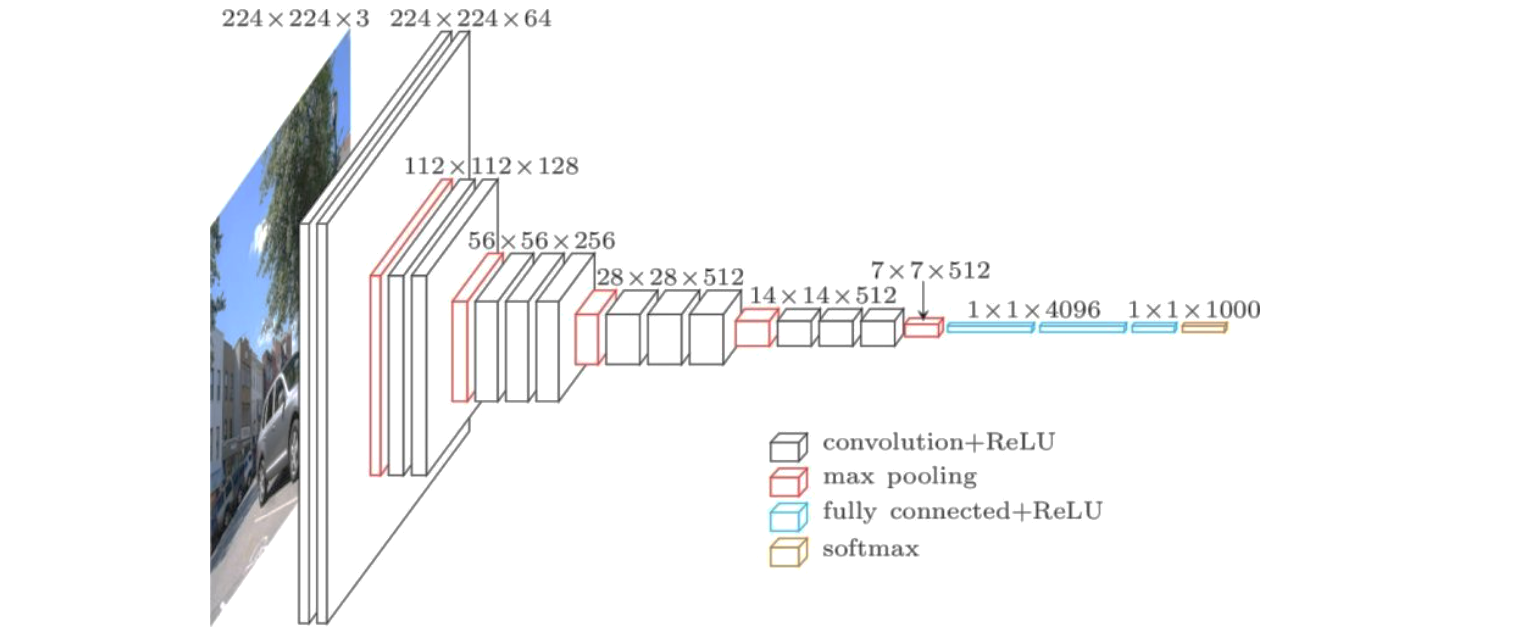
*Figure 1: Samples from the data*

The objective of the second assignment is to train a model which is able to classify the images very well. However, the focus is to really understand how each part of the architecture (type of layer, number of neurons, learning rate, regularization technique..) affects the performance of the model. Therefore, the assignment is split into several steps which are defined as follows:

* The first step is to create an initial model which is powerful enough to classify the training samples. Typically, some overfitting will occur and there will be a gap between the training and validation accuracy. In order to find the optimal architecture, we change each hyperparameter individually, while fixing the values for all the other hyperparameters. Once all the different hyperparameters are considered, it is possible to select the best performing model and continue to the second step.
* The second step is to optimize the validation performance and close the gap between the training and validation accuracy. This is typically done by using regularization techniques. In the second step of the assignment, several regularization techniques such as L1, L2, maxnorm and dropout layers are investigated. Here, again each hyperparameter is being tuned individually while keeping the other hyperparameters fixed.
* Once the gap between the training and validation accuracy is less big due to the regularization techniques, it is still possible to evaluate the impact of data augmentation. In this third step, each image is augmented which means that certain modifications (translation, rotation, zoom, shear..) are applied to the image such that the training data size is increased. This should result in a better performance of the model as more data is available.
* The final step is then to train the regularized model on the whole training set (the 50,000 samples), along with the augmentation technique and then check the performance on the unseen test data. The next sections will discuss these steps more in detail and give an overview of the methodologies we used.

# Unregularized model

In order to not start with a completely random model, some prior research was done to determine our initial model architecture. Having read the VGG16 paper[[1]](#footnote-0), we decided to start with a batch size of 128 as this is somewhere in between the 32 which was given in the start notebook and below the 256 which was used in the VGG16 paper. Also the number of epochs was set to 30 for the initial model (based on prior experience from assignment 1). The Adam optimizer was used with its initial learning rate set to 0.001 (default value). However, these are just some initial values and will be further tuned in our process. As we assume we are dealing with a somewhat less complex task than the one described in the VGG16 paper (200 classes and +- 500.000 training samples vs 20 classes and +- 50.000 training samples), we decide to start with the first 2 “block structure” layers from the VGG16 paper and add a fully connected dense net of 512 neurons, followed by an output layer of 20 neurons as we have 20 classes.



*Figure 1: VGG16 Network architecture[[2]](#footnote-1)*

The initial architecture we use looks as follows:

* Block 1: 3x3 Conv2D 64 - Relu - 3x3 Conv2D 64 - Relu - Maxpool 2x2
* Block 2: 3x3 Conv2D 128 - Relu - 3x3 Conv2D 128 - Relu - Maxpool 2x2
* Flatten layer
* Dense layer with 512 neurons
* Dense layer with 20 neurons

The table below describes our tuning process in which we have described the changes that were made to the model and hyperparameters during each step. Next to that, we discuss how the optimal architecture changed at the end of each step compared to the previous step. Also the train and validation accuracies are given from the best obtained model when the step was completed.

| **Step** | **Model architecture or hyperparameter which is being tuned during this step** | **Optimal architecture at the end of this step** | **Train/**  **Valid accuracy** | **Notebook version** |
| --- | --- | --- | --- | --- |
| 0 | Initial architecture based on first 2 blocks of the VGG net | The initial model architecture was described above. | 98%  52% | GA2\_Training\_v0 |
| 1 | Check performance when adding Batchnorm Layers in the blocks (before or after relu function) | Same architecture as the one in step 0, but now with batchnorm layers before the relu activation function. As we can see, the performance on the training accuracy drops while the validation accuracy increases. This is mainly due to the fact that batchnorm also adds some regularization to the model. We decide to continue with the batchnorm layers as we can increase the power of our model later on by adding more channels or more blocks. | 93%  58% | GA2\_Training\_v1 |
| 2 | Defining the number of blocks | Adding more blocks (we copy the last block which was defined as Block 2 in the initial model, but now also with the batchnorm layers included as well) seems to improve the performance of the model as the training accuracy increases compared to the previous obtained training accuracy. However, adding too many blocks seems to decrease the performance again. The optimal number of blocks to add was 1 which gave the train/test accuracy of 95%/64%. When adding 2 blocks, this accuracy decreased a little to 93% and 63%). | 95%  64% | GA2\_Training\_v2 |
| 3 | Defining the number of layers per block | For this step, we try to see if the number of Conv2D/BatchNorm/ReLu layers in each block is appropriate. The number of such layers was 2 in step 2, we tried 1 and 3 here, as well as additional configurations with increased/decreased number of blocks to compensate for the changed number of layers. None of the configurations improved upon the 2 layer-block. This means that the optimal architecture after this step is still the same as the one obtained from the previous step. | 95%  64% | GA2\_Training\_v3 |
| 4 | Number of channels per block | We have three blocks with corresponding number of channels in each block: 64, 128, 128. We will keep the non-decreasing (because this makes most sense with CNNs conceptually) number of channels, but try several different configurations (different number of channels within a block and different number of channels over the blocks). Empirically, 64 -> 128 -> 512 got the best results and also the most impressive training curves. This architecture obtained a training accuracy of 98%. | 98%  65% | GA2\_Training\_v4 |
| 5 | Kernel size | Several different kernel sizes were used in the ‘blocks’ ranging from 2x2 to 5x5. Also a combination of 5x5 in the first block and 3x3 in the following blocks was inspected but none of them seemed to improve the accuracy from the current model that was the best so far. We concluded to continue with the already existing 3x3 kernel sizes. | 98%  65% | GA2\_Training\_v5 |
| 6 | Pooling size and type | Max pooling is better able to extract the most important features like edges while average pooling extracts more smooth features. Therefore, we expect that the MaxPooling2D will result in higher performance. However, we also looked at AveragePooling2D and our initial thoughts were confirmed when inspecting the plots. Next to the pooling types, we also considered the pooling sizes (2x2), (3x3) and (5x5). However, the (3x3) pooling size still showed the best result and therefore no changes were made to the current model. | 98%  65% | GA2\_Training\_v6 |
| 7 | Number/neurons of dense layers | Here, we tried several different architectures (both wider and deeper) in the dense layers. Some of the architectures we tried look as follows: 1024->512->20; 512->256->128->20; 256->20 but none of them seemed to improve the existing performance, therefore we chose to continue with the existing model architecture. We also tried Global Average Pooling (and Global Max Pooling) as alternatives to the fully connected layers, but | 98%  65% | GA2\_Training\_v7 |
| 8 | Initial learning rate and batch size | Several values were used for the initial learning rate, starting from the default rate which is 0.001, we also considered 0.0005, 0.00025 and 0.00001. However, by changing the initial learning rate, still some fluctuations in the training accuracies could be seen on our plots. Therefore, we had to iterate back and forth between adjusting the initial learning rate and the batch size. Batch sizes that were considered are 128, 256, 512 and 1024.The learning rates 0.0005 and 0.00025 seemed to give similar performance in combination with a batch size of 512. As they performed similarly, we chose to continue with the 0.00025 learning rate. |  | GA2\_Training\_v8 |

