École Polytechnique Fédérale de Lausanne

Deep Learning Mini Projects:

Classification, weight sharing, auxiliary losses
&
Mini Deep-Learning Framework

Pierre CHASSAGNE
289475
Thomas PEETERS
288239
Nada GUERRAOUI
263850

May 28, 2021

***Abstract* - Deep Learning is a domain in constant evolution where a lot of research have been made those last decades and where we have seen new architectures more and more complex appearing. The goal of those two mini-projects is to understand the basics of deep learning by : implementing different architectures to solve a simple task using the pytorch library and by designing a mini "deep learning framework" without using autograd or the neural network module.**

1 Classification, weight sharing, auxiliary losses

The project consists in testing different architectures to compare two numbers contained in two different images in order to show the impact of the different additional parameters that characterize each architecture and to see how they impact the classification performances. To do this, we start by implementing a basic architecture, then we progressively improve it by making it more complex and modulable, adding each time a single parameter to distinguish the effect of each one.

1.1 Structure & Implementation

1.1.1 Basic convolutional neural network

First, we start by implementing a basic convolutional neural network. It is composed of two convolutional layers followed by two linear layers. As an activation function, we chose Relu and this for its various advantages such as the speed of learning compared to other activation functions and the fact that the gradient does not vanish as much as the others. Using this model, we get an important error rate (see result table). The main objective is to try to improve this in the next section by implementing a complex architecture to obtain the lowest error for the testing set.

1.1.2 Complex convolutional neural network

We start by implementing a new architecture using weight sharing, batch normalization, dropout and auxiliary loss.

Weight sharing Since both images come from the same dataset (MNIST), we can optimize our Network by extracting the same features from both images. We can do that by using Weight sharing. We first separate our input (pair of images) into two different input (first image of the pair and second image of the pair) then we apply the same convolution and linear layers to this two images (weight sharing) so that we extract the same features for each one. At the end we combine the two output obtained together and use two linear layers to get the desired output : which image has the biggest number.

Auxiliary loss Auxiliary loss is a partial loss that we compute using output found before the end of the network. The two auxiliary losses are obtained by computing the cross entropy loss between the auxiliary outputs and the training classes (number of the two images of the pair). We tried to implement those auxiliary losses at different layers. First one just after the two first convolutions so that we have more features (256) for each image. Second at the end of the two linear layers where we have less features (10 for each image). We noticed that we obtained a worse result using the first auxiliary loss we compute earlier in our deep network. This result is surprising because we thought that computing loss earlier where we have more information would improve the performance of our model and not the other way around.

Batch normalization and dropout To regularize the network, we add a batch normalization, by shifting and rescaling the layers inputs, and dropout, by removing some units at random during the forward pass and putting them back on test in order to improve the performance, we used two dropout with two different probabilities, p=0.1 and p=0.2, this represents the experimental values that give us the optimal performance.

1.1.3 Residual Neural Network : Resnet

It exists a lot of residual network such as ResNet-50 or ResNet-18 that gives really good results for image classification. The goal was to build a similar network with residual block to test its performance. We use a similar block as the one given in exercise 6 of the course. We notice that by increasing the number of blocks (10 blocks) and a channel size of 32 we obtain intermediate result between the simple and complex neural network implemented before (see result table) but the time computation was much higher since we have much more parameters.

We could probably obtain better result by building a more complex model where the size of the channel of our convolution layers will increase progressively when we advance in the network (such as ResNet-18 and ResNet-50).

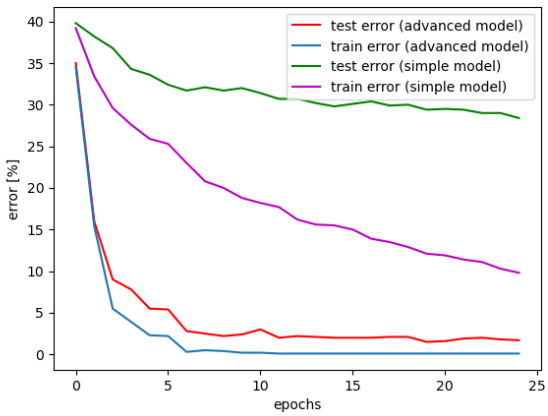
1.2 Results :

Here is the mean test error obtained for the different models. For each model, we use an epoch of 25, a minibatch size of 100, the *Adadelta* optimizer and the *CrossEntropyLoss* criterion:

Table of the results for each architecture

model	mean error	std	lr
Basic model	22.4100%	1.3972%	1
W.S.+A.L.	19.2300%	2.7203%	0.1
W.S.+A.L.+D.	16.00 %	2.5316 %	0.2
W.S.+A.L.+B.	2.4800%	0.5541%	1
W.S.+A.L.+B.+D.	2.2600%	0.5542%	1
R.N.	17.7600%	2.2634%	1

We observe that we improve the performance of our model by using weight sharing and auxiliary loss. Adding batch normalization and dropout allow us to reduce the overfitting and to get even better result. Our final model have a mean error of 2.27%, have 76846 parameters and using an intel core i7 processor takes 11.1 seconds to train the model.



Comparison of the errors for the train and test error for the two models

With this project, we were able to implement different architectures. We have seen the importance of the different techniques used in deep learning and how they improve the model performance.

2 Mini Deep-Learning Framework

This project consists on creating a mini Deep Learning framework, using only the *empty()* method of PyTorch and with autograd globally off. The framework should contain at least five modules : *Linear* (Fully connected layer), *ReLU*, *Tanh*, *lossMSE* and *Sequential*.

Then the goal is to build a network and train it for binary classification on a data-set which consists on 1,000 data-points uniformly distributed between [0,1] x [0,1]. The points are labeled 1 if they are inside the disk centered at (0.5,0.5) of radius $1/\sqrt{2\pi}$ and 0 otherwise.

