Department of Computer, Control, and Management Engineering Antonio Ruberti

Sapienza
Università di Roma

Machine learning

Report Homework 2

Filippo Betello

1835108

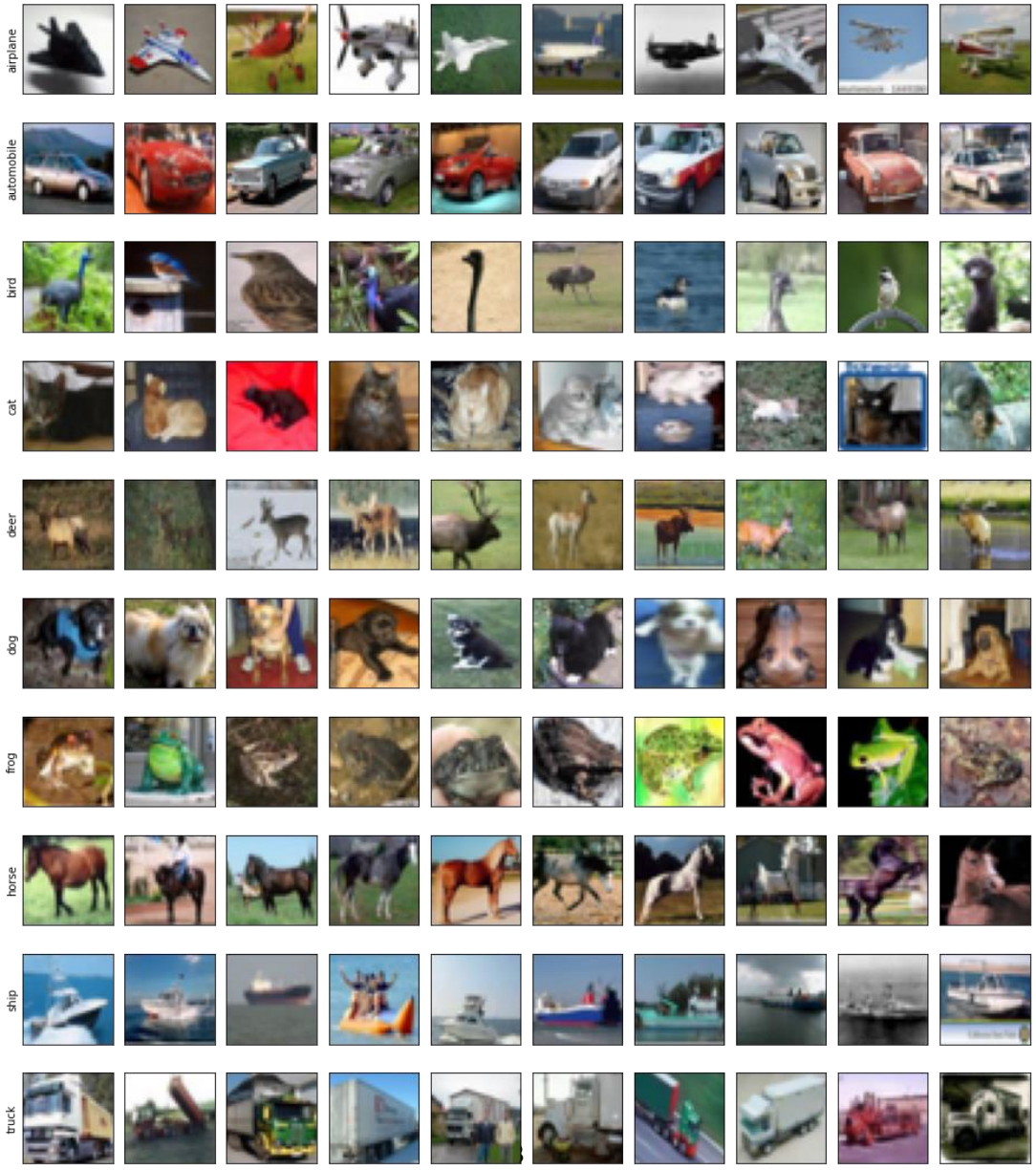
SUMMARY

CHAPTER 1: PROJECT OVERVIEW

The goal of this homework is to provide a model trained on a small image dataset. Doing this every student will produce a different model that will contribute to create a heterogeneous ensemble. A git hub repository is given to modify only some files. Our goal is to add an architecture, add an optimizer method, add a choosable hyperparameter and try to beat the baseline result (58%).

