Machine Learning  
Second Homework

Luca Tirel 1702631

*Abstract* — The aim of this report is to discuss the problem of image classification on a reduced version of the dataset ciFAIR-10, using some well-known “small” convolutional neural networks, trying to improve the results obtained in the classification problem.

*Index Terms* — Machine Learning, Image classification, Neural Networks, Convolutional Neural Networks, ResNet, Wide ResNet, Stochastic Gradient Descent

# Introduction

T

HE aim of this report is to provide an analysis of the already defined problem of image classification and trying to improve the research on it by obtaining a large and heterogeneous ensemble, varying some of the parameters, optimizers and architectures.

This image classification problem starts from a reduced version of the ciFAIR-10 dataset, a balanced dataset with 60.000 training images of dimension 32x32x3, 10 output classes. The images represent common objects like different kinds of vehicles and animals.

The training phase is done on a very small subset of the pictures, using only 50 of them for each class.

The testing of the model is performed on a set of 1000 images per classes.

In the report will we use an already defined structure available on GitHub [1] in which the images are preprocessed with some standard operations like scaling, rotating, zooming, and flipping.

The main reason behind this study is that small-scale convolutional networks provide good accuracy in small-data settings and the evaluation of the best tuning of hyper-parameters, that is expensive in terms of computational cost and a critical factor to ensure highest performance, can be achieved better with a crowdsourcing approach.

In the next picture is shown a small sample of the ciFAIR-10 dataset.

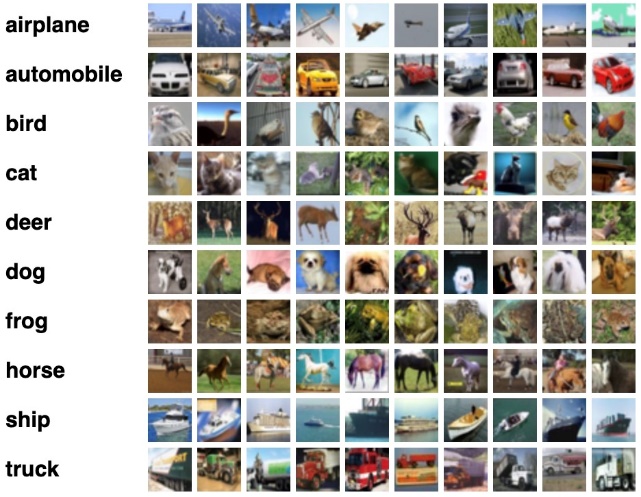


Fig.1 ciFAIR-10 Dataset

# Convolutional Neural Networks

The CNN are similar to the standard ones: they are made by layers containing neurons that have learnable weights and biases. After the dot products are performed, like in standard networks, some non-linear transformation is performed after each of the layers.

The convolutional neural networks are based on the mathematical concept of convolution between two functions­:

The convolutional operator translates one of the two functions while their product is integrated (or summed in discrete case). The two functions are in this case the input of a convolutional layer of the network, and the kernel choose for that particular layer. Also, the kernel can be moved in multiple dimensions.

The Convolutional N.N. are made to handle inputs with special structure, like audio, images, and videos.

In their definition is fundamental the concept convolution and the definition of the kernel, that can happen in each layer. The kernel can be thought as moving sub-structures that have the purpose of “exploring” the input space and map the features.

Others important concepts are Padding, in which we can shape the output size by extending some vectors, and Pooling stages, in which we apply a filter function on the tensor, sometimes with stride that can reduce the output size.

They introduce subsampling and invariance to local translations.

The parameters of the feature map for the first layer of this example are defined in the following table:

|  |  |
| --- | --- |
| Input width |  |
| Input height |  |
| Input depth |  |
| Kernel width |  |
| Kernel height |  |
| Kernel depth |  |
| Number of kernels |  |
| Padding |  |
| Stride |  |

Each layer based on these parameters, also considering the bias, will produce the following output:

|  |  |
| --- | --- |
| Output width |  |
| Output height |  |
| Output depth |  |

The number of parameters of one of the convolutional layers is the following:

This number is computed for each layer.

