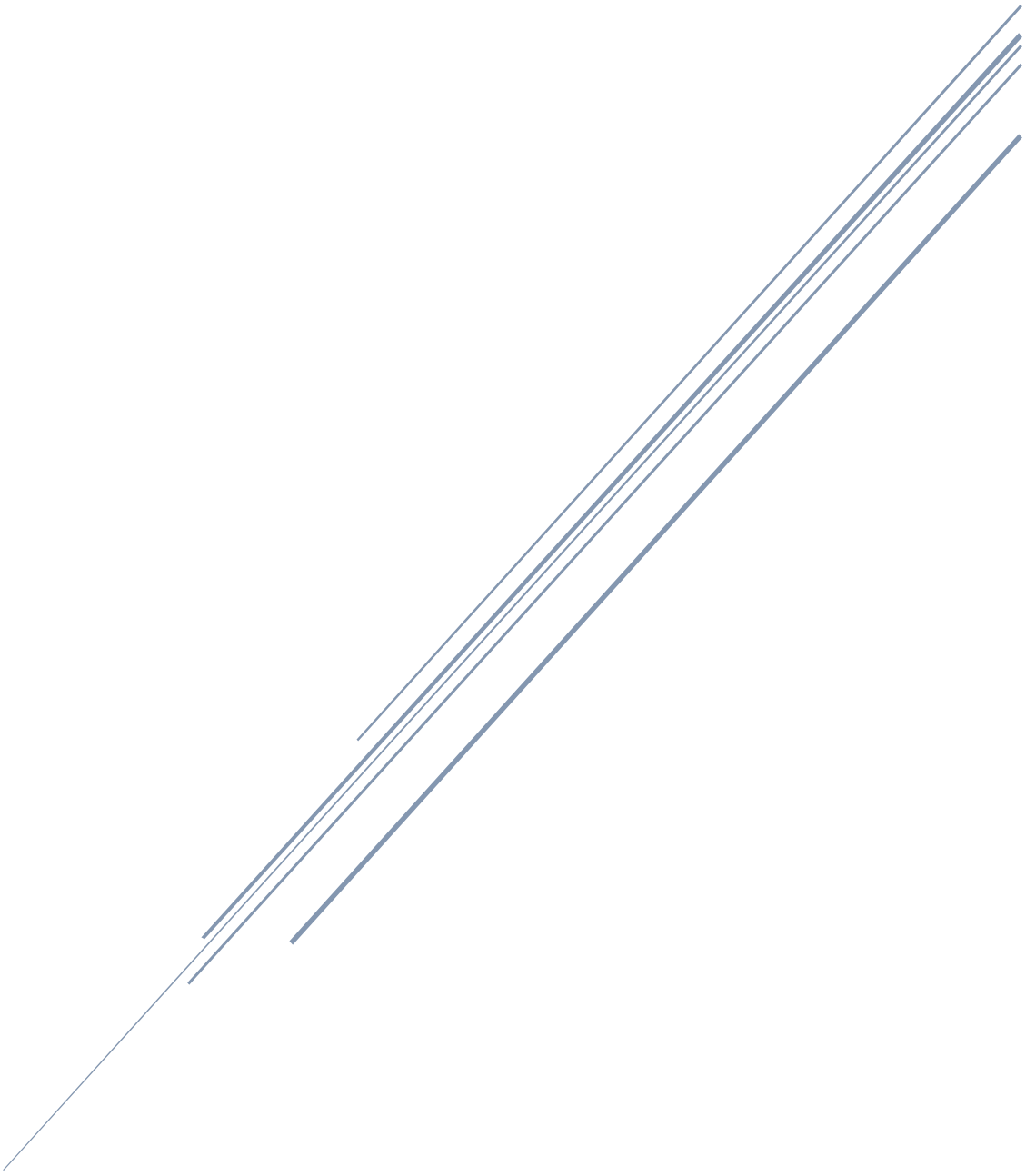
Image Classification with Small Data

Luca Nunziante



Università degli Studi di Roma "La Sapienza"

Problem overview

Throughout this report are discussed solutions to an image classification problem using a small amount of data. The dataset of images is ciFAIR-10 which consists of 10 classes: the training set is made of 50 training images per class and the test set of 1000 images per class. Out of the 50 images per class that can be used during training, 20 are used solely for validation and the remaining for training. The validation set is the one on which all the plots that will be shown are based on, while the test set is the set on which the accuracies that are reported are computed on.