1.1 Dataset

The dataset is ciFAIR-10 of the DEIC Benchmark, it is a variant of the popular CIFAR dataset, which uses a modified test set avoiding near-duplicates between training and test data. It comprises RGB images of size 32x32 spanning 10 classes of everyday objects:



1.2 Dataset Split

The dataset is split in different ways:

1. **train** → there are 30 images per class, for a total of 300/3000 images.
2. **val** → there are 20 images per class, for a total of 200/2000 images.
3. **trainval** → is a combination of the previous two, 50 images per class, for a total of 500/5000 images.
4. **fulltrain** → is the original train set composed of 50.000 images
5. **test** → is the test set composed of 10.000 images

It is used only the *trainval* dataset, in order to be consistent with the task of the homework. One big problem is that the size of the training set is too small to obtain a good accuracy ($\frac{number\ of\ correct\ predictions}{total\ number\ of\ predictions}$). To avoid this, I performed some data augmentation, described in the next chapter.

CHAPTER 2: DATA AUGMENTATION

In order to provide a bigger training set, I tried to use different types of data augmentation. To do this I have extended the *get_data_transforms* function inside the *xent.py* file. I added four data augmentation functions:

1. **Gaussian Blur** → blurs an image with randomly chosen gaussian blur. The parameters are *kernel_size*, that specifies the size of the kernel, and *sigma*, which is the standard deviation to be used for creating kernel to perform blurring.
2. **Auto Augment** → is a common Data Augmentation technique that can improve the accuracy of Image Classification models, as described in [1]. The unique parameter is *value* that specifies the policy: in this case, I used *AutoAugmentPolicy.CIFAR10.*
3. **Color Jitter** → randomly changes the brightness, contrast, saturation, and hue of an image. The parameters are all floaters.
4. **Random Perspective** → performs a random perspective transformation of the given image with a given probability. The parameters are *distortion_scale* (to control the degree of distortion, it ranges from 0 to 1. Default is 0.5) and *p* (probability of the image being transformed. Default is 0.5)

It is not possible to use the *GrayScale* function because the net accepts only an image on 3 channels (RGB). I performed some preliminary experiments to understand which

combination of techniques is the best: I decided to use only the *Auto Augment* function because I noticed a small improvement in the performance.

CHAPTER 3: ARCHITECTURES

In the git hub folder, there are two architectures: the *ResNet* (rn) and the Wide *ResNet* (wrn). For the baseline result the wrn architecture was used. I decided to add two more architectures: *VGG* and *ShuffleNet*.

3.1 VGG

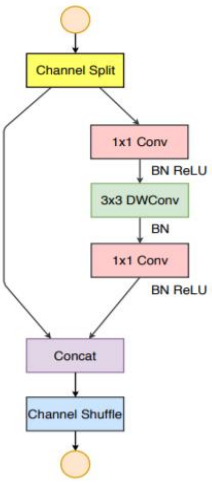
It was developed by the Visual Geometry Group of the university of Oxford in 2015: one of the main advantages is that the convolution filters are very small (3 x 3) and it is fully connected. Originally, it was designed for 224x224 images, now the minimum input size of the model is 32x32. Four types of VGG have been inserted inside the *VGG.py* file:

1. **vggA** has 11 layers depth.
2. **vggB** has 13 layers depth.
3. **vggD** has 16 layers depth.
4. **vggE** has 19 layers depth.

The most used is the *vggD* and because of this I decided to use this one. Each of his hidden layers use the *ReLu* activation function. However, the last one uses *softmax*. Finally, it puts two dropouts to prevent the model to the problem of overfitting. It has also *BatchNormalization* layers to improve the performance.

More details can be found in [2].