Usually, the convolutional neural networks have some fully connected layers in the end, that slightly increase the number of parameters in the model.

This figure shows an example of a CNN:

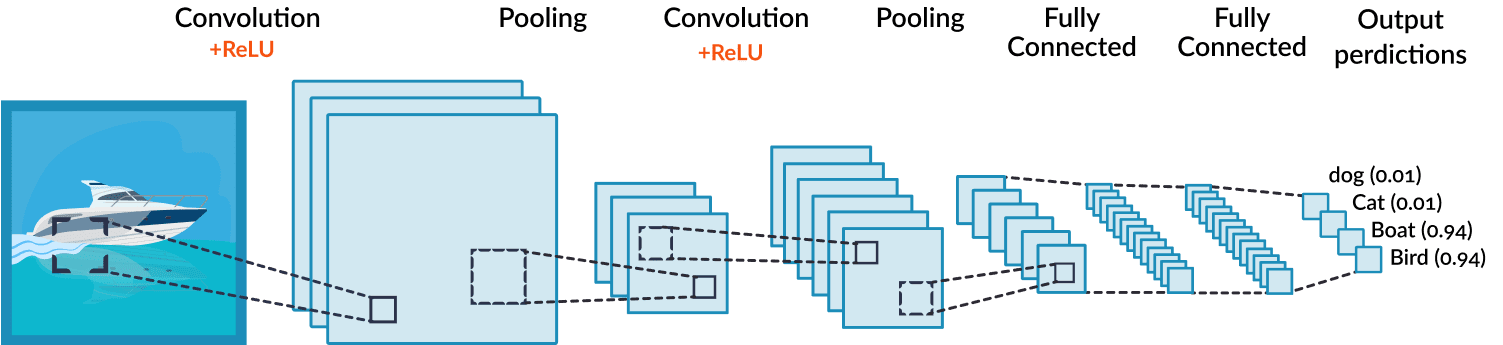


Fig.2 Convolutional Neural Network

# Resnet architecture

The ResNet architecture is based on the presence of some shortcuts between the layers, that allow the network to skip connections.

The main reason for their use is the avoidance of the vanishing gradients problem or to reduce the saturation of the accuracy.

The ResNet is defined in the file: “wrn.py” that specify the possible structures of the network. The standard versions of the ResNet are composed as follows:

1. 2-Dimensional convolutional layer with a kernel of size 3, stride 1 and padding 1
2. Batch normalization layer
3. Three intermediate layers
4. A linear mapper to shape the last layer size

In the approach, we started from a different version of the ResNet named WideResNet.

The main difference is the increased number of channels. The two numbers used after the name means the deepening factor, and the widening factor.

# Approach

Our task is to explore the classification problem by varying some of the parameters, the architectures and the optimizers used.

We will start from a predefined setup with a given setup that achieve 58.22% of accuracy with 500 epochs. Its parameters are summarized in the following table:

|  |  |
| --- | --- |
| **Parameter** | **Value** |
| architecture | Wrn-16-8 |
| rand-shift | 4 |
| batch-size | 10 |
| lr | 4.55e-3 |
| weight-decay | 5.29e-3 |

My approach is summarized as follows:

1. Use some of the optimizers available on PyTorch with the WideResNet-16-8 architecture and choose the best performer among them, using a baseline of 20 epochs
2. Test the best among them on a baseline of 70 epochs and choose the best performer
3. Try these the optimizer with all the feasible predefined architectures (wrn-16-8, wrn-16-10, …) and chose the best among them for the chosen optimizer.
4. Add and tune a new parameter
5. Tune the models on a strong baseline and compare