The models presented in this report are Convolutional Neural Networks based on SqueezeNet, a network that has been created with the aim of reducing the parameters of a CNN while maintaining competitive accuracy [1] . However, some changes have been made to the basic structure for various reasons that are discussed below. The 2 solutions that will be compared share the same architecture, while they vary in the value of the hyperparameters used for training.

There are 2 versions of this network and the one that has been used, with the modifications that will be presented, is the 1.1: SqueezeNet 1.1 has 2.4x less computation and slightly fewer parameters than SqueezeNet 1.0, without sacrificing accuracy [2]

Data Preprocessing

First, the data is normalized: for each channel – 3 in this case – of the input image the mean μ and standard deviation σ are computed and the value x of each pixel is modified as follows.

$$z = \frac{x - \mu}{\sigma}$$

Then, the following transformations are allowed:

- Translation of the image of at most 4 pixels
- Random horizontal flipping

This preprocessing is used for each training that will be discussed through this report.

Architecture used

To justify the customised architecture used in the two models that will be discussed at a later stage, a few trials and their outcomes are presented in this section.

Using the basic architecture, I have found that the error function got stuck in a local minimum achieving an accuracy of 10% (Figure 1). To overcome this issue, a Batch Normalization layer has been added both at the end of each Fire module and after the first convolutional layer: these changes have helped the model in avoiding the problem but still achieving an accuracy of just about 37% (Figure 2).

Seeking for improvements, it has been noted that the first convolutional layer of the network originally had a kernel size of 3 and a stride of 2, downsampling the image as soon as it entered: however, in the paper about SqueezeNet [1] – and in other sources [3] – it is advised to keep the feature maps large in the first layers to not *wash away* the features that may be present at the borders of the image. Indeed, using a stride of 1 and a padding 'same' has proved to be a better choice giving an increase of 9% in the accuracy, and for this reason it has been used (Figure 3).

Finally, the network appeared to be deep and not very wide. So, to overturn this characteristic, the number of Fire modules has been decreased and the number of filters increased: that has caused the accuracy to increase at the expense of a higher number of parameters (Figure 4).

Clearly, these changes have been done in an incremental fashion: in other words, the padding has been added keeping the batch normalization layers, and the architecture has been widened keeping the normalization and the padding. These tests are summarised in Table 1.

Model	Number of epochs	Accuracy
Basic Squeezenet	150	10%
BatchNorm added	800	37.39%
Padding added	800	46.67%
My architecture	300	48.17%

Table 1. These are the models that have been tried. The number of epochs is of only 150 for the first model since it got stuck at 10% after just a few as shown in Figure 1 below so it was pointless to keep going.

In Figure 1 to 4 we can see the plot of the accuracy, in blue computed on the training set and in orange the one computed on the validation set. All the trainings of these models have been done with the same algorithm and parameters, namely Stochastic Gradient Descent (SGD) with a learning rate of 0.00455, a weight decay factor of 0.00529 and a momentum of 0.9.

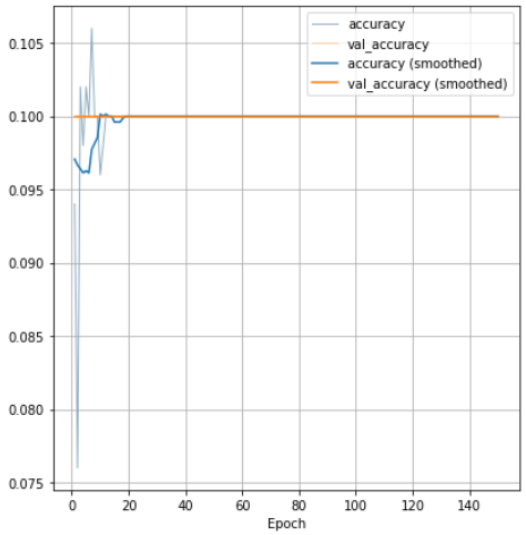


Figure 1. Accuracy plot for the basic architecture.