3.2 ShuffleNet V2



At the beginning of each unit, the input is split into two branches as in the figure: one branch remains as identity, while the other consist of three convolutions with same input and output channel. Doing this activation functions like *ReLu* exist only in one branch. The high efficiency in each building block enables to use more features channels and larger network capacity. Another important thing is that half of the input goes directly through the block and joins the next one. Moreover, I put four different types of architecture:

1. **sn05** with x0.5 output channels.
2. **sn10** with x1.0 output channels.
3. **sn15** with x1.5 output channels.
4. **sn20** with x2.0 output channels.

For this work, I used sn15. More details can be found in [3].

CHAPTER 4: OPTIMIZER

In the repository, there is already an optimizer: the *Stochastic Gradient Discent (SGD).* It is implemented in the *get_optimizer* function which I override in the *xent.py* file in order to add other two optimizers: *Adam* and *AdaMax*.

The name **Adam** derives from adaptive moment estimation. It only requires first-order gradients with little memory: it combines the best properties from *AdaGrad*, which works well in settings with sparse gradients, but struggles in non-convex optimization of neural networks, and *RMSProp,* which works well in on-line settings.

AdaMax is an extension of *Adam* based on infinite norm: in *Adam* the update rule for individual weights is to scale gradients inversely proportional to a L^2 norm of their individual current and *past* gradient. It is possible to generalize to a L^p norm and when $p \to \infty$ is stable.

These two optimizers are explained in deep in [4].

4.1 User Choices

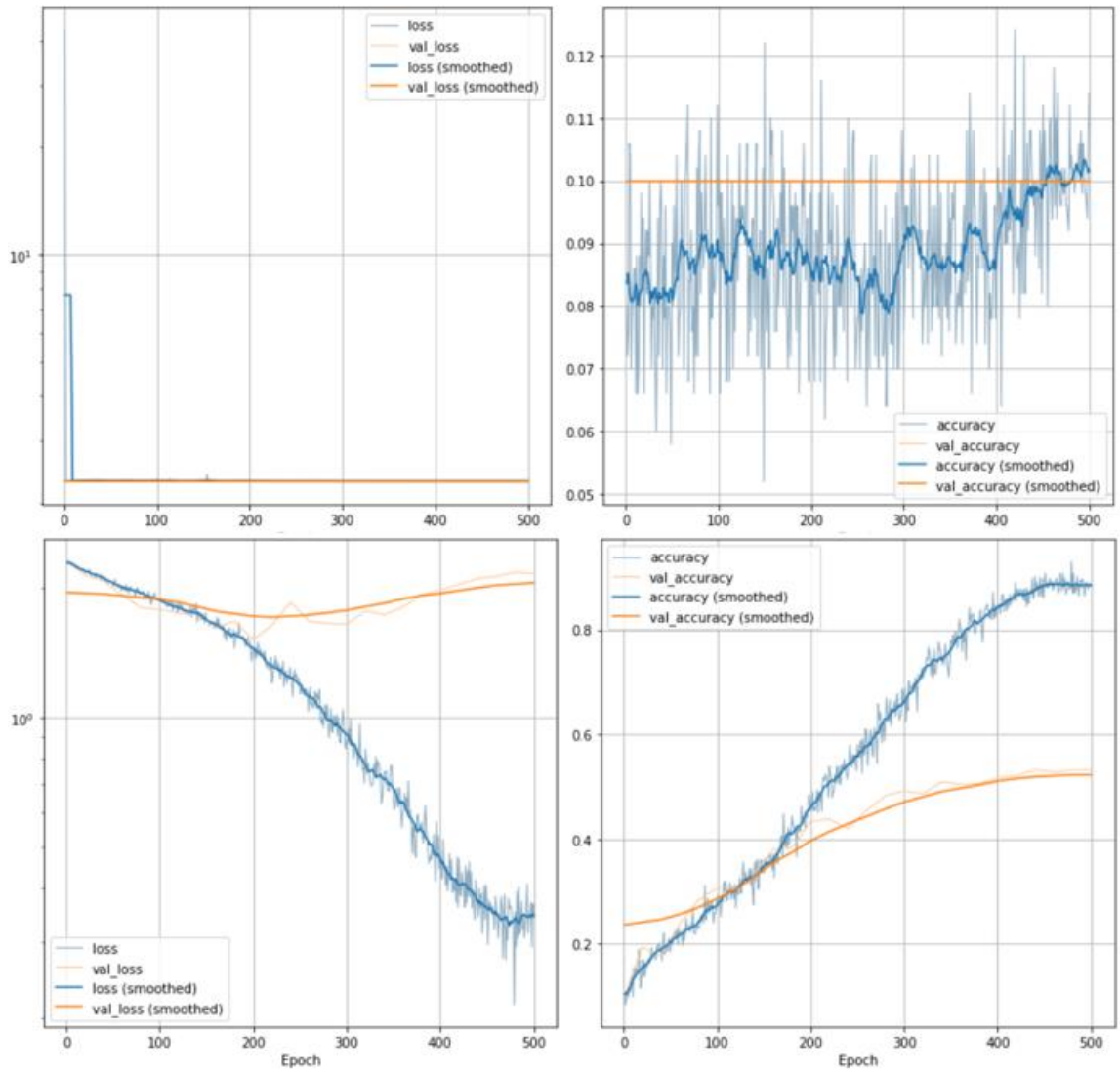
The user that makes use of this code can choose at every run the type of optimizer. A simple question is made "*Which type of optimizer would you choose?"* and three options were given *sgd, adam, adamax*. If the user fails to enter correctly the type of the optimizer, the program will be interrupted and the error "*ERROR WHILE SELECTING AN OPTIMIZER"* will be printed. Furthermore, if the selected optimizer is *sgd* the user can also choose the value for the momentum (the recommended value is 0.9). All of this has been made inside the *get_optimizer* function inside the *xent.py* file.