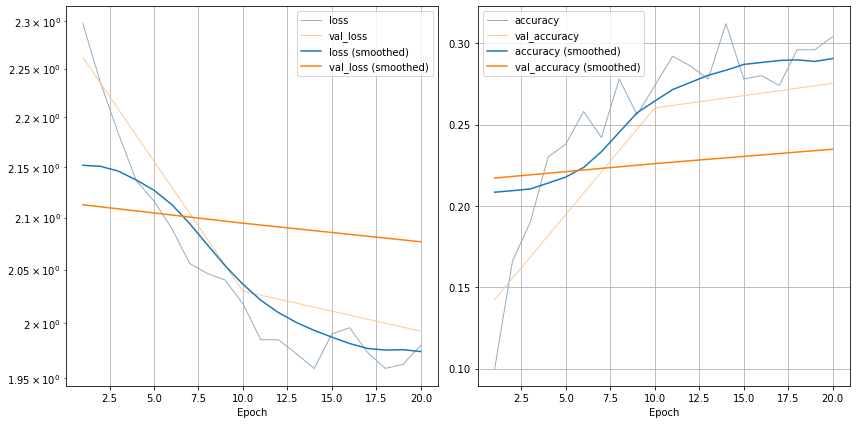
# Optimizers

All the available optimizers on PyTorch have been tried on the resnet20 architecture, the results are shown in the following table:

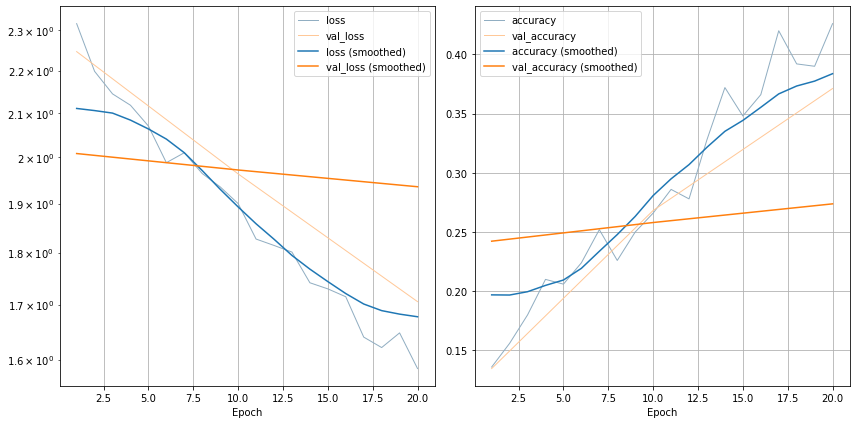
|  |  |
| --- | --- |
| **Optimizer** | **Accuracy on Test Set** |
| Adadelta | 27.53 % |
| Adagrad | 37.12 % |
| Adam | 23.98 % |
| AdamW | 33.73 % |
| Adamax | 33.92 % |
| ASGD | 29.57 % |
| NAdam | 21.98 % |
| RAdam | 32.75 % |
| RMSprop | 23.47 % |
| SGD | 30.18 % |

In the following plots are shown the plots of the results:

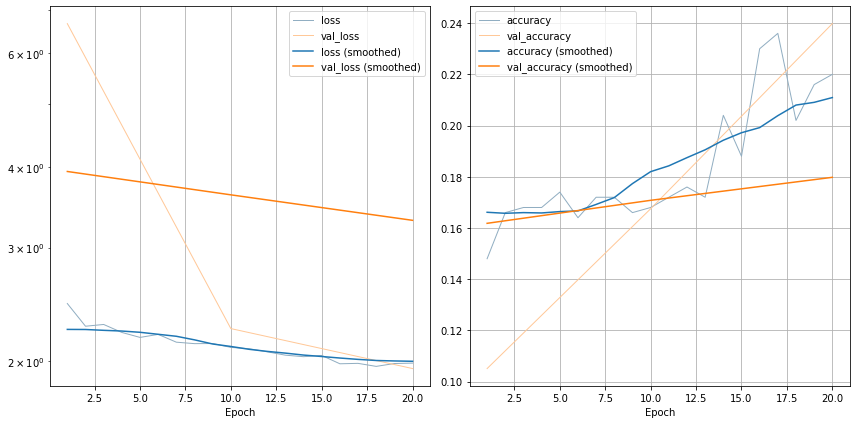
* Adadelta: (20 epochs)



