Deep Learning: EE559 – Mini-projects

Guilhem AZZANO 300531
Pierre ERBACHER 300533
Anthony PIQUET 300541

***Abstract*—In this paper, we exposed our approach to solve the comparison of two digits with deep learning methods and how we managed to write a custom mini-deep learning framework.**

I. Project 1

In this section we are going to compare two different architectures of neural networks: Fully connected and Convolutional neural networks with and without weight sharing and auxiliary losses. Models will be assigned to the same goal, to compare and scale two digits from images.

A. *Fully connected neural network*

The more common and accessible model in deep learning is the multilayer perceptron or Fully connected network. Literature has already establish that this type of model isn't the most efficient for image classification but in our case it will be our first point of comparison. Thus, we started to build our model for the recognition of a single digit and analyze the best parameters by looping on hidden sizes, number of layers, batch sizes, learning rate values and by trying regularization technique (dropout) and learning rate decay.

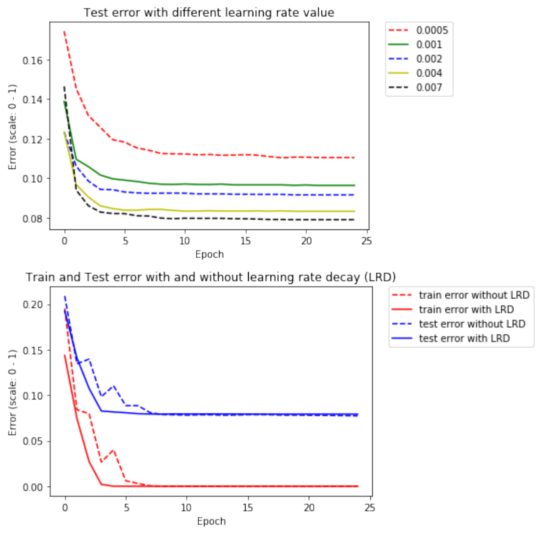


Figure 1: Performances achieved with multiple learning rate and LRD (with batch size: 20 and hidden neurons: 2000)

In the figure 1 we assessed different learning rates and the use of the learning rate decay (LRD) with a scheduler. As shown in the first graph of 1 a higher learning rate allow the model to converge quickly to a more optimal solution. Thanks to the learning rate decay the model can then learn faster, thus achieving better performance, while avoiding over-fitting and instability through epochs.

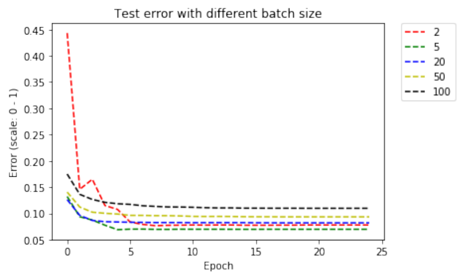


Figure 2: Performances achieved through multiple mini batch size

With the figure 2, we can compare the convergence's speed, a mini-batch size of 5 seems to be the best choice for our problem with the actual parameters: hidden neurons; 500, LR: 0.0004 and learning rate decay turned ON.

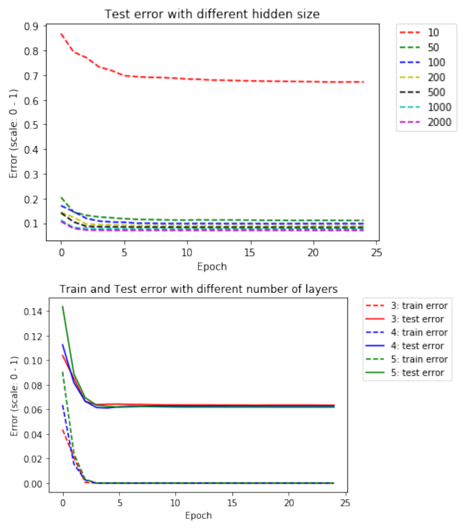


Figure 3: Graphs showing hidden size en layer size influence

The next step was to study if a wider or deeper network would be more efficient at the task than the actual one.The figure 3 show if increasing the number of layers or the depth make our model more efficient. Unfortunately, increasing the number of layers and hidden neurons didn't really improve much our results. The width of our network had more importance than it's depth, probably because deep model are really expressive and we already have 3 layers for this test.

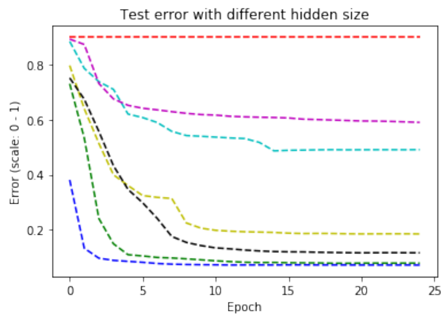
From the digit recognition we added few layers to our model to do the digit comparison. The models have been evaluated ten times with a data-set of size 1000. Our weight sharing configuration is: 3 layers for the digit recognition (196 to 50; 50 to 50; 50 to 10)then 3 layers for the digit comparison (20 to 200; 200 to 200; 200 to 2).