2.1 Structure & Implementation

The structure of the code is the one advised in the project's statement. All the modules inherit from the abstract class *Module* which defines three global methods : *forward*, *backward* and *param*. The first two are in charge of the forward pass and the backward pass respectively while the last one is in charge of passing the relevant parameters of the entity e.g. the weights and biases of a linear layer. The *Linear* module has also two additional methods : *gradient_step* and *gradient_reset*, which are in charge of updating the gradients after the backward pass and resetting them to zero if needed respectively.

2.2 Architecture

It is stated in the project description that the architecture should be composed of two inputs units, one output unit and three hidden layer of 25 units, the activation function should be *ReLU* or *Tanh*, and the network should be trained with MSE loss. The overall architecture was respected except that for convenience the data-points were labelled with one-hot encoding, hence there are two output units to the network, however it should not affect much the final result. The *Tanh* activation function was chosen over *ReLU* indeed, even if *ReLU* is broadly use in neural network nowadays, *Tanh* strongly maps the negative inputs with -1 and the positive inputs with 1 and is therefore much adapted for binary classification.

The weights and biases of the Linear layers can be initialised uniformly or using a normal distribution. By default the initialisation of the weights is done using the simplified formula for standard *ReLU* of the Kaiming initialisation without taking

into account the factor 2 since it is here to compensate for the fact that *ReLU* layers map the negative inputs to zero [1], here is the formula :

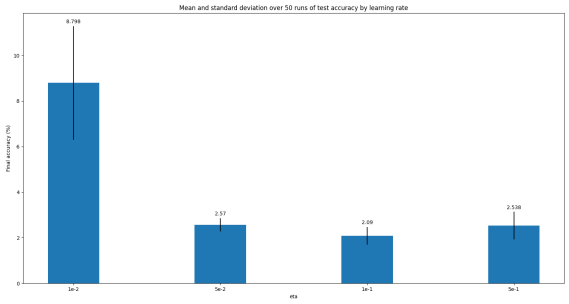
$$W \epsilon \mathcal{N}(0, \sigma) \times \sqrt{\frac{1}{size\_input\_vector}}$$

The biases are initialised to zero.

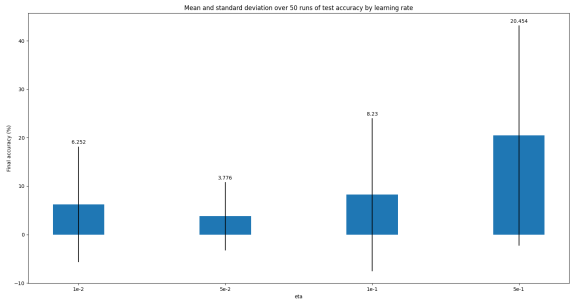
2.3 Results

The scope of the project is to build a framework for deep-learning, the study of the results is only relevant to show that the framework works and behaves as expected. In order to demonstrate that an emphasis will be made on the pertinence and the coherence of these results.

Mean and standard deviation of the testing accuracy over 50 runs

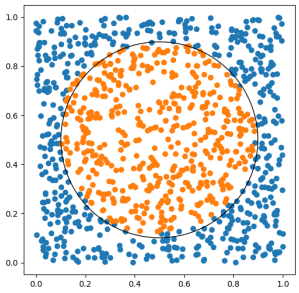


The best learning rate is 0.1, it gives the smallest standard deviation and the best accuracy : 2.09%. However, 0.5 and 0.05 are suitable too.



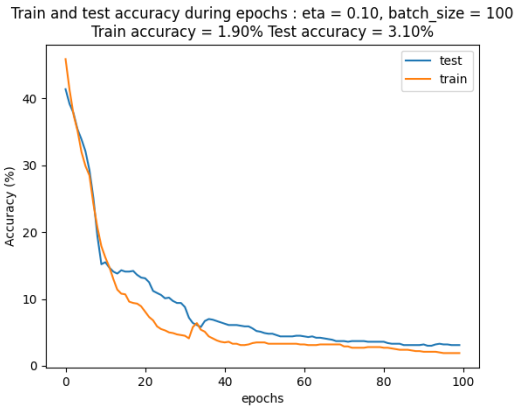
Same but with ReLU activation functions instead of tanh

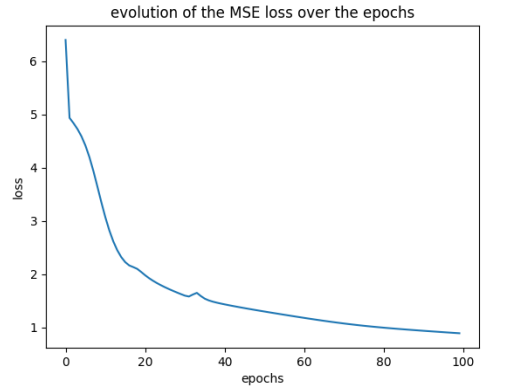
As shown above, using a *ReLU* activation function gives poor results compared to *tanh*, especially in term of variance. These results both demonstrates a good and regular behavior of the model and therefore of the forward and backward passes.



Prediction made by the model on the testing set

Evolution of the accuracy and the loss of the network over the epochs





The loss as well as the accuracy are exponentially decreasing. The network is obviously not overfitting and gives good results. Again it shows that the modules of the framework behaves successfully.

2.4 Conclusion

This mini-project enabled us to have a better insight of how a framework is made and how the basics concepts of deep learning are handled in python. It is an harder tasks than it looks like and it made us appreciate a lot more the powerful tool that is the *autograd* function.

References

[1] Kaiming He et al. *Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification*. 2015. arXiv: 1502.01852 [cs.CV].