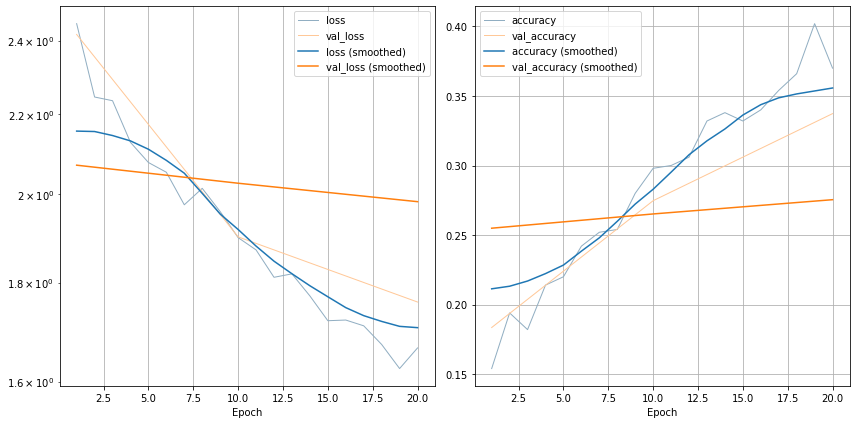
* Adagrad (20 epochs)



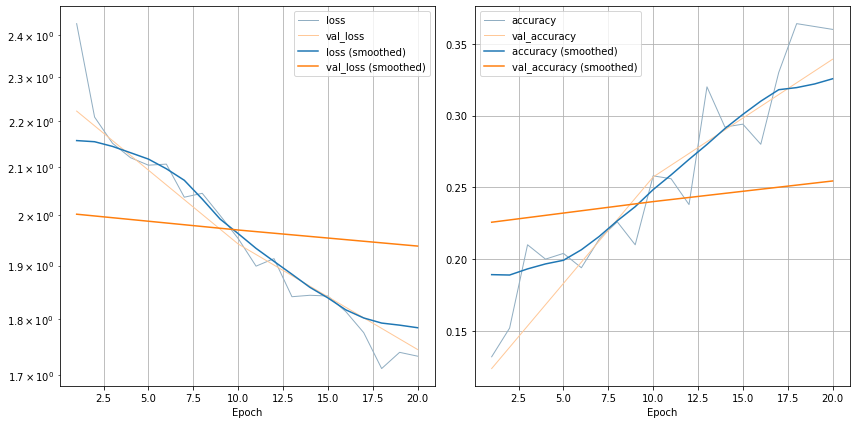
* Adam (20 epochs)



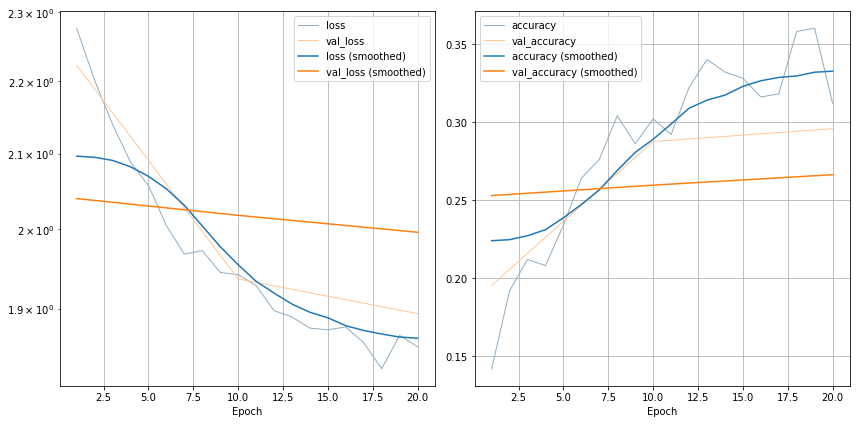
* AdamW (20 epochs)