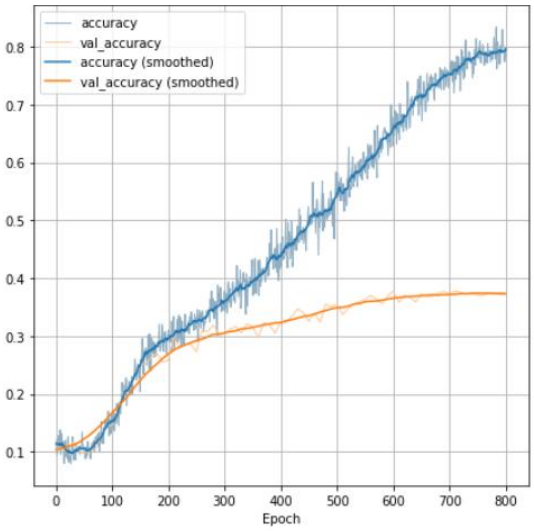


Figure 2. Accuracy plot after adding Batch Normalization.

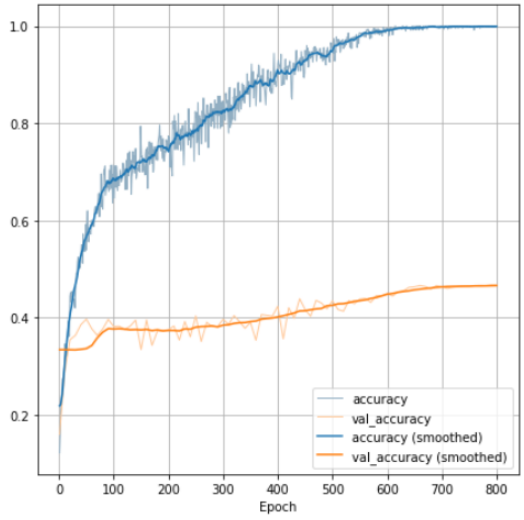


Figure 3. Accuracy plot after adding the padding at the first Conv layer.

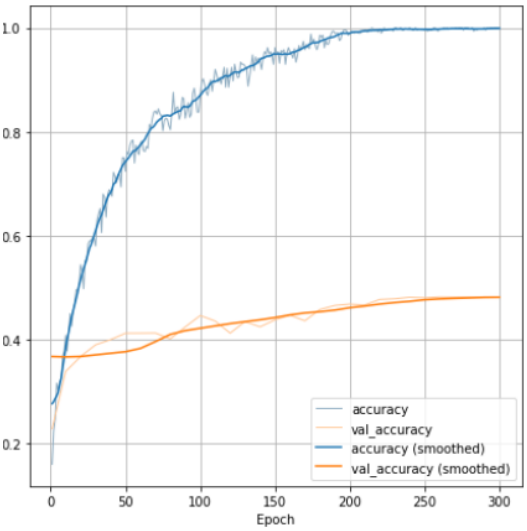


Figure 4. Accuracy plot of my architecture.