CHAPTER 5: COMPARISON

In order to find the best combination of *learning rate (lr)* and *weight decay (wd)*, I run some preliminary experiments. What I found is that for the *VGG* the best values are $2x10^{-3}$ and $4x10^{-3}$, respectively for *lr* and *wd*. On the other hand, for both *ResNet* and *ShuffleNet* the best values are 0.0072 for each of *lr* and *wd*. Certainly, the global optimality of these results is not assured. Furthermore, I decided not to use the *init_weights* flag, because it is not much useful to improve the accuracy and I wanted to compare these three architectures without any pre training. While using the *sgd* optimizer, the value for the momentum is set at 0.9.

5.1 VGG

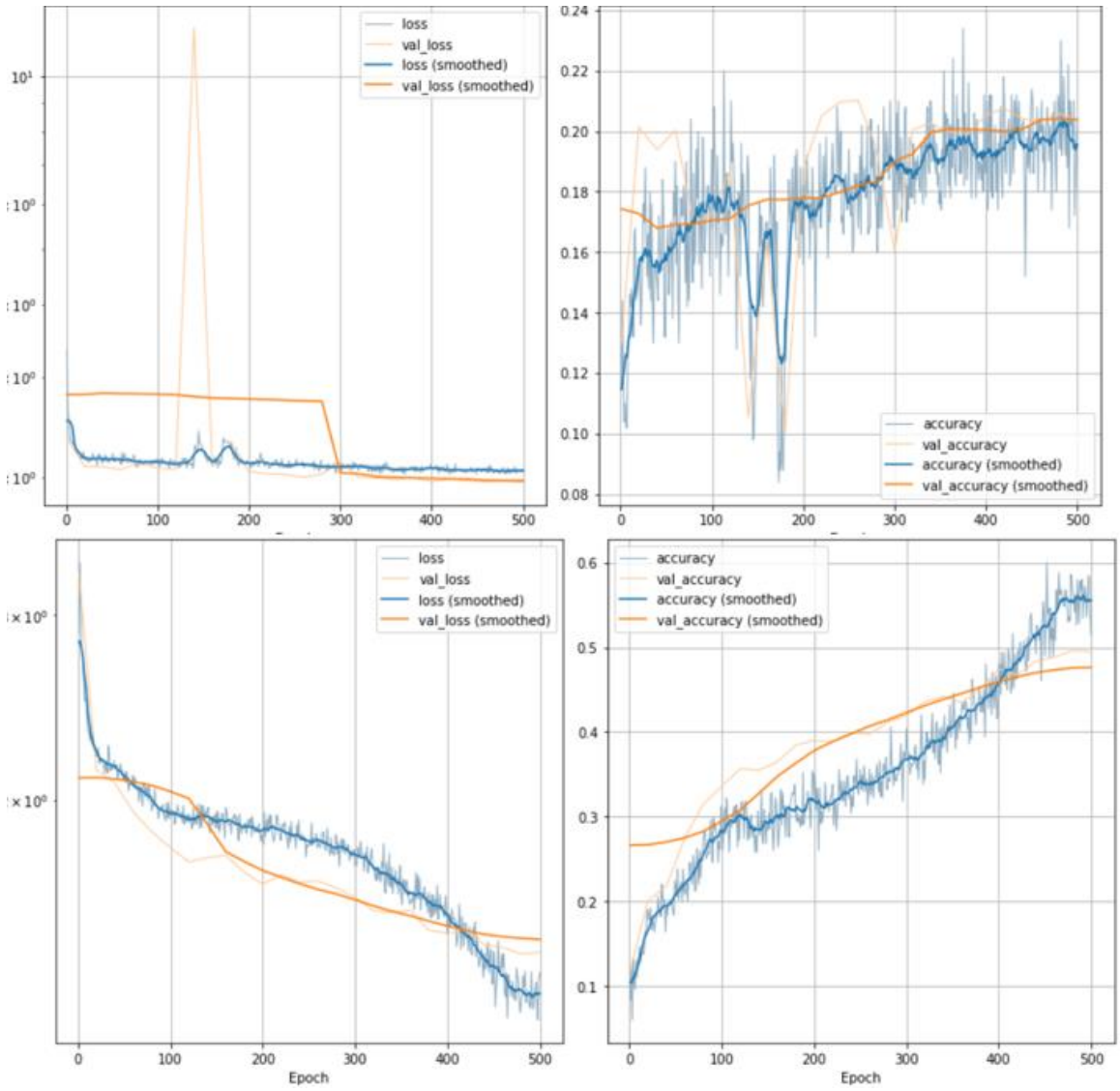
I compared *vvgD* (the one with 16 layers) with *Adam* and *SGD* optimizers for 500 epochs.



As evidenced in the two plots, the one with the *SGA* performed much better. The accuracy is 53.2% computed in less than 1h, versus an accuracy of 10% in 1h and 15 minutes.