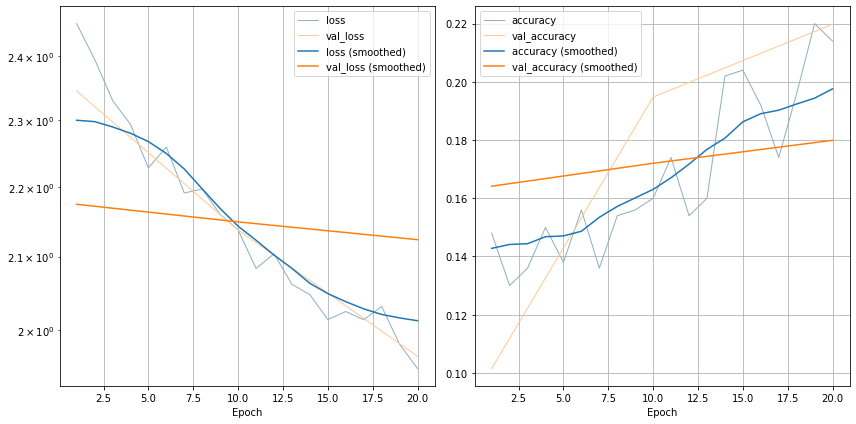
* Adamax (20 epochs)



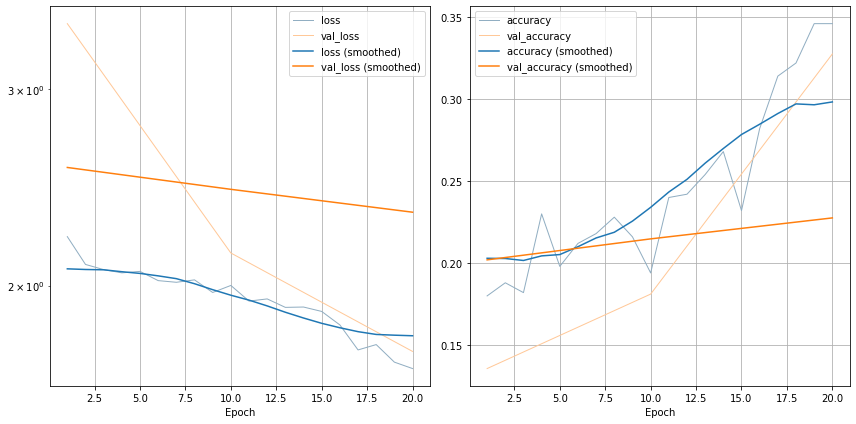
* ASGD (20 epochs)



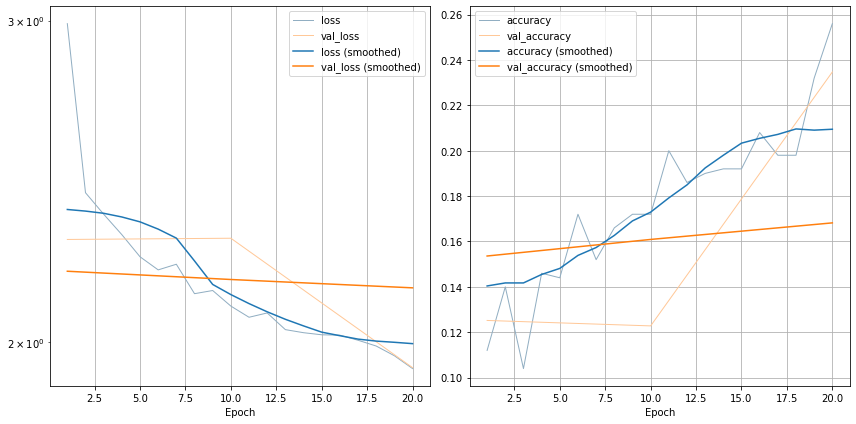
* NAdam (20 epochs)