*Table 1: Tuning process for unregularized model*

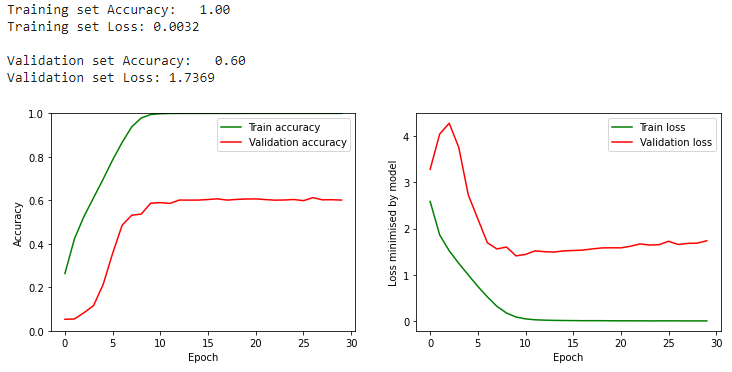
The final model architecture looks as follows:

* Block 1:
  + 3x3 Conv2D 64 -> Batchnorm -> Relu
  + 3x3 Conv2D 64 -> Batchnorm -> Relu
  + Maxpool 2x2
* Block 2:
  + 3x3 Conv2D 128 -> Batchnorm -> Relu
  + 3x3 Conv2D 128 -> Batchnorm -> Relu
  + Maxpool 2x2
* Block 3:
  + 3x3 Conv2D 512 -> Batchnorm -> Relu
  + 3x3 Conv2D 512 -> Batchnorm -> Relu
  + Maxpool 2x2
* Flatten layer
* Dense layer with 512 neurons
* Dense layer with 20 neurons

And has the following hyperparameters:

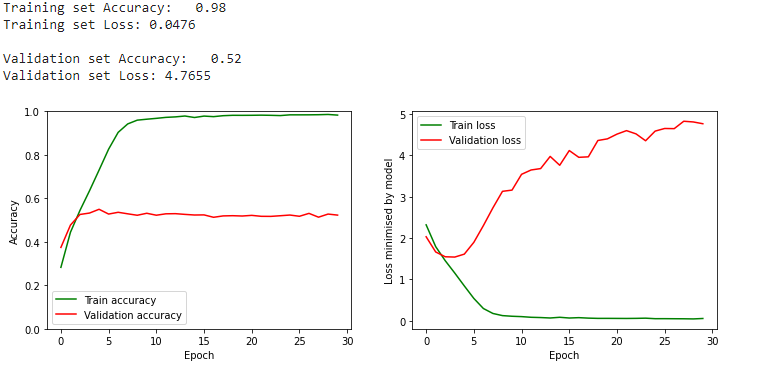
* Batch size of 512
* Initial learning rate of 0.00025
* Number of epochs: 30

The plot for the corresponding model looks as follows. As can be seen, the model is highly overfitting on the training data as it reaches 100% accuracy. It also has a very steep curve as can be seen from the training accuracy.

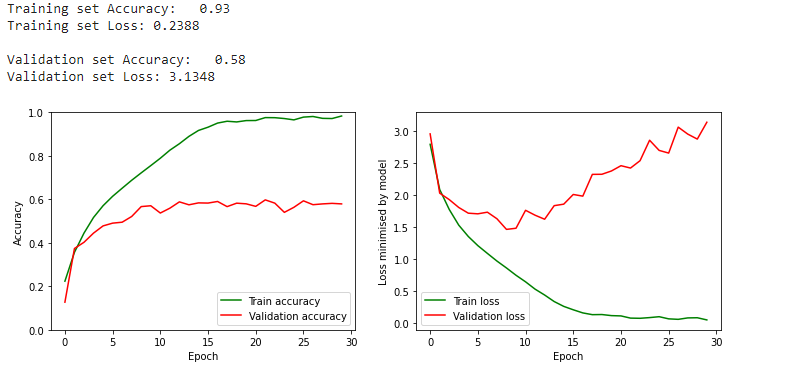


*Figure 2: final model performance*

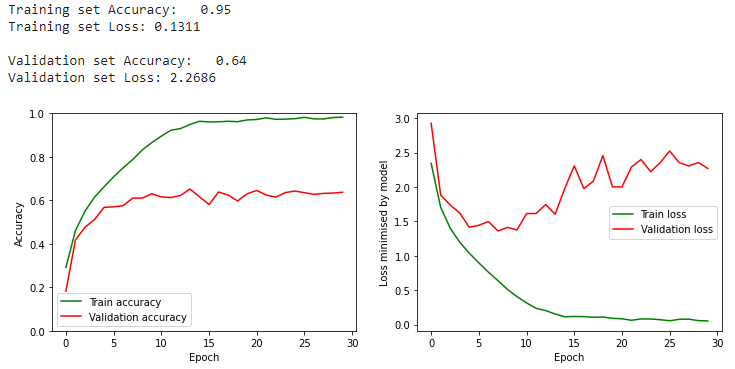
Intermediate model 0 is visualized below



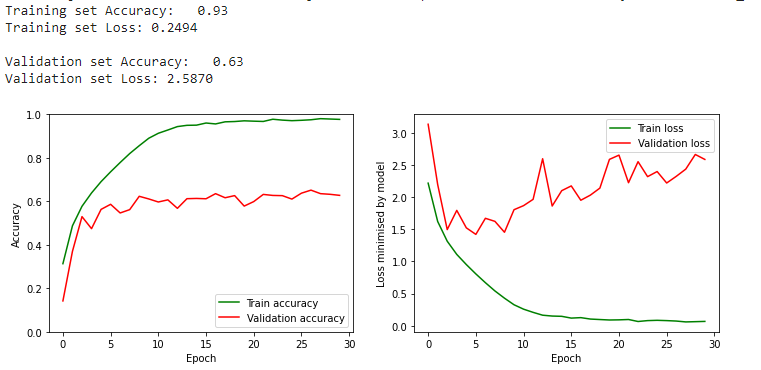
Intermediate model 1 (with batchnorm layers)



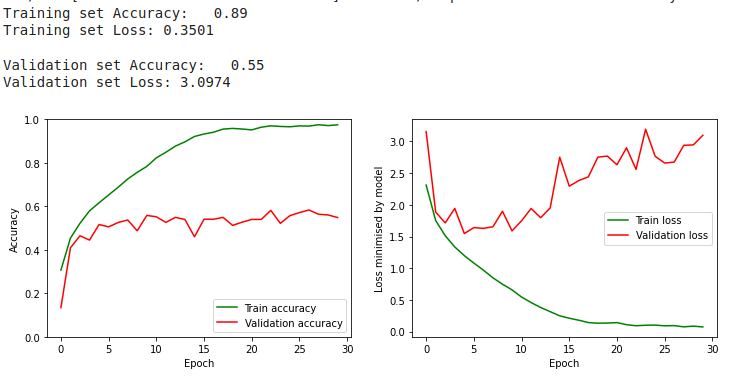
Intermediate model 2 (with 3 blocks)



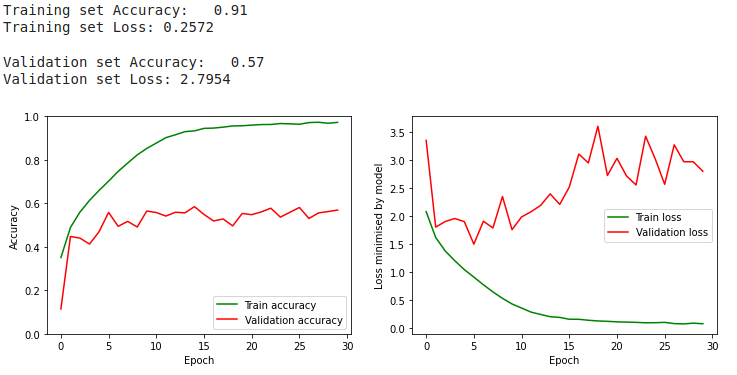
Intermediate model 2 (with 4 blocks)



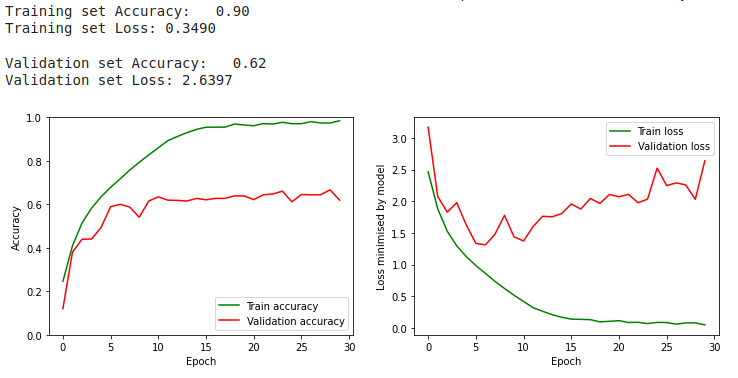
Intermediate model 3 (with 1 conv layer per block)



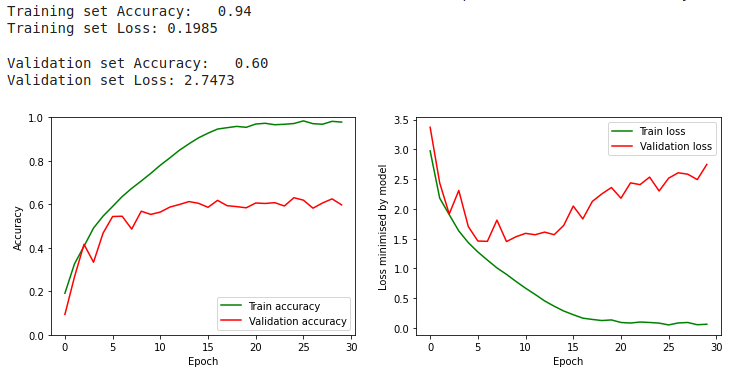
Intermediate model 3 (with 1 conv layer per block and 2 added blocks)