My architecture, the one on which more will be tried in the next section, is the one below where the shapes are computed considering the input shape of the ciFAIR-10 dataset, namely images of size 32x32 with 3 colour channels. In red the layers that I have added and in blue the convolutional layer where stride and padding have been modified.

```
----------------------------------------------------------------
        Layer (type)               Output Shape         Param #
================================================================
            Conv2d-1          [-1, 64, 32, 32]           1,792
              ReLU-2          [-1, 64, 32, 32]               0
       BatchNorm2d-3          [-1, 64, 32, 32]             128
         MaxPool2d-4          [-1, 64, 16, 16]               0
            Conv2d-5          [-1, 16, 16, 16]           1,040
              ReLU-6          [-1, 16, 16, 16]               0
            Conv2d-7          [-1, 64, 16, 16]           1,088
              ReLU-8          [-1, 64, 16, 16]               0
            Conv2d-9          [-1, 64, 16, 16]           9,280
             ReLU-10          [-1, 64, 16, 16]               0
      BatchNorm2d-11         [-1, 128, 16, 16]             256
             Fire-12         [-1, 128, 16, 16]               0
        MaxPool2d-13           [-1, 128, 8, 8]               0
           Conv2d-14            [-1, 32, 8, 8]           4,128
             ReLU-15            [-1, 32, 8, 8]               0
           Conv2d-16           [-1, 128, 8, 8]           4,224
             ReLU-17           [-1, 128, 8, 8]               0
           Conv2d-18           [-1, 128, 8, 8]          36,992
             ReLU-19           [-1, 128, 8, 8]               0
      BatchNorm2d-20           [-1, 256, 8, 8]             512
             Fire-21           [-1, 256, 8, 8]               0
        MaxPool2d-22           [-1, 256, 4, 4]               0
           Conv2d-23            [-1, 64, 4, 4]          16,448
             ReLU-24            [-1, 64, 4, 4]               0
           Conv2d-25           [-1, 256, 4, 4]          16,640
             ReLU-26           [-1, 256, 4, 4]               0
           Conv2d-27           [-1, 256, 4, 4]         147,712
             ReLU-28           [-1, 256, 4, 4]               0
      BatchNorm2d-29           [-1, 512, 4, 4]           1,024
             Fire-30           [-1, 512, 4, 4]               0
           Conv2d-31            [-1, 80, 4, 4]          41,040
             ReLU-32            [-1, 80, 4, 4]               0
           Conv2d-33           [-1, 512, 4, 4]          41,472
             ReLU-34           [-1, 512, 4, 4]               0
           Conv2d-35           [-1, 512, 4, 4]         369,152
             ReLU-36           [-1, 512, 4, 4]               0
      BatchNorm2d-37          [-1, 1024, 4, 4]           2,048
             Fire-38          [-1, 1024, 4, 4]               0
           Conv2d-39            [-1, 96, 4, 4]          98,400
             ReLU-40            [-1, 96, 4, 4]               0
           Conv2d-41          [-1, 1024, 4, 4]          99,328
             ReLU-42          [-1, 1024, 4, 4]               0
           Conv2d-43          [-1, 1024, 4, 4]         885,760
             ReLU-44          [-1, 1024, 4, 4]               0
      BatchNorm2d-45          [-1, 2048, 4, 4]           4,096
             Fire-46          [-1, 2048, 4, 4]               0
          Dropout-47          [-1, 2048, 4, 4]               0
           Conv2d-48            [-1, 10, 4, 4]          20,490
             ReLU-49            [-1, 10, 4, 4]               0
AdaptiveAvgPool2d-50            [-1, 10, 1, 1]               0
================================================================
Total params: 1,803,050
Trainable params: 1,803,050
Non-trainable params: 0
```

The layers ‘Fire-N’ are not actually layers but just denote the ending of a Fire module.

Since the purpose of SqueezeNet is to keep the number of parameters low, it is worth noticing that even though the tweaks resulted in an increase of 50% in the number of parameters with respect to the basic structure, this number is still relatively low. Indeed, the training time needed for 800 epochs is of about 30 minutes on Google Colaboratory and this has allowed to make several tests to tune the hyperparameters. More on this matter is discussed below.

Comparison between training configurations

The algorithm used to train all the models is SGD, the batch size is always 10, but the hyperparameters – namely the learning rate, weight decay factor and the momentum – have been tuned based on the performance of the resulting models. In addition, also the SGD variant with the Nesterov momentum has been tried. The training configurations that will be compared are the one summarised in Table 2.

	Algorithm	Learning rate	Weight decay	Momentum	Accuracy [%]
1st config	Basic SGD	0.00455	0.00529	0.9	48.84
2nd config	SGD w/ Nesterov	0.0019	0.027	0.9	52.30

Table 2. Training configurations for the discussed models.

The first configuration is the one on which the tests discussed in the previous section have been made to find the right architecture, and the plots for the validation loss and accuracy are shown in Figure 5. The accuracy reported here is slightly different from the one reported in Table 1 since the number of training epochs is different.

On the other hand, the values chosen for the second configuration are the result of tens of trials. These tests have shown that the most impactful changes on the accuracy occurred when changing the learning rate and weight decay factor: in fact, the same configuration of the second one but with the momentum equal to 0.95 has an accuracy of 50.07%, while with other choices for the learning rate and the weight decay factor the accuracy has always been lower than 49%.