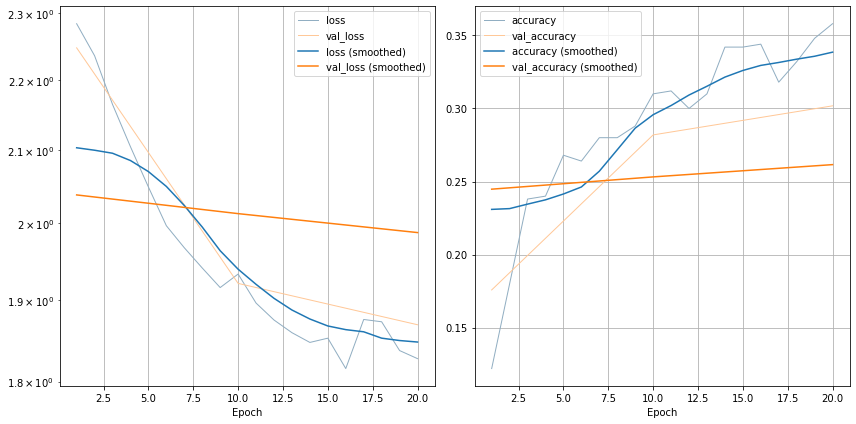
* RAdam (20 epochs)



* RMSprop (20 epochs)



* SGD (20 epochs)

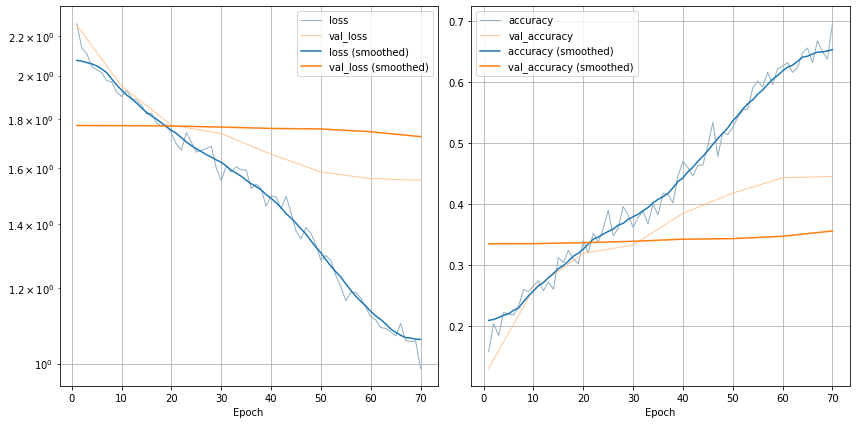


The methods chosen for further analysis are Adagrad, Adam, NAdam, and RAdam.

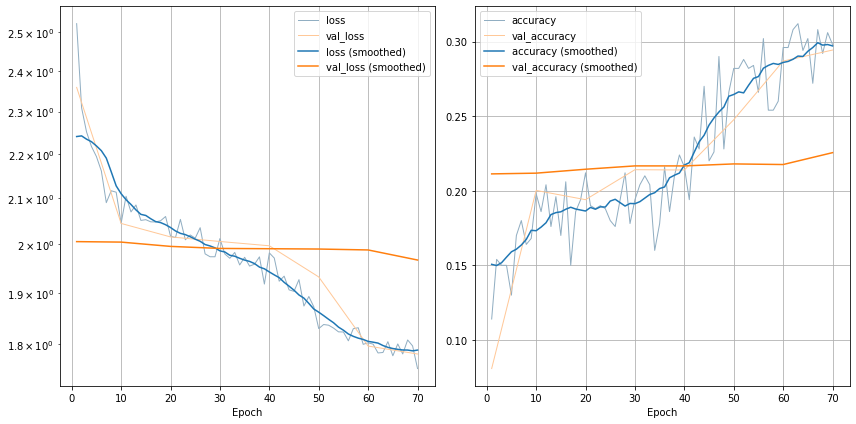
All these four optimizers have been tested on a baseline of 70 epochs. The results are summarized in the following table:

|  |  |
| --- | --- |
| **Optimizer** | **Accuracy on Test Set** |
| Adagrad | 44.53 % |
| Adam | 29.43 % |
| NAdam | 29.76 % |
| RAdam | 32.04 % |
| SGD | 39.98 % |