5.2 ShuffleNet

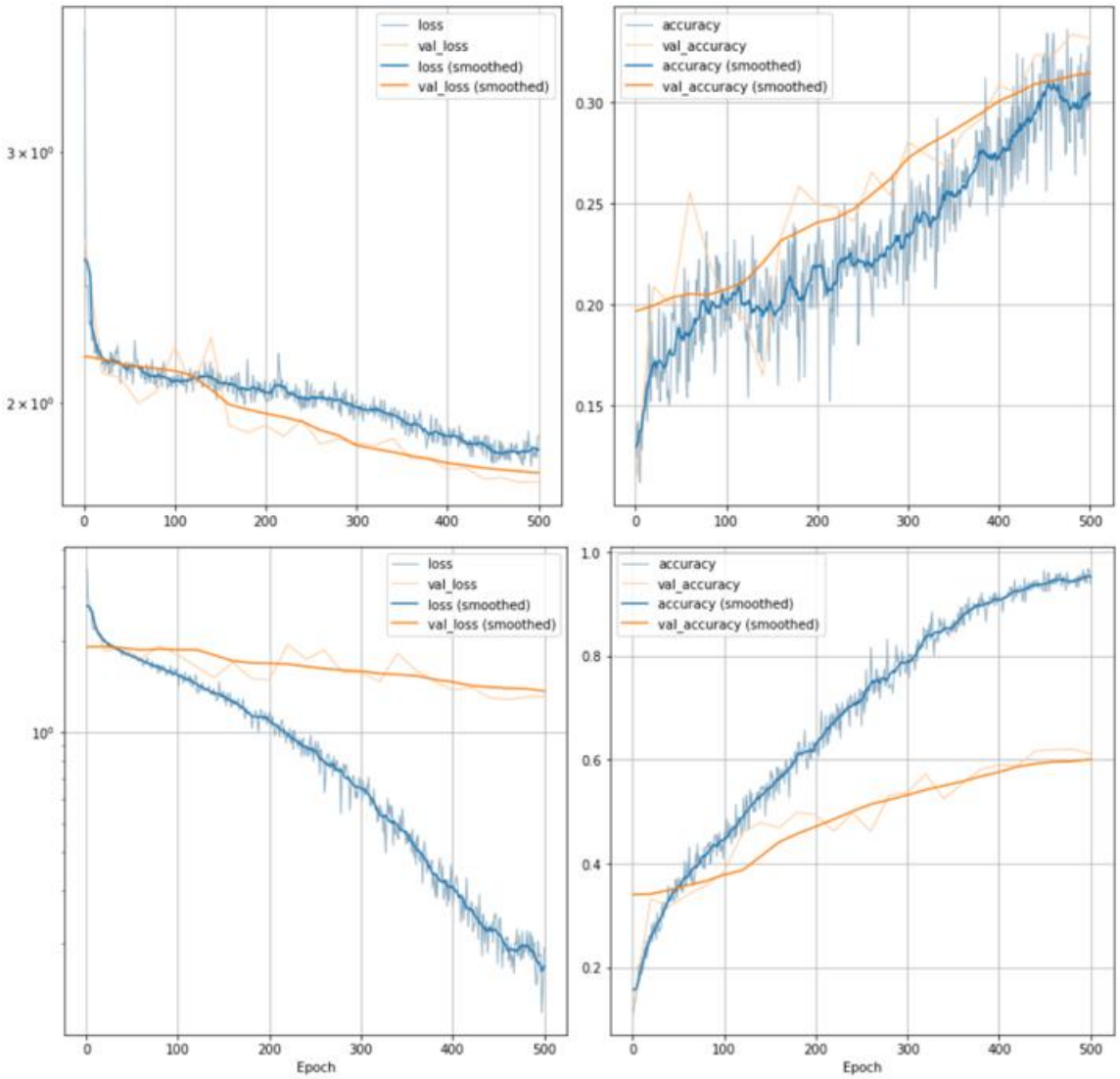
I compared the *sn15* with *Adam* and *SGA* for 500 epochs.



Even in this scenario, the SGA outperform the Adam. The final accuracy is 49.5% computed in less than 30 minutes, while on the other hand the accuracy is 20.4% in 35 minutes.