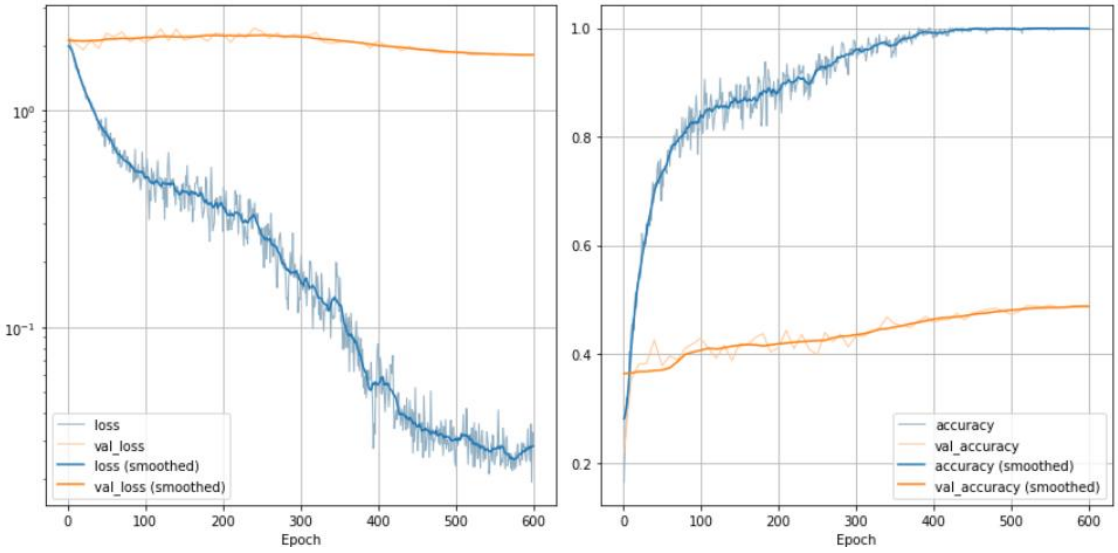


Figure 5. Loss and accuracy plots for the model trained with the first choice of hyperparameters.

In Figure 5 and 7 we can see that the models have been trained for 600 epochs: this choice was done based on the results shown in Figure 6. The loss and accuracy plots represented in this Figure span over an interval of 650 epochs, and at the end the model appears to be in the overfitting region since the validation accuracy is decreasing and the loss increasing, and so the training needs to be stopped before. Stopping the training at 600 epochs has shown to be the right choice for both models.

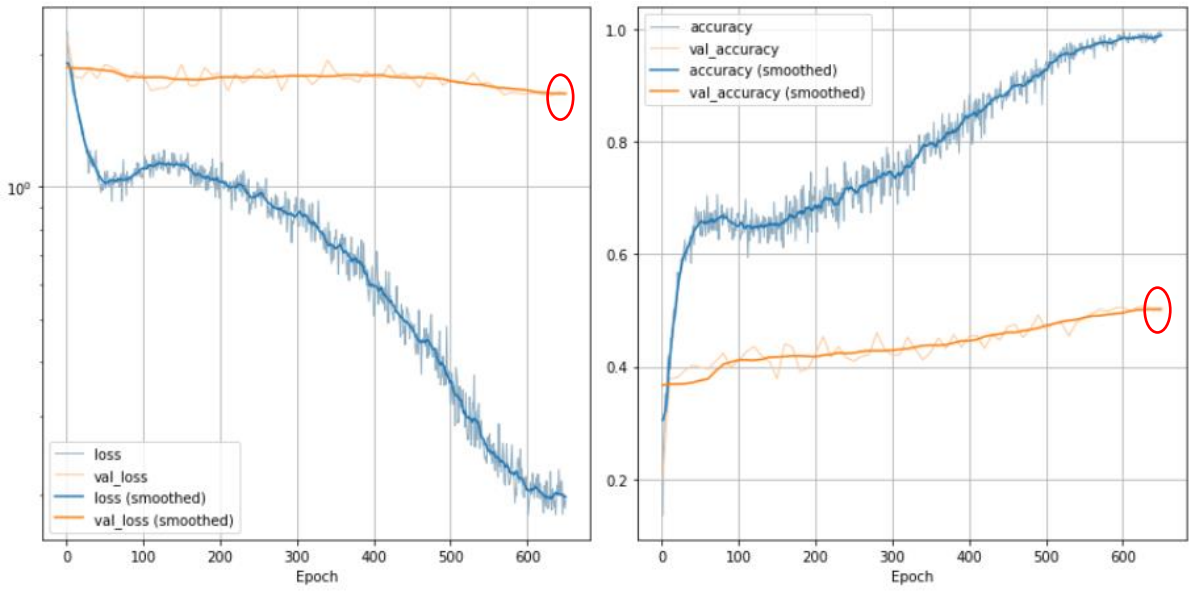


Figure 6. Plots for the second configuration with 650 training epochs. Overfitting region is marked with a red circle.