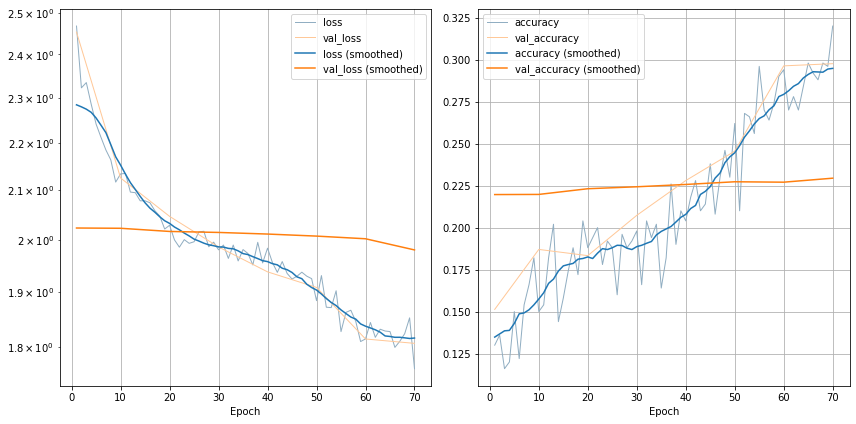
* Adagrad (70 epochs)



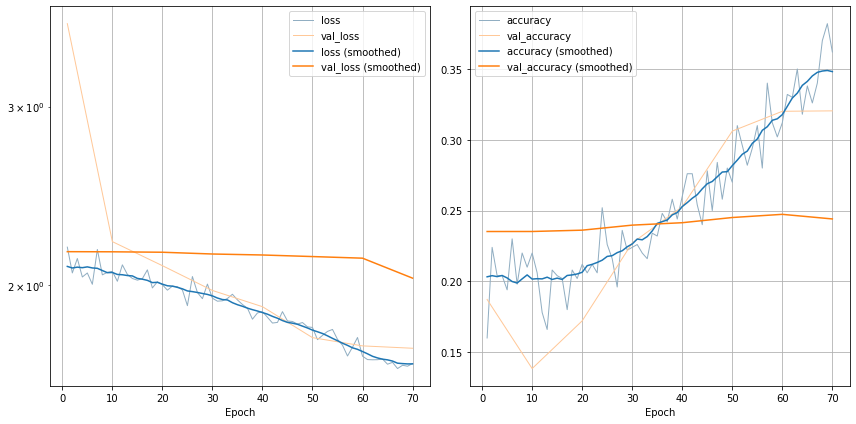
* Adam (70 epochs)



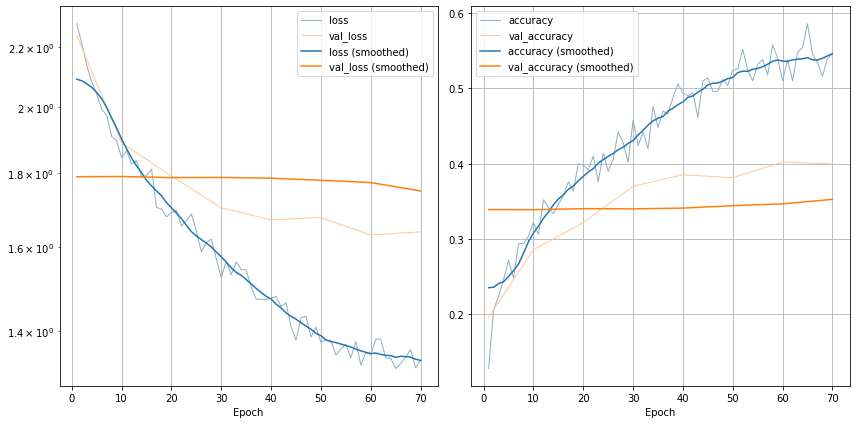
* NAdam (70 epochs)



* RAdam (70 epochs)



* SGD (70 epochs)



The optimizer chosen is Adagrad since it shown an accuracy on test set of 44.53 % on a baseline of 70 epochs.