5.3 ResNet

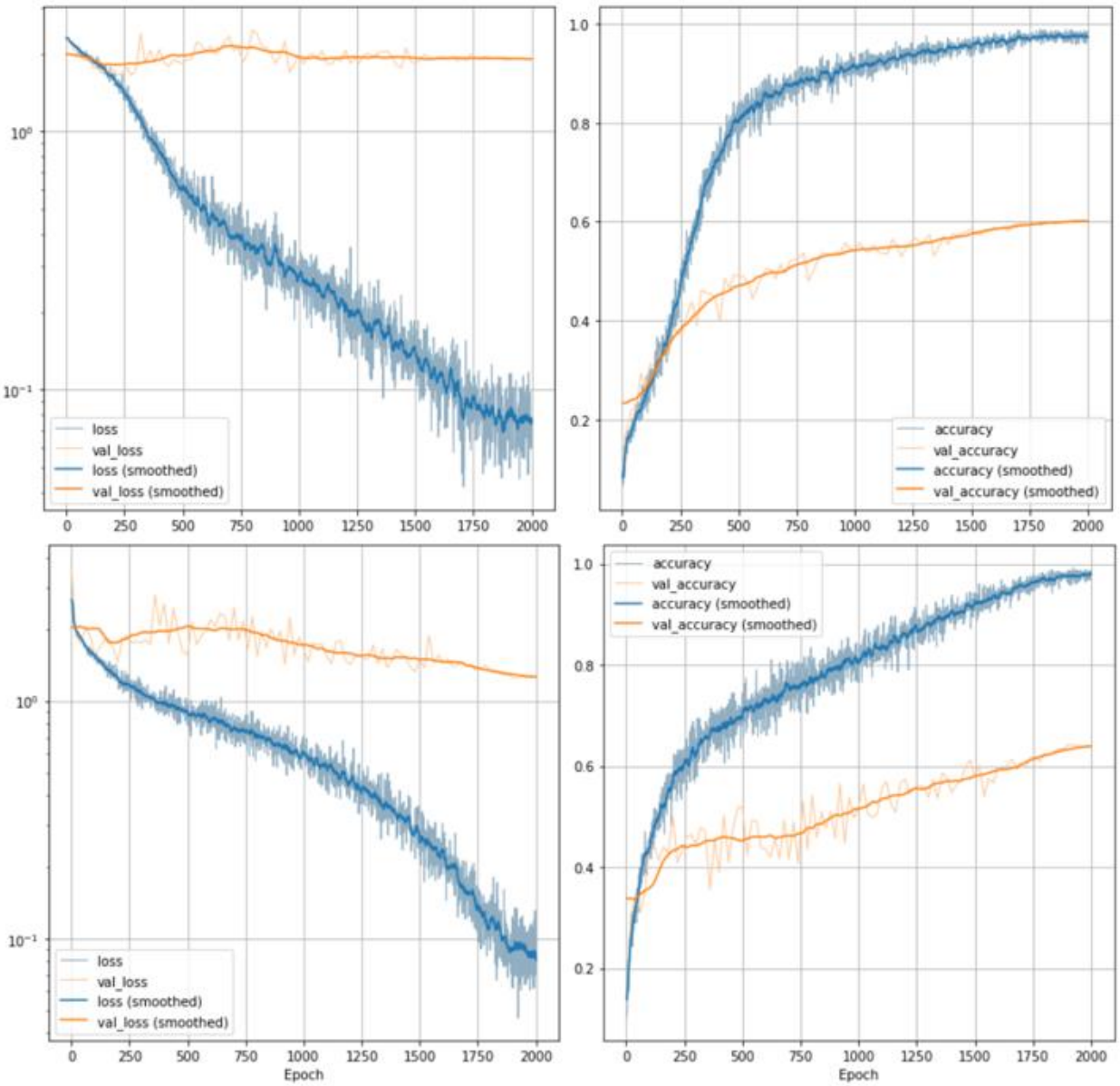
I compared the *rn56* with *Adam* and *SGA* for 500 epochs.



Even in this case, *SGA* beats *Adam*. The final accuracy is 61.2% computed in 37 minutes: it beats the baseline performance of 58%. On the other hand, I obtained 33.2% in 50 minutes.

5.4 Final considerations

The results are clear: *SGA* optimizer outperforms the *Adam* optimizer in every architecture. A motivation of that could be the *lr* and *wd* factors are far from optimality with *Adam*. In order to be consistent with the results I decided to train for 2000 epochs the *vggD* and the *rn56*. The comparison is shown in the next page. The *vggD* showed an accuracy of 60.21% with a computational time of 5h, while the *rn56* obtained 64.1% in 3h. Both of them beat the baseline result of 58%: I decided to submit the result obtained with *rn56.*


loss
val_loss
loss (smoothed)
val_loss (smoothed)
accuracy
val_accuracy
accuracy (smoothed)
val_accuracy (smoothed)
Epoch
Epoch

REFERENCES

1. E.D. Cubuk, B. Zoph, D. Mané et al. (2019). **AutoAugment: Learning Augmentation Strategies from Data**.
2. Karen Simonyan & Andrew Zisserman (2015). **Very deep convolutional networks for large-scale image recognition.**
3. Ningning Ma, Xiangyu Zhang et al. (2018). **ShuffleNet V2: Practical Guidelines for Efficient CNN Architecture Design.**
4. Diederik P. Kingma, Jimmy Lei Ba (2017). **ADAM: A method for stochastic optimization.**