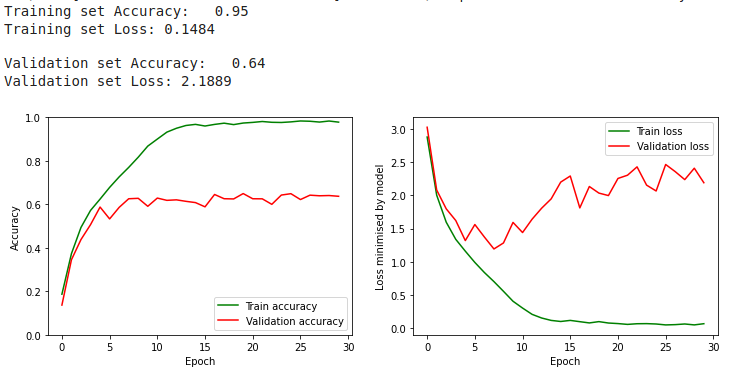
Intermediate model 3 (with 3 conv layers per block)



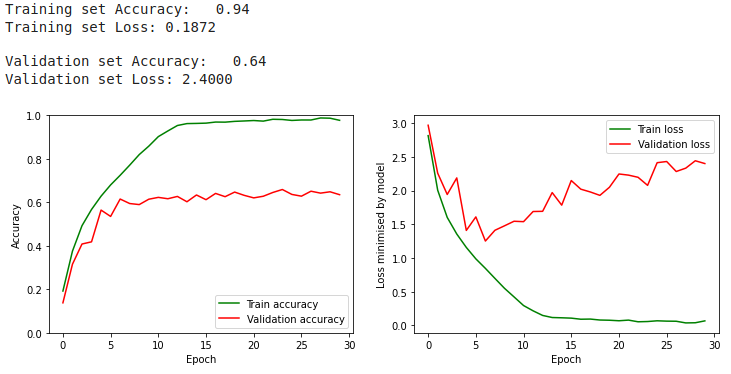
Intermediate model 3 (with 3 conv layers per block and 1 removed block)



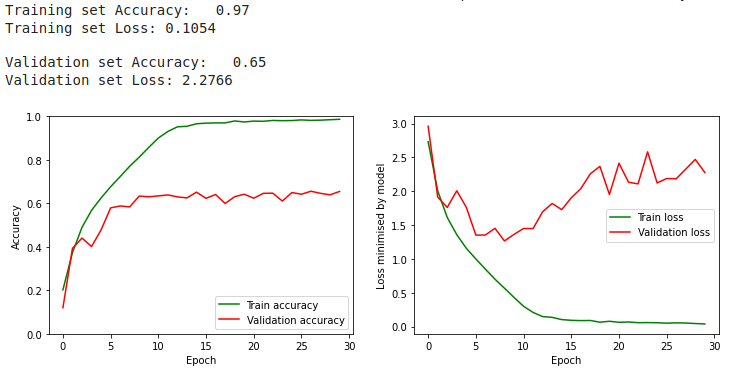
Intermediate model 4 (with 64, 128, 256)



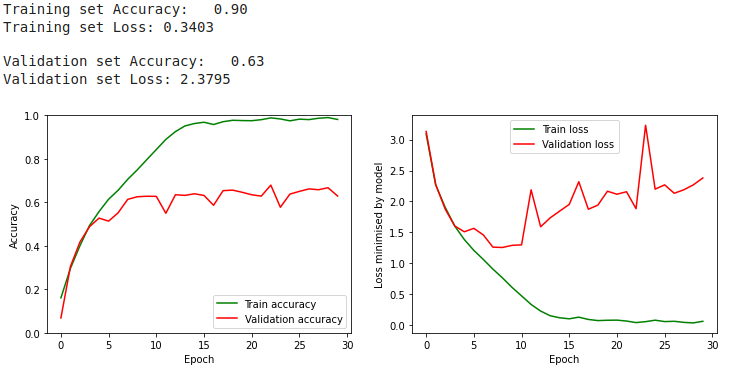
Intermediate model 4 (with 128, 128, 256)



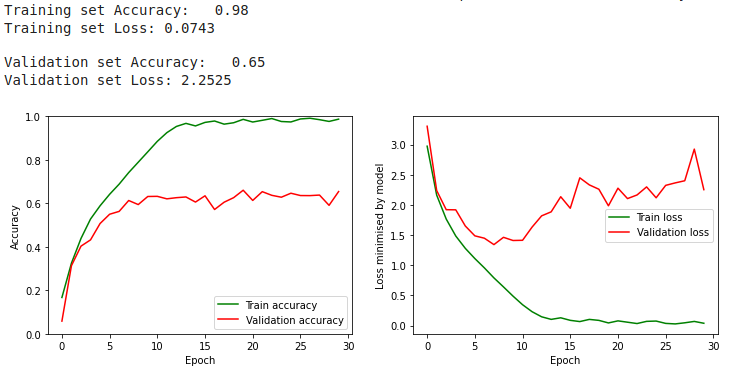
Intermediate model 4 with (64, 256, 256)



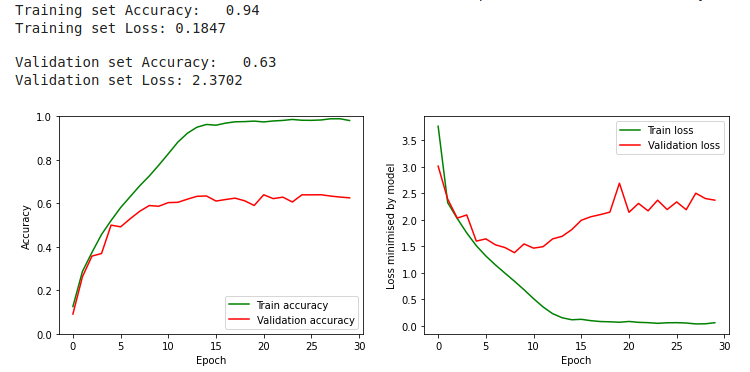
Intermediate model 4 with (64, 256, 512)



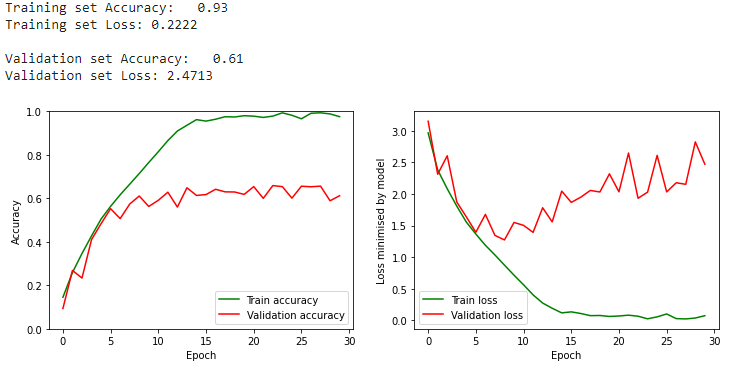
Intermediate model 4 with (64, 128, 512)



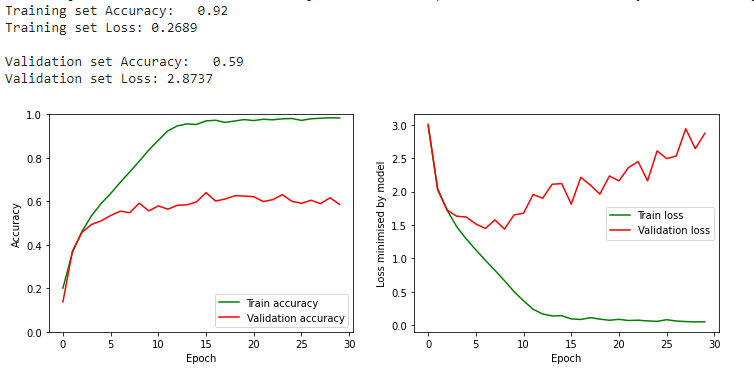
Intermediate model 4 with (64, 128, 1024)



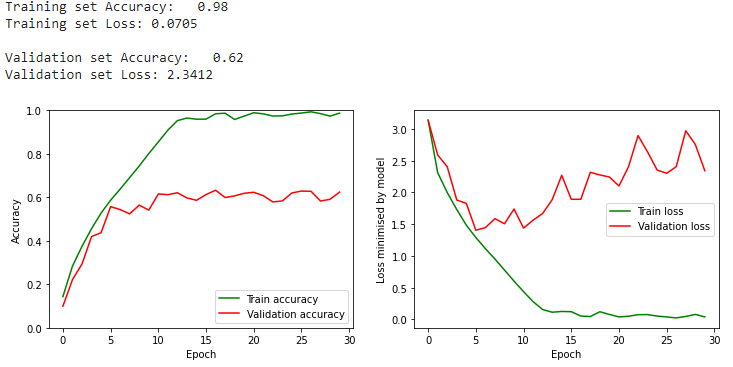
Intermediate model 5 with kernel sizes (5x5)



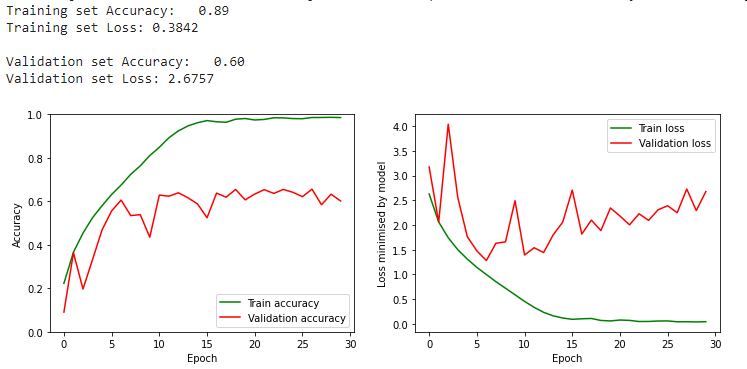
Intermediate model 5 with kernel sizes (2x2)



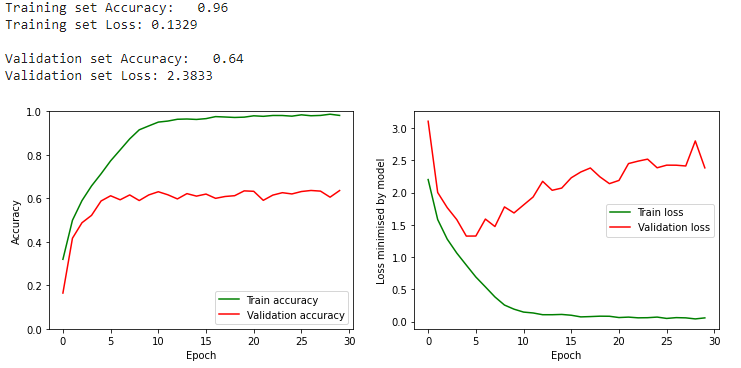
Intermediate model 5 with kernel sizes (5x5) in the first block and (3x3) in the following blocks