Of course, the other optimizers have not been discarded since they will achieve a worst overall accuracy. Instead, by changing their architectures and tuning their parameters, they could achieve a better one.

But the purpose of this homework is to explore the possible cases from a predefined setup that achieves 58.22% accuracy on a baseline of 500 epochs.

# Network Settings

In the following section, we show the accuracy for the chosen optimizer, for each of the possible network settings, on a baseline of 70 epochs:

|  |  |
| --- | --- |
| **Network Setting** | **Accuracy on Test Set** |
| wrn-16-8 | 45.75 % |
| wrn-16-10 | 43.23 % |
| wrn-22-8 | 44.28 % |
| wrn-22-10 | 43.02 % |
| wrn-28-10 | 46.26 % |
| wrn-28-12 | 39.57 % |

Since the previous results, we choose to use the wrn-28-10 and wrn-16-8 settings for the WideResNet architecture.

To both architectures will be added a tunable parameter. After the tuning, they will be trained for 750 epochs and the best performer will be choose.

# Parameter Tuning

I will now try to add and tune a new parameter of the chosen optimizer with both the architectures, trying to improve the accuracy of the model.

The parameters I will try to vary is the **learning rate decay** (lr\_decay).

* **Architecture 1: wrn-28-10**:

The results obtained are shown in the following table, given a baseline of 20 epochs:

|  |  |
| --- | --- |
| **Value** | **Achieved accuracy** |
| 10 | 17.00 |
| 1 | 22.85 |
| 0 | 34.47 |
| 0.1 | 29.12 |
|  |  |
| 0.05 | 32.37 |
| 0.02 | 35.05 |
| 0.01 | 39.65 |
| 0.009 | 34.52 |
| 0.005 | 35.05 |
|  |  |
| 0.001 | 36.40 |
| 0.0001 | 35.11 |
| 0.00001 | 35.58 |
| 0.000001 | 33.38 |

The parameter is now set to the value for which we have the best accuracy rate (in green) and I will train the model for 750 epochs.

The trained model with the architecture wrn-28-10 achieves an accuracy of 43.71%, not improving the general overall result.

The performances of the model (750 epochs) are shown in the next plot:

Chart, histogram

Description automatically generated

* **Architecture 2: wrn-16-8**:

The results obtained are shown in the following table, given a baseline of 20 epochs:

|  |  |
| --- | --- |
| **Value** | **Achieved accuracy** |
| 100 | 12.95 % |
| 10 | 16.08 % |
| 1 | 26.62 % |
| 0 | 41.28 % |
| 0.1 | 26.72 % |
| 0.01 | 38.17 % |
|  |  |
| 0.008 | 38.49 % |
| 0.005 | 39.44 % |
| 0.003 | 39.88 % |
| 0.0025 | 37.36 % |
| 0.002 | 40.14 % |
| 0.0015 | 39.34 % |
|  |  |
| 0.001 | 38.87 % |
| 0.0001 | 38.02 % |
| 0.00001 | 38.92 % |
| 0.000001 | 38.18 % |
| 0.0000001 | 38.44 % |

The parameter is now set to the value for which we have the best accuracy rate (in green) and I will train the model for 750 epochs.

The trained model with the architecture wrn-16-8 achieves an accuracy of 49.20%

The performances of the model are shown in the next plot:

Chart, histogram

Description automatically generated

Further tuning can be performed on the model by adjusting the learning rate and the weight decay.

# Conclusions

Starting from a preset scenario, I compared the different optimizers, I trained the best of them on a stronger baseline and choose the best performer (Adagrad). Then I tried to change the architecture within the predefined ones. I have selected the two best architectures (wrn-28-10 and wrn-16-8), added to them a new tunable parameter (learning rate decay), and searched for the best value for it on a narrow set, on both the architectures.

The two best performers (Adagrad, wrn-28-10, lr\_decay = 0.1) and (Adagrad, wrn-16-8, lr\_decay = 0.002) has been trained on a baseline of 750 epochs.

The model choose is (Adagrad, wrn-16-8, lr\_decay = 0.002), that showed an accuracy of 49.20%.