Finally, in Figure 7 the plots for the best model are shown.

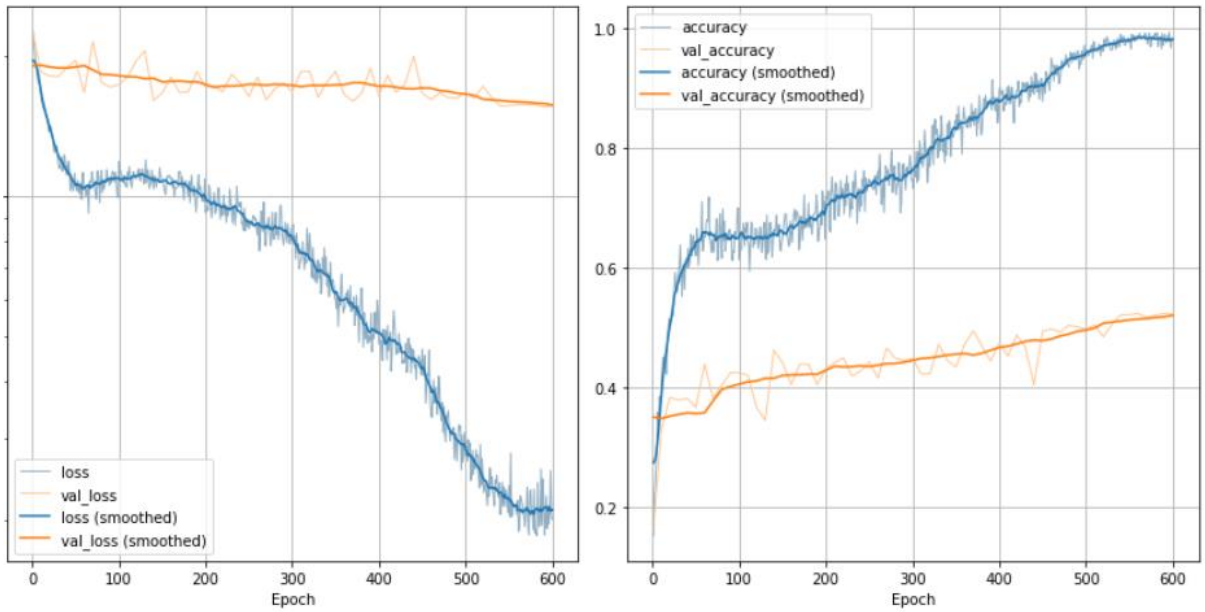


Figure 7. Plots for the second configuration with 600 training epochs.

Conclusion

Throughout all the tests, the Nesterov variant has turned out to be always slightly better. However, as it has been said, the choice of the right learning rate and weight decay has been vital to improve the accuracy, along with finding the right architecture. In addition, it is worth mentioning that also other training algorithms have been tried, such as Adagrad and Adam, but the test phase showed poorer results and so they have not been discussed in this report.

One thing that has been at the core of everything that has been done, even if it has been overlooked previously, is the training time: the fact that a model could be trained in about 20 minutes for 600 epochs has been critical. This factor has been exploited and about 20 trainings have been done to find the right value of the hyperparameters and decide between the training algorithms – namely SGD with or without the Nesterov variant, Adagrad and Adam. On top of that, a little less than 10 trainings were needed to find the right architecture: other than the architectures presented in the first section, also other architectures have been tried but were less powerful – decreasing the size of the kernels in the pooling and in the convolutional layers, and the addition of more dropout layers are some things that have been experimented.

References

[1] Iandola, F. N., Han, S., Moskewicz, M. W., Ashraf, K., Dally, W. J., and Keutzer, K., "SqueezeNet: AlexNet-level accuracy with 50x fewer parameters and <0.5MB model size", 2016.

[2] https://pytorch.org/vision/stable/models.html

[3] https://cs231n.github.io/convolutional-networks