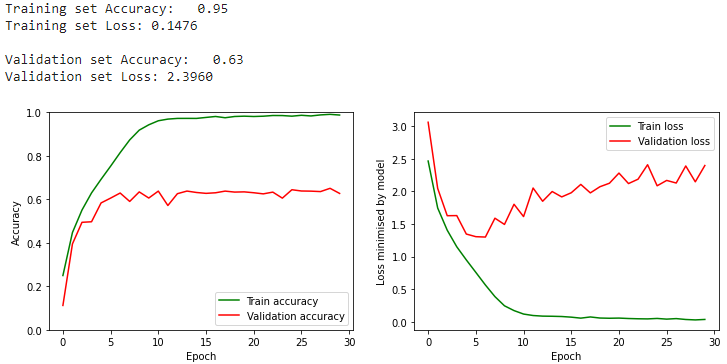
Intermediate model 6 with AveragePooling2D (2x2)



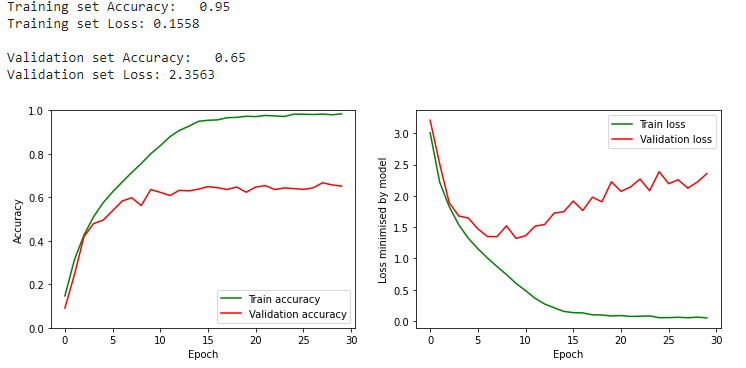
Intermediate model 6 with MaxPooling2D (3x3)



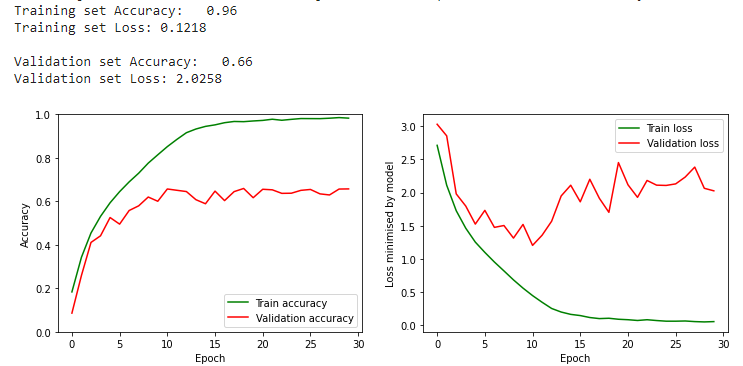
Intermediate model 6 with MaxPooling2D (3x3) in first block, (2x2) in the other blocks



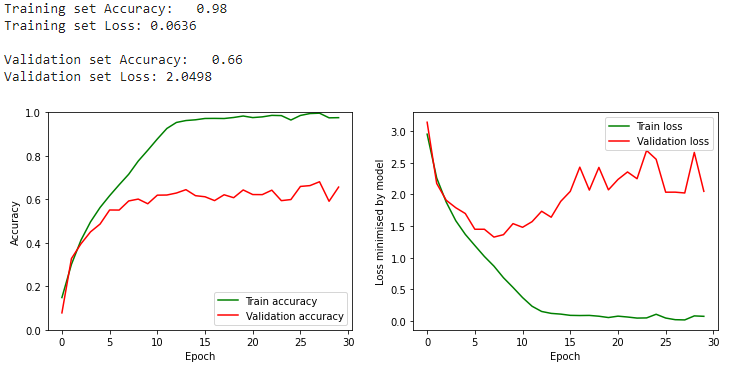
Intermediate model 7 with Dense layers 1024, 512, 20



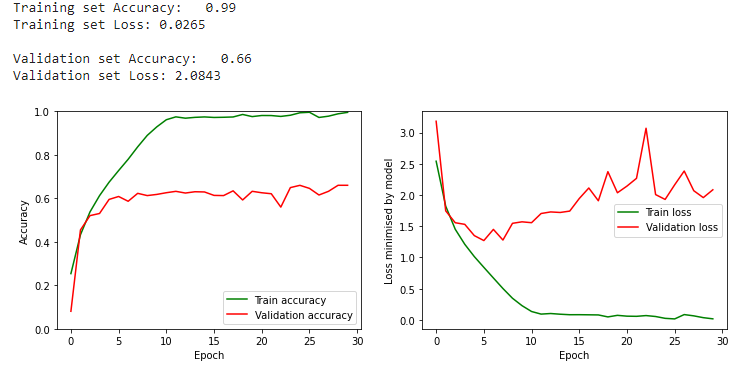
Intermediate model 7 with Dense layers 512, 256, 128, 20



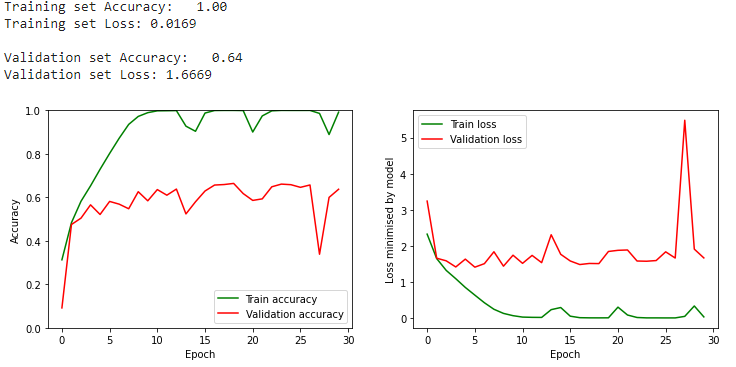
Intermediate model 7 with Dense layers 256, 20



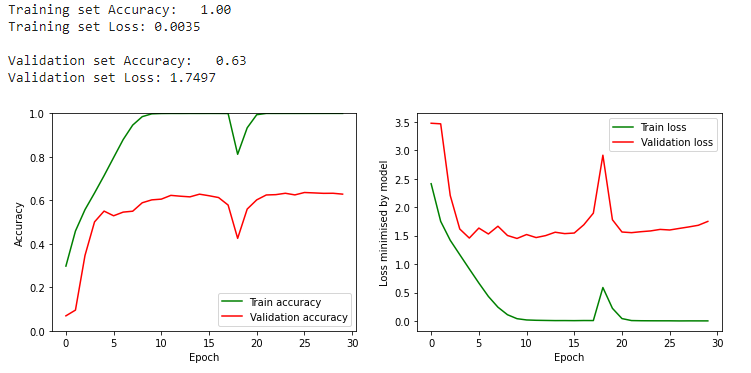
Intermediate model 8 with learning rate 0.0005 and batch size 128



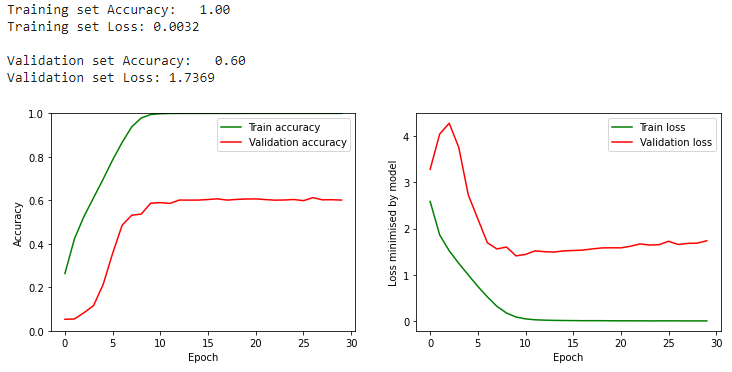
Intermediate model 8 with learning rate 0.00025 and batch size 128



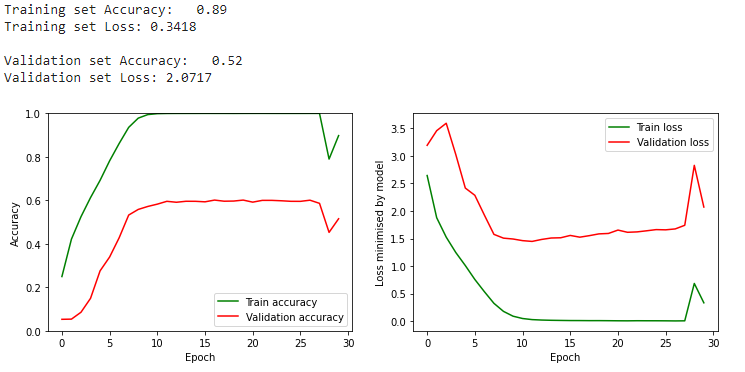
Intermediate model 8 with batch size 256 and learning rate 0.00025



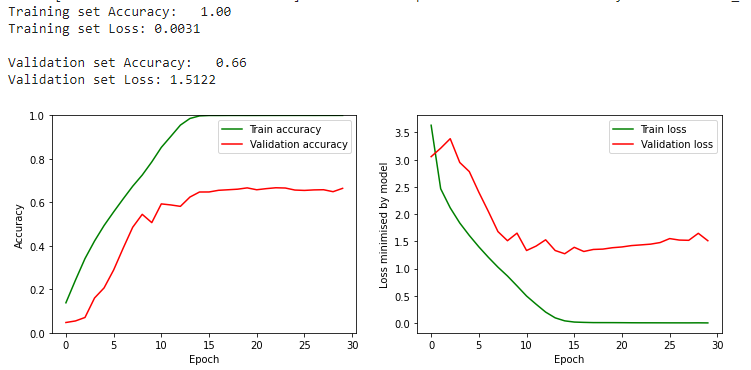
Intermediate model 8 with batch size 512 and learning rate 0.00025