We achieved in a few seconds, 25 epochs and batch size of 50: a mean of 9.4% with a standard deviation of: 0.86 for the digit recognition and a mean of 24% with a derivation of: 2.6 for the digit comparison. But, without the limitation of a few seconds for the training we set a batch size of 1 (SGD), we got in a minute of training a mean of 6.36% with a derivation of: 1.3 for the digit recognition and a mean of 9.05% with a derivation of: 1.4 for the digit comparison. The learning rate decay was turned ON for all the tests.

B. *Convolutional neural network*

As discussed before, a CNN is usually more efficient at image classification as most of the operations are translation-invariant. In that section we built and tried to optimize the different parameters of the convolutional layers to be as performing as possible. Contrary to the fully connected neural network, increasing the number of the hidden size parameter in our CNN changes a lot the accuracy. Our assumption is that with too few hidden neurons our model isn't deep enough but too many will reduce the gradient too much to impact the convolutional layer,thus becoming useless. As we can see in the second graph of 4 the model do not fit correctly our data-set, therefore two fully connected layers is more appropriate.

Once again, reducing the batch size(Figure 5) for the training step has a big impact on our learning. A batch size of 5 seems to be the optimal but will increase a lot the computation time. In the same way the number of channels used in our model has a big impact on its flexibility as each channel can be seen as a transformation of the input. For the CNN, in the weight sharing and auxiliary loss configuration, we used 2 convolutional layers in sequence: Convolution 1: input: 1, output: 64, kernel size: 5 Convolution 2: input: 64, output: 128, kernel size: 5 With ReLU activation function and max pooling: kernel size:3 and stride: 3 after the convolutional layers. The last 2 fully connected layers are of size: 64 to 50 and 50 to 10. We then use 3 fully connected layers for the digit comparison: 20,200; 200,200; 200,2. We achieved with the CNN, 4.5% error for the digit recognition and 2.7% for their comparison. Therefore CNN can not only achieve better performance but can also required less parameters, as the convolution is translation invariant, for similar performance.



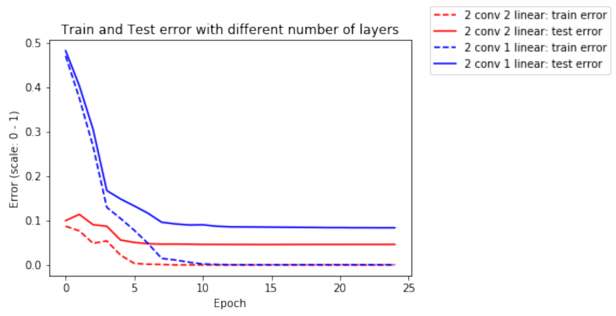
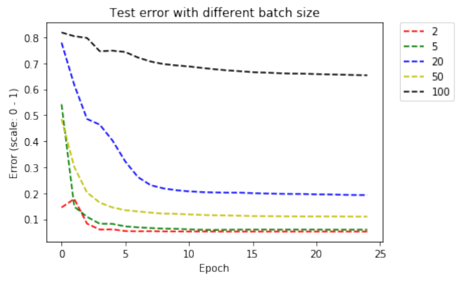


Figure 4: Performance achieved for different hidden neurons and different number of fully connected layers



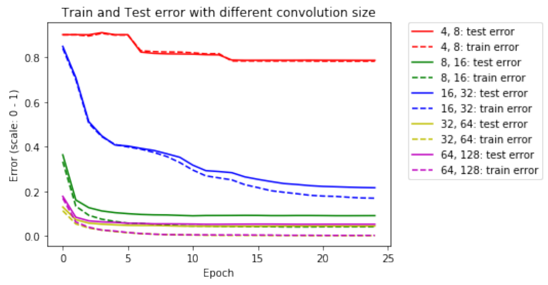


Figure 5: Graphs showing batch size and convolution's parameters influences

C. *Weight sharing and auxiliary loss*

For every architecture presented before we were using weight sharing and auxiliary losses. Indeed the model was trained by reducing the loss on the digit recognition and digit comparisons. Both input images were going through the same layers(Fully connected or Convolutional), one after the other and then the outputs were used as inputs for a new fully connected network. Most of the time the digit recognition was the hardest to tune, and often a simple binary comparison gave us better result. At the end we achieved the results in the Table 6

II. Project 2

In this Pytorch project, we developed a functional mini-framework for a sequential model without using the autograd or the neural-network modules . We tested this framework with a generated set of data.