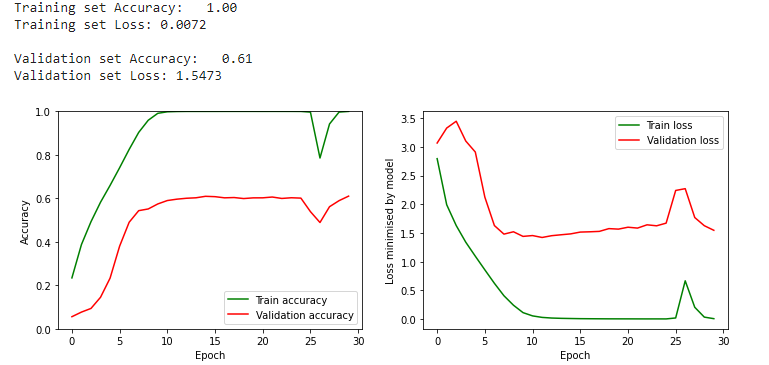
Intermediate model 8 with batch size 512 and learning rate 0.00025 (second try)



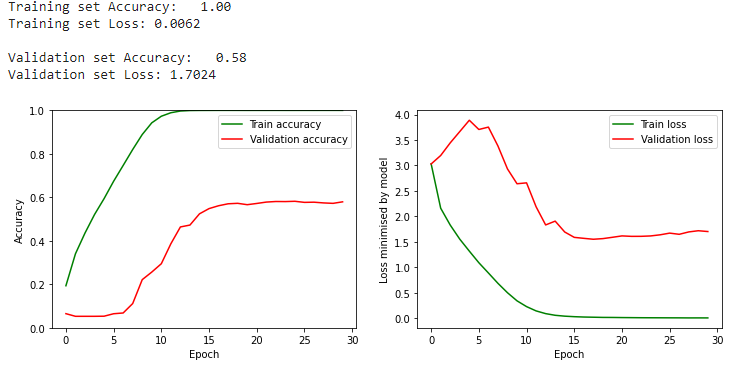
Intermediate model 8 with batch size 512 and learning rate 0.0005



Intermediate model 8 with batch size 512 and learning rate 0.0005 (second try)



Intermediate model 8 with batch size 1024and learning rate 0.00025



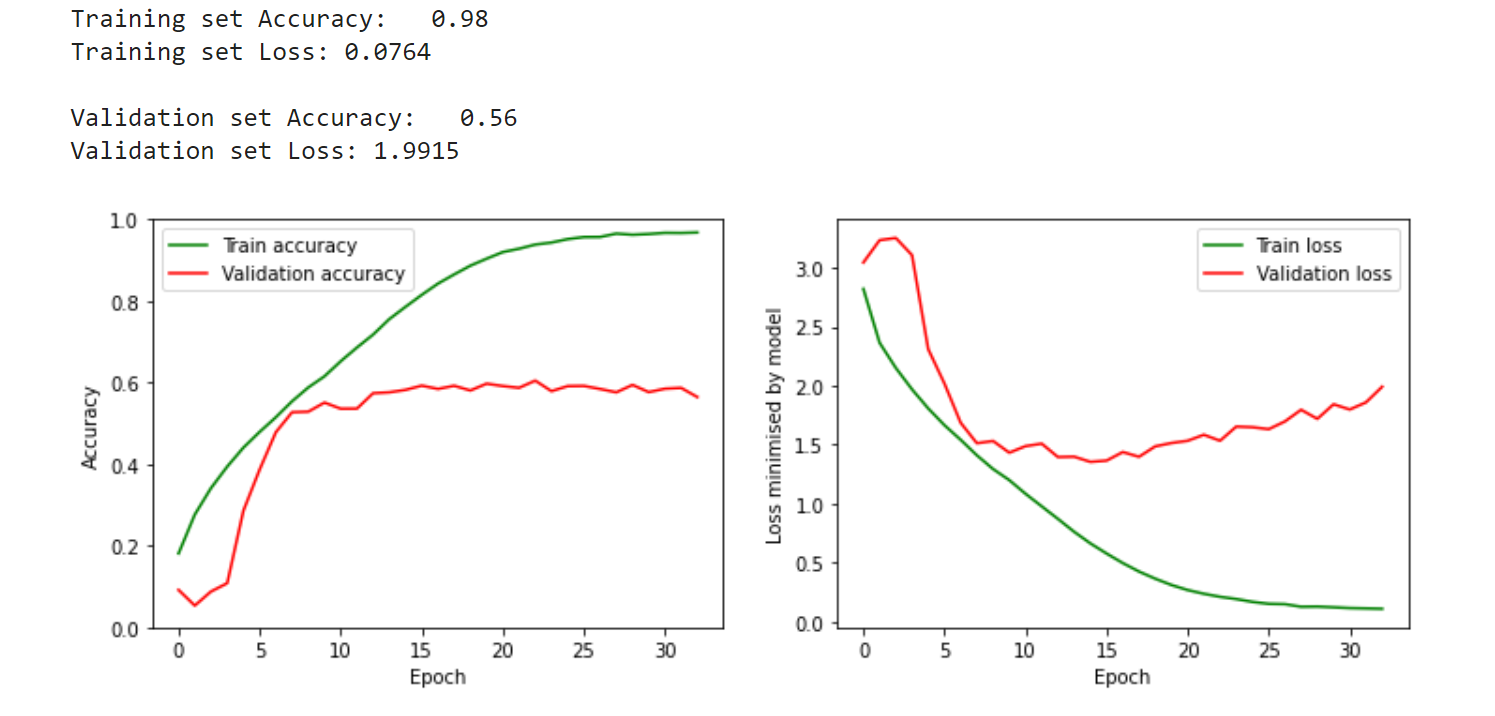
# Regularization

During the regularization phase, we use early stopping with the patience hyperparameter set to 10 and the min\_delta hyperparameter set to 0.00001. When running a certain model, we also make sure that the training is not stopped when the performance is still improving. This means that the training will only be stopped by the early stopping and the number of epochs is increased if 100 epochs is reached.

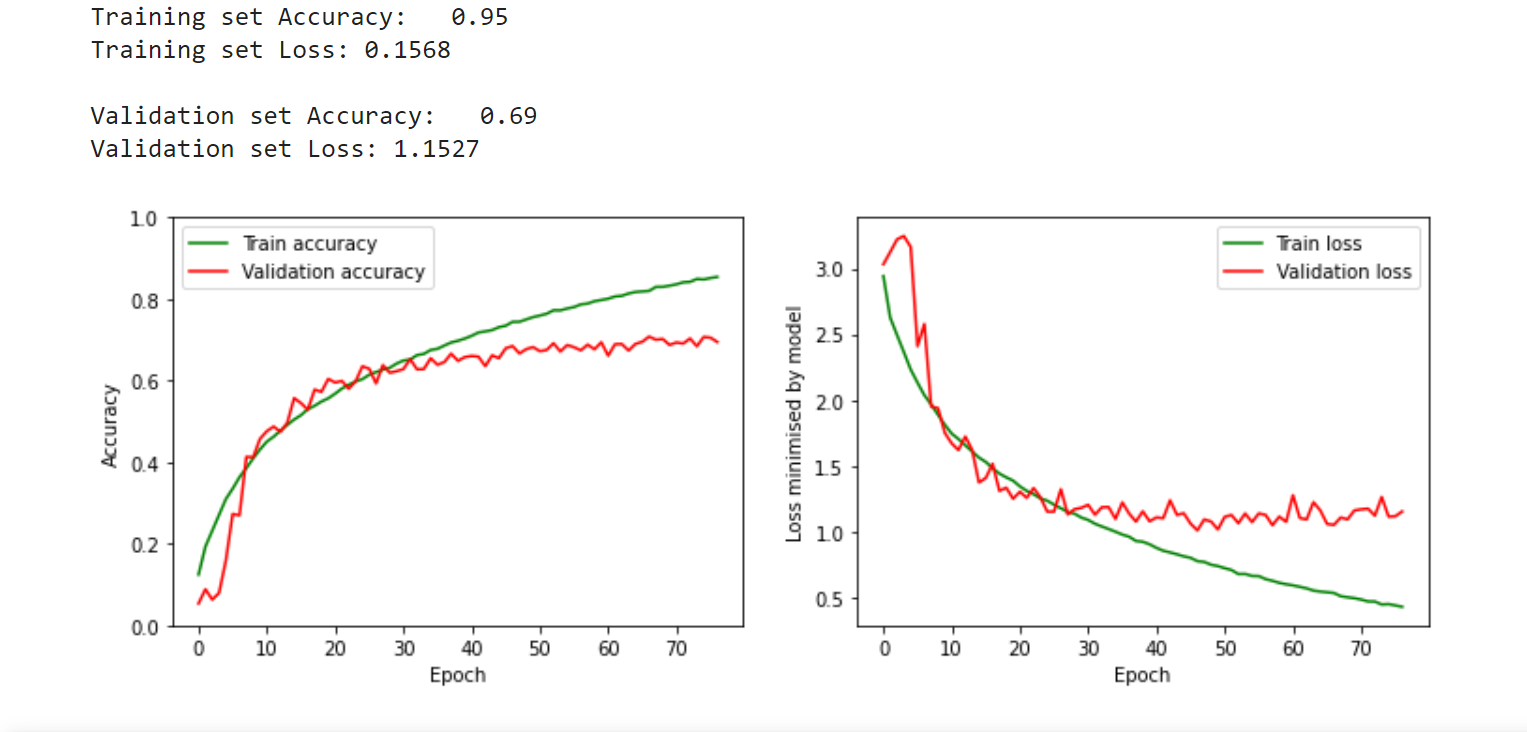
| **Intermediate model** | **Dropout** | **Optimal values for the hyperparameter at the end of this step** | **Train/valid accuracies** | **Notebook version** |
| --- | --- | --- | --- | --- |
| 1 | A single dropout layer (p=0.5 default value) was added to the model at the dense layer. |  | 98%/56% | GA2\_Training\_v11 |
| 2 | In addition to the above layer, three dropout layers were added at each convolution block randomly starting from low dropout (p=0.2) to increasing it as the number of neurons increased subsequently in the layers.  p=0.2 (Conv2D, 64)  p=0.3 (Conv2D, 128)  p=0.4 (Conv2D, 512)  p=0.5 (Dense, 512) |  | 95%/69% | GA2\_Training\_v10 |
| 3 | From the above layers the dropout layer at the dense layer was removed and the dropout layers at convolution layers were kept.  p=0.2 (Conv2D, 64)  p=0.3 (Conv2D, 128)  p=0.4 (Conv2D, 512) |  | 100%/69% | GA2\_Training\_v9 |
| 4 | The dropout layers were kept same for all the 4 dropout layers (p=0.2)  p=0.2 (Conv2D, 64)  p=0.2 (Conv2D, 128)  p=0.2 (Conv2D, 512)  p=0.2 (Dense, 512) |  | 99%/67% | GA2\_Training\_v12 |
| 5 | From the above regularization model 2, the value of the last dropout layer was decreased to 0.4.  p=0.2 (Conv2D, 64)  p=0.3 (Conv2D, 128)  p=0.4 (Conv2D, 512)  p=0.4 (Dense, 512) |  | 97%/71% | GA2\_Trainingv13 |
| 6 | The dropouts were increased from the above model.  p=0.3 (Conv2D, 64)  p=0.4 (Conv2D, 128)  p=0.5 (Conv2D, 512)  p=0.5 (Dense, 512) | Overfitting significantly reduced in the model, however, val accuracy is currently at 71%. | 93%/71% | GA2\_Trainingv14 |
| 7 | The dropouts were increased from the above model.  p=0.3 (Conv2D, 64)  p=0.4 (Conv2D, 128)  p=0.6 (Conv2D, 512)  p=0.6 (Dense, 512) | The overfitting on training data reduces as the last two dropout layers are increased. This will be the best performing model which will be used for adding the next regularization techniques. | 90%/71% | GA2\_Trainingv15 |
| 8 | The dropouts were increased from the above model.  p=0.5 (Conv2D, 64)  p=0.5 (Conv2D, 128)  p=0.6 (Conv2D, 512)  p=0.6 (Dense, 512) | Adding even higher values for the dropout results in a slight decrease in performance which could mean that too high values are used for the dropout. | 82%/69% | GA2\_Trainingv16 |

*Table 2: Tuning process for regularized model with dropout*

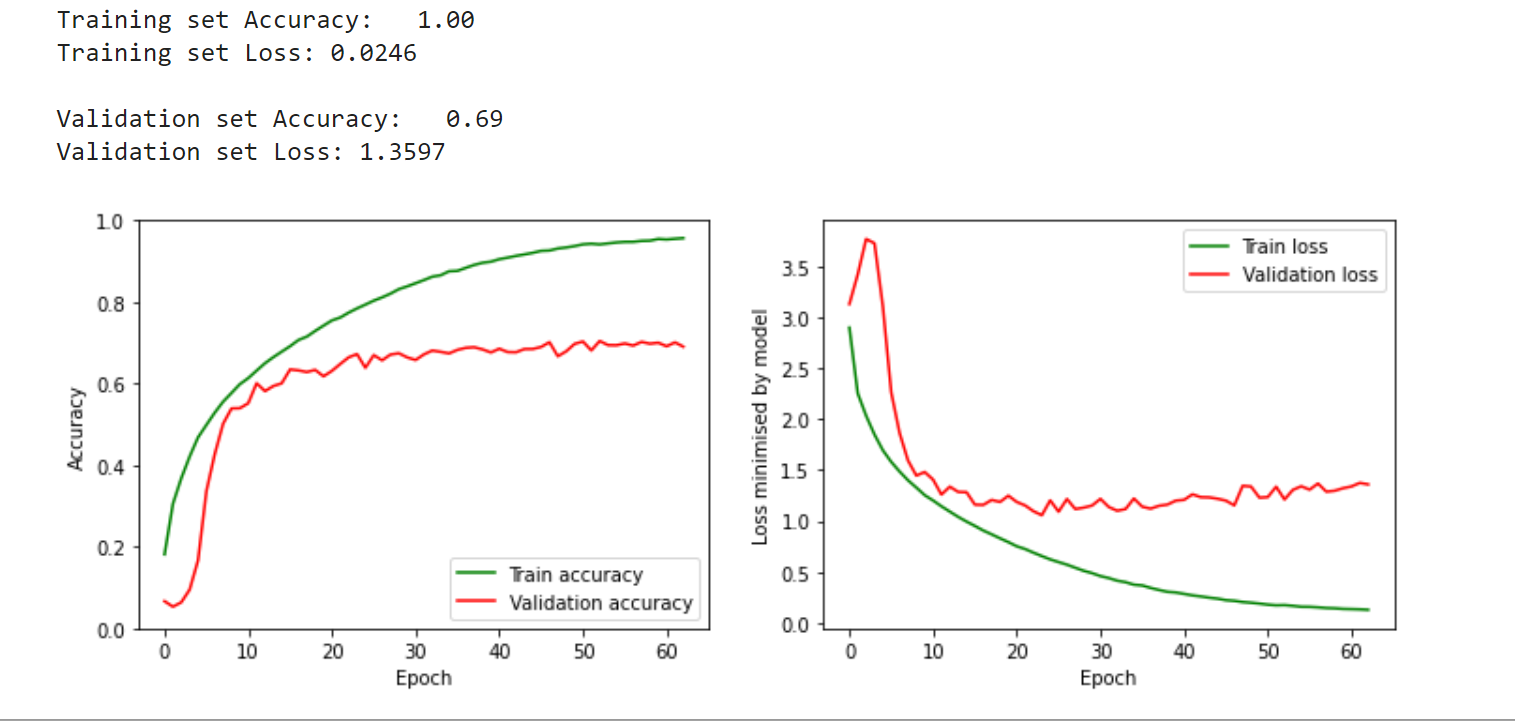
1. Dropout layer at Dense Layer(p=0.5)



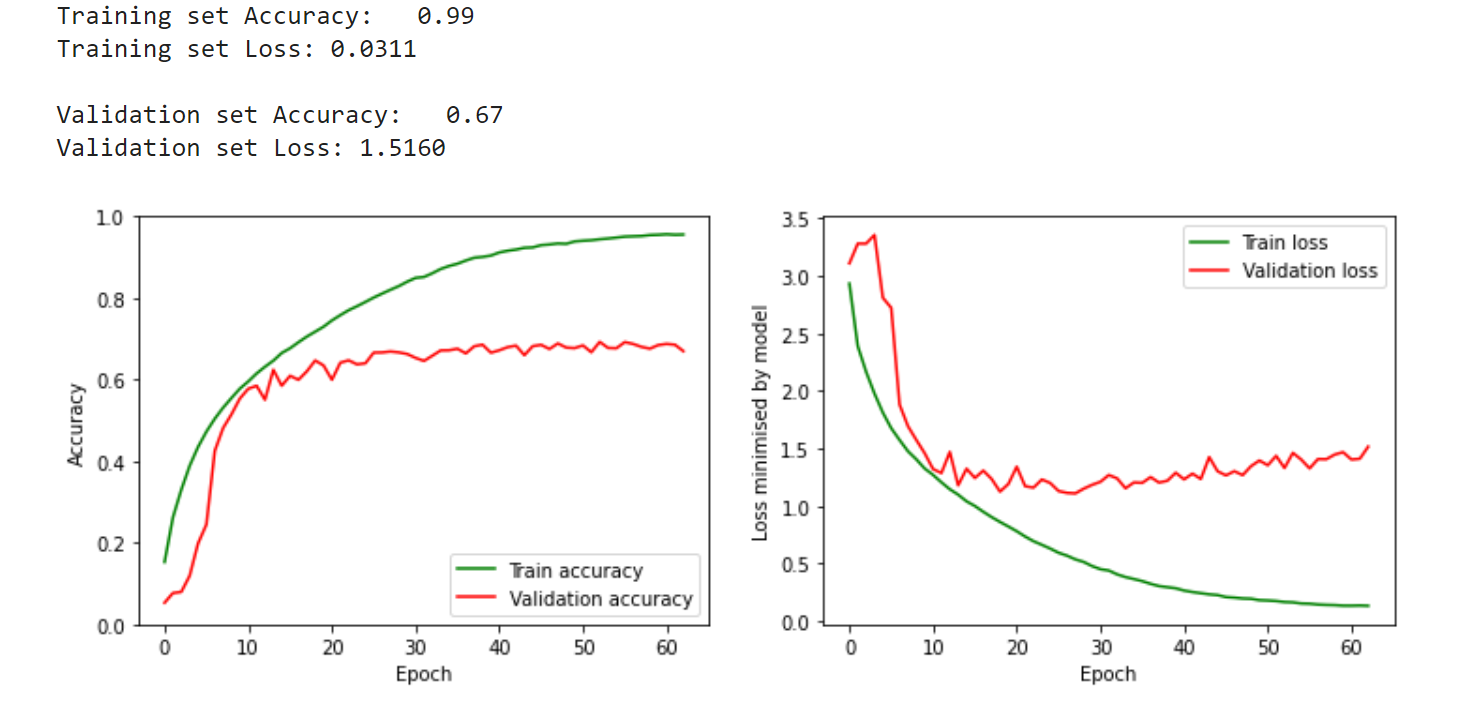
2. dropout layers (p=0.2, p=0.3, p=0.4, p=0.5)