Model	W_Sharing	Aux Losses	Recognition	Comparison
FCNN	Yes	Yes	6.36 ±1.3	9.05 ±1.4
	Yes	No	47.7 ±11.5	16 ±0.93
	No	Yes	7.6 ±0.74	10.8 ±0.99
	No	No	54.4 ±4.5	18.4 ±1.1
CNN	Yes	Yes	4.5 ±1.09	2.7 ±0.44
	Yes	No	43 ±9	16 ±1.6
	No	No	46.8 ±8.8	18.25 ±0.97

Figure 6: Tables of results

In order to test our models and see how it performs, we generated a dataset from randomly generated coordinates uniformly sampled from [0,1]*[0,1]. Each point with a label 0 if outside the disk of radius $1/\sqrt{2\pi}$ and 1 inside. We displayed 1000 points above.

A. Data

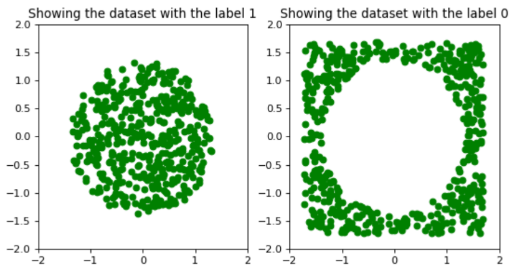


Figure 7: Generated data

B. Structure of mini-framework

The framework is composed of different class : Module, ReLU, Tanh, Sequential, Linear and LossMSE. All the classes inherit from Module class with a forward and backward methods not implemented with learning rate as attribute. We decided to write the Sequential object as a sequence (list) of objects given in parameter. These objects can be Linear, ReLu or Tanh.The loss LossMSE class is given as parameter to sequential. In order for a Layer to have the values of the previous layer, all these objects have a forward and backward method and takes as parameter the previous object in the sequential list.

Like the Pytorch framework, the linear object takes as parameter the input layer size and the output layer size and build a Weight tensor of shape (input,output)

The backward function of all module takes as input the previously computed gradient of the next module. The gradient is then computed back to the 1st layer and is initialized in the last layer after calculation of the loss with the MSE module.

C. Result

For a sequential model with 2 linear layers of 128 neurons each and ReLu and MSE loss and a learning rate fixed at lr = 0.003 we get the Figure 8 for the loss.

With this network we get around 3% of test error and train error. We did not implement any optimizer in our framework so the learning rate is constant. The result are promising for a basic framework as we can see the test and train error are falling below 3% We wanted to check the influence of the size of the hidden layer on the performance of the network as we did the same measurement in the Practical 5. We ran the same data set for multiple layer size and we can see in Figure 10 that wider is the model,smaller is the loss. We tried to test the Tanh module. We built the same network with 2 linear layers but we replace the ReLu module by Tanh module. Even after playing with parameters, and after re-checking the equations in the Tanh module, our results were really bad. We have around 50% of error, so basically the network does not learn.

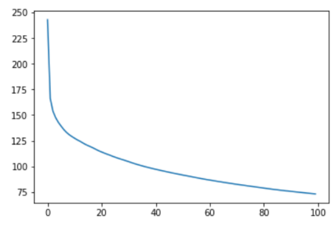


Figure 8: Evolution of the MSE loss

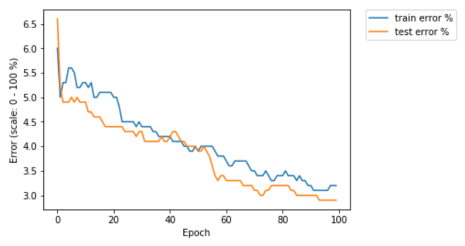


Figure 9: Evolution of train error(blue) and test error(orange

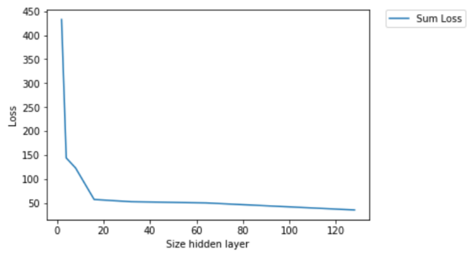


Figure 10: Influence of the hidden layer size on the performance

III. Conclusion

With the 1st project, we learned to build our own neural network with the framework Pytorch. We compared different architecture and their performances. We have now a better understanding of the influence of parameters on neural networks's performances . Thanks to the 2nd project we have a deeper understanding of the sequential framework and how to build a neural network from scratch with the backward pass and the calculation behind.