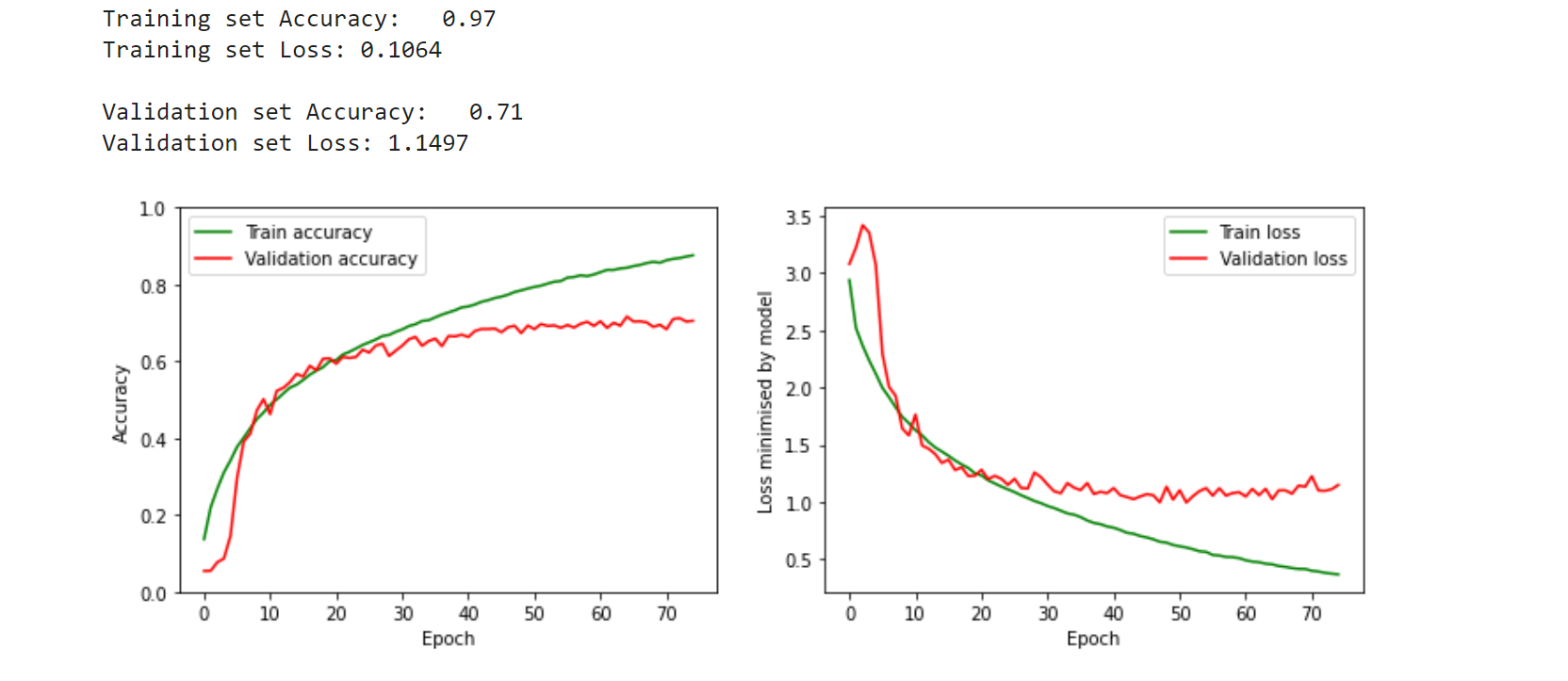
3. dropout layers (p=0.2, p=0.3, p=0.4)



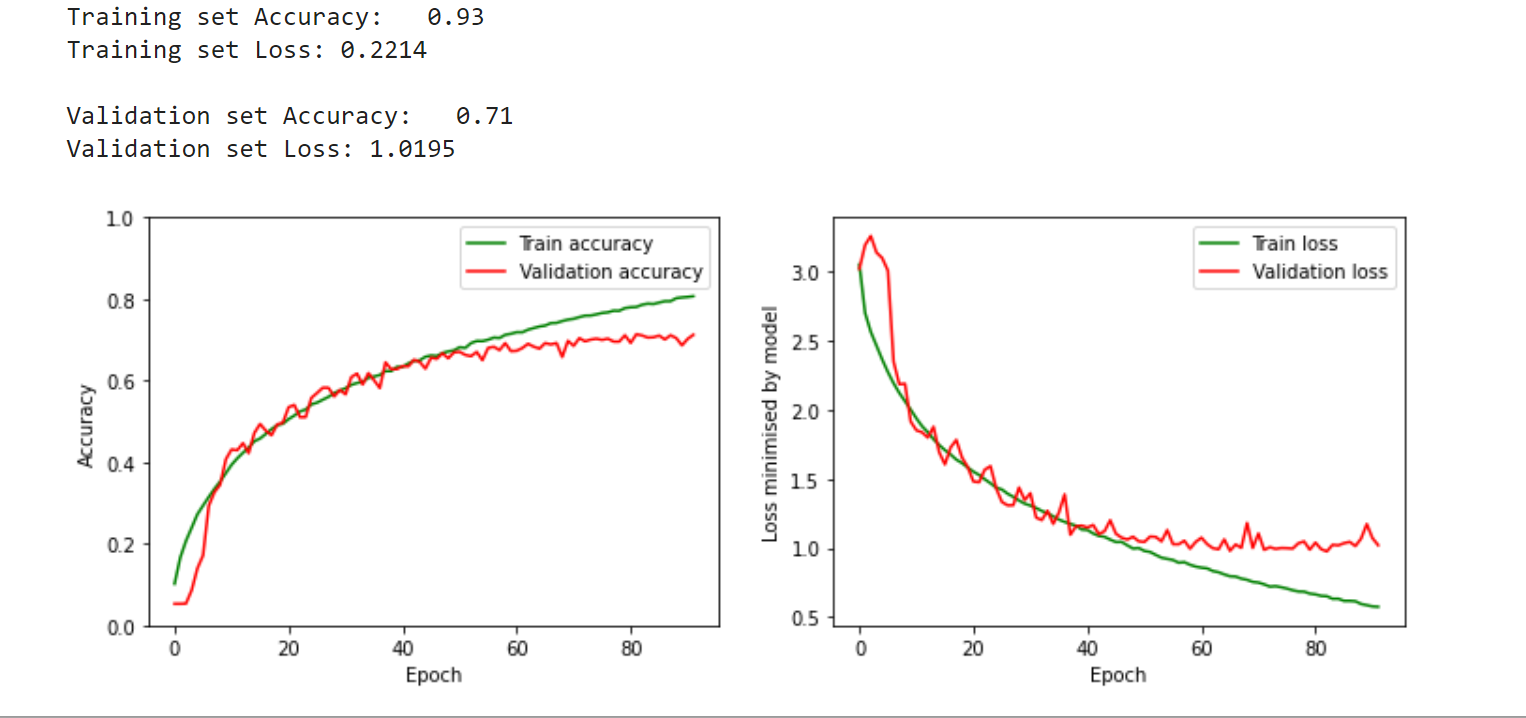
4. dropout layers (p=0.2, p=0.2, p=0.2, p=0.2)



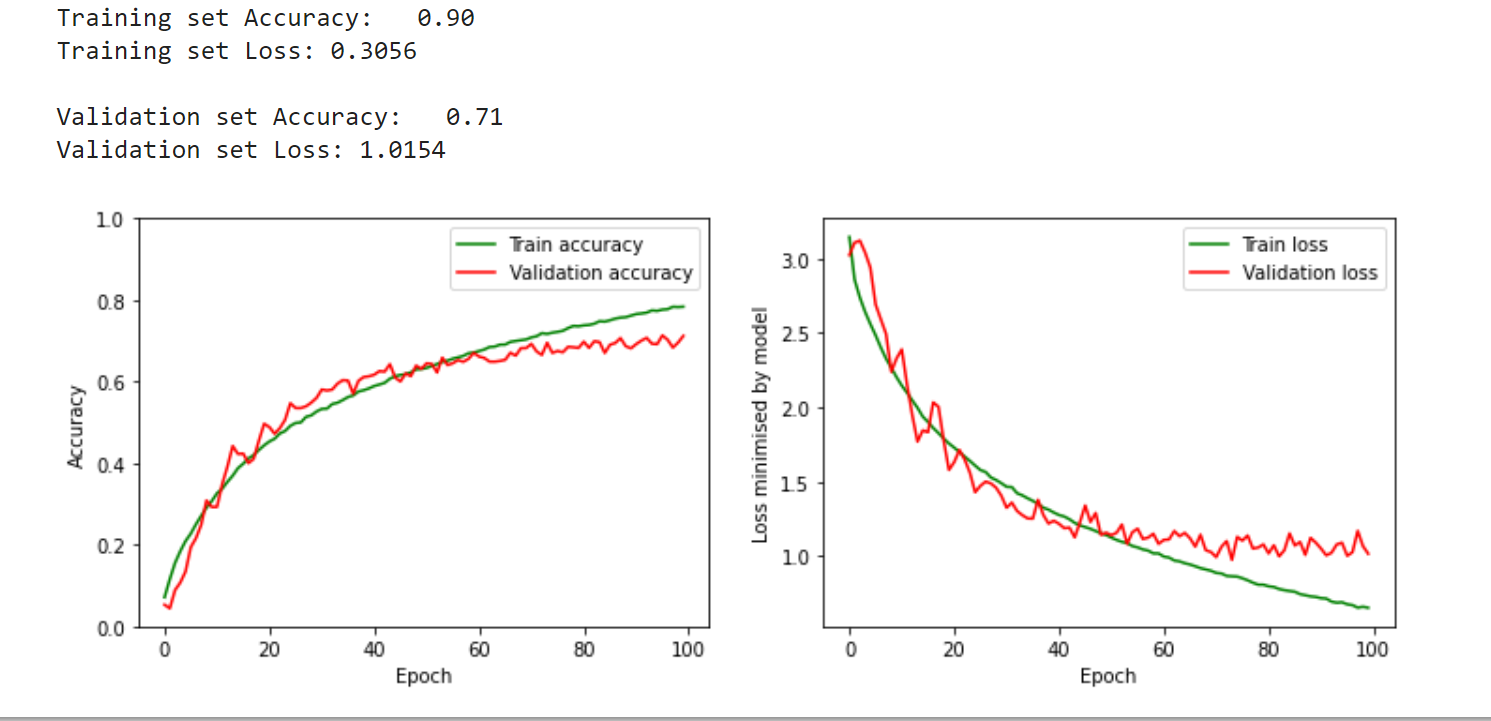
5. dropout layers (p=0.2, p=0.3, p=0.4, p=0.4)



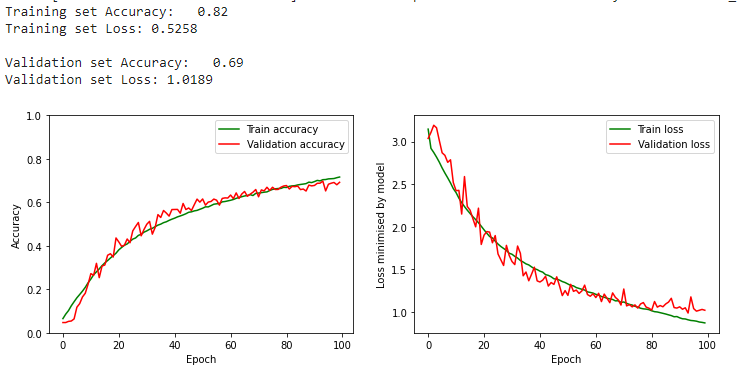
6. dropout layers (p=0.3, p=0.4, p=0.5, p=0.5)



7. dropout layers (p=0.3, p=0.4, p=0.6, p=0.6)



8. dropout layers (p=0.5, p=0.5, p=0.6, p=0.6)

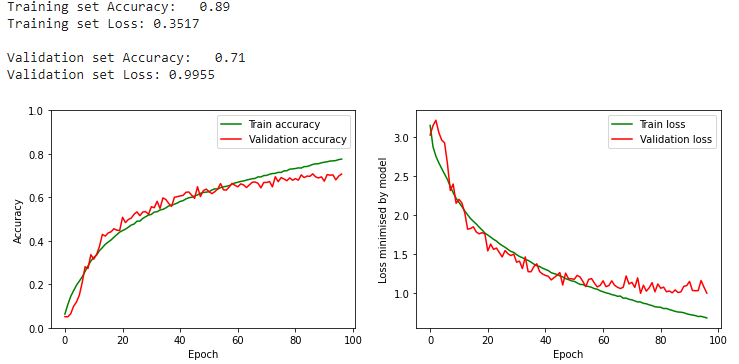


| **Intermediate model** | **Maxnorm** | **Optimal values for the hyperparameter at the end of this step** | **Train/valid accuracies** | **Notebook version** |
| --- | --- | --- | --- | --- |
| 1 | Maxnorm of 2 in each Conv2D and Dense layer (together with the dropout layers) |  | 89%  71% | GA2\_Trainingv17 |
| 2 | Maxnorm of 4 in each of the Conv2D layers and maxnorm of 2 in the Dense layer |  | 90%  71% | GA2\_Trainingv17 |
| 3 | Maxnorm of 1 in each of the Conv2D and Dense layer | Performance with best model when using the best model after applying early stopping | 93%  71% | GA2\_Trainingv17 |
| 4 | Maxnorm of 3 in each of the Conv2D layers and maxnorm of 3 in the Dense layer |  | 88%  70% | GA2\_Trainingv17 |
|  |  |  |  |  |

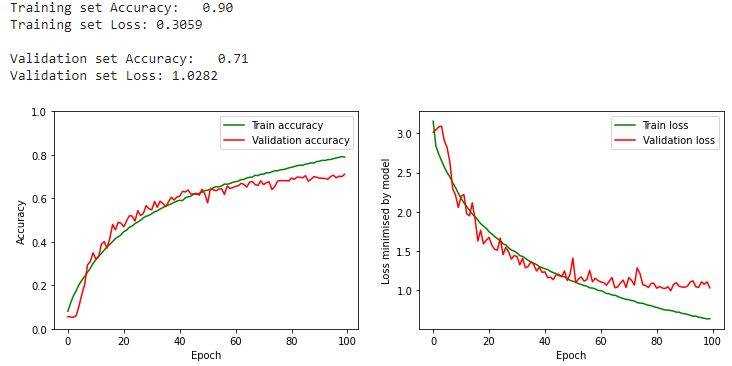
*Table 2: Tuning process for regularized model with Maxnorm*

Typical values for the max norm hyperparameter are on orders of 3 or 4[[3]](#footnote-2).

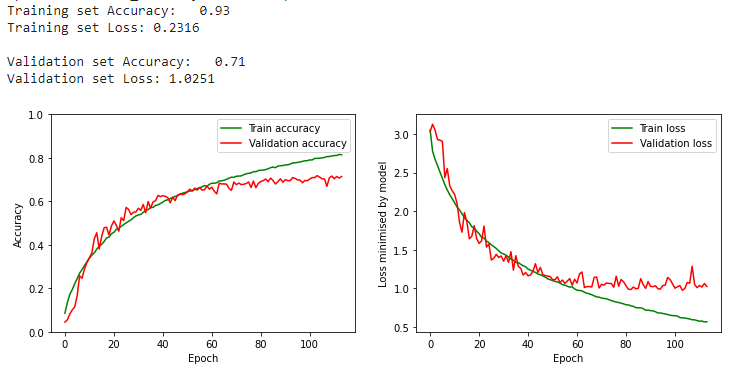
Maxnorm of 2 in each conv2D and dense layer



Maxnorm of 4 in each conv2D and 2 in dense layer



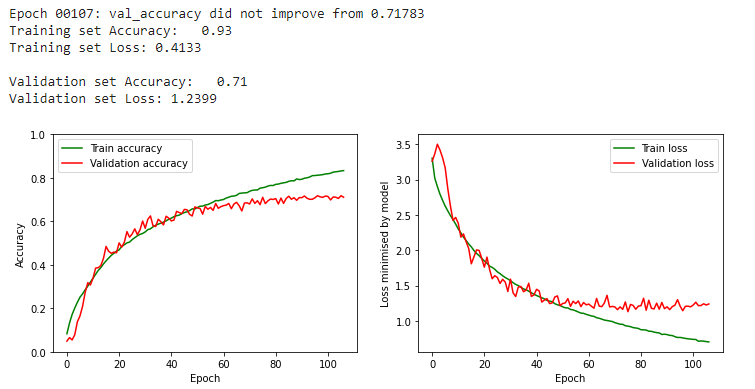
Maxnorm of 1 in each conv2D and dense layer (along with the dropout)



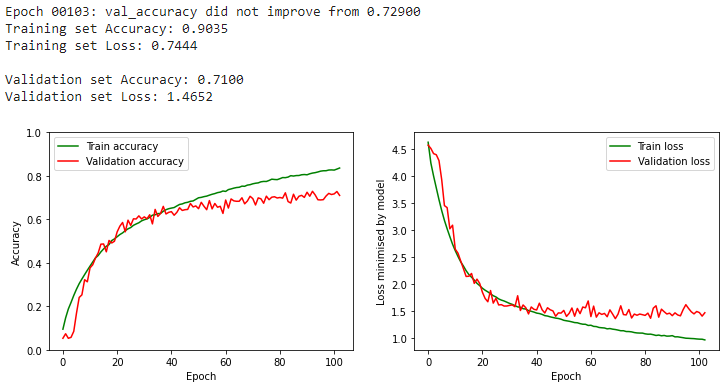
| **Intermediate model** | **L1 and L2** | **Optimal values for the hyperparameter at the end of this step** | **Train/valid accuracies** | **Notebook version** |
| --- | --- | --- | --- | --- |
| 1 | l1 = 1e-6  l2 = 1e-4 |  | 93%  71% | GA2\_Trainingv18 |
| 2 | l1 = 1e-6  l2 = 1e-3 |  | 87%  70% | GA2\_Trainingv18 |
| 3 | l1 = 1e-5  l2 = 1e-2 |  | 65%  55% |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |

*Table 3: Tuning process for regularized model with L1 and L2*

L1 = 1e-6, l2 = 1e-4

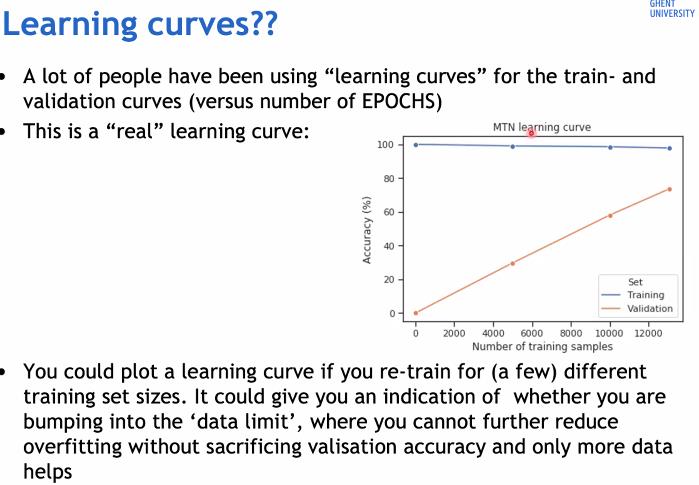


L1 = 1e-6, l2 = 1e-3



# Augmentation

Add table with all the architectures and hyperparameters we tried, along with their results. Also add a similar learning curve which was given in the lecture (see below)



In order to obtain even better results, an approach could be to obtain more data. However, when data is rather limited, another idea is to generate synthetic data which can be done through data augmentation. When using augmentation techniques, new data (synthetic pictures) is generated ‘on the fly’ when the training batches are read. Before the training samples are fed to the model, the sample is augmented by performing several transformations[[4]](#footnote-3). Transformations that were considered for this assignment relate to the following ones:

* **rotation\_range** which randomly rotates images in a predefined range (0 to 180 degrees)
* **width\_shift\_range** randomly shifts the images horizontally based on a fraction of the total picture width
* **height\_shift\_range** randomly shifts the images horizontally based on a fraction of the total picture height
* **horizontal\_flip** randomly flips the image horizontally
* **vertical\_flip** randomly flips the image vertically
* **fill\_mode** tells what happens with ‘new’ pixels that occur due to a certain transformation (for example when the image is shifted to the right, it means that new pixels are created on the left of the image and a technique is needed to fill these pixels)
* **shear\_range** the shear angle which is applied in counter-clockwise direction (in degrees)
* **zoom\_range** defines the range for the random zoom

As mentioned in the notebook, an example image was taken and different values for the augmentation technique were used such that the new created images still looked realistic. The values that were used and gave us reasonable data are as follows:

# Final model result

1. <https://arxiv.org/pdf/1409.1556.pdf> [↑](#footnote-ref-0)
2. <https://neurohive.io/en/popular-networks/vgg16/> [↑](#footnote-ref-1)
3. <https://cs231n.github.io/neural-networks-2/#reg> [↑](#footnote-ref-2)
4. <https://www.tensorflow.org/api_docs/python/tf/keras/preprocessing/image/ImageDataGenerator>

   <https://machinelearningmastery.com/how-to-configure-image-data-augmentation-when-training-deep-learning-neural-networks/> [↑](#footnote-ref-3)