EE559-Deep Learning Project 1 : Classification, weight sharing, auxiliary losses

Loïc Vandenberghe - Mathieu Toubeix - Walid Ben Naceur
Department of Electrical Engineering, EPFL, Switzerland

***Abstract*—In this report, we tackle the task of recognizing the smaller of two digits from images as inputs. In the machine learning vocabulary, it corresponds to a binary classification problem, with a multi-channel image from the MNIST dataset as input. We propose several models, evaluate and compare their performances in different settings, and analyze the effect of weight sharing and auxiliary losses on this task.**

I. Introduction

Artificial Intelligence is the idea that machines can be built in order to have the capabilities of performing tasks normally requiring human intelligence. This includes many fields such as decision-making, translation, speech recognition and, the subject of this article, image recognition. In this last one, deep learning, a subset of machine learning focusing on algorithms based on how our brain actually works, has been a breakthrough in terms of performances. Nowadays we can recognize with error-rates around or less than 20% [1] [2] facial expression, objects, hand writing or even weimaraners !

In this report we will study digit comparison from hand-writting using the MNIST database. This database is composed of hand-written digits from 0 to 9 stocked in images of 28x28 pixels. For this project we will compress those images into 14x14 images and try to tell if the first digit is strictly less than the second one.

At first sight, we could think that this problem is no different from recognizing digits. Nevertheless, if we take a simple case where digit two is 8 and digit one is either 3,4 or 5 you don't need to precisely identify the second digit to conclude on the comparison and that's why we should try several models that take into account those kind of reasoning a human being can do and therefore use deep learning to help us.

In a first part, we will present the way we worked on this project to ensure an easy use of the code and coherent/comparable results at the end. Then we will analyze the different models we implemented and their performances.

II. Working protocol

A. Code architecture

Regarding the code organisation, the idea was to develop a back-end composed of 3 main files. First a *train.py* where we put all the different methods we used to train our models, those methods are parametric in order to be able to change loss criterion (default is cross entropy loss), epochs or η for example. Then we have a *tool.py* only used for debugging purposes and show some elements of interests.

At the end we have methods files where we describe the behaviour (essentially the forwarding path because we use autograd as backward path) of our different models. Models are in two different categories, the ones going from 14*14 images to labels and the ones going from the labels to comparison. It is important here to note that combining the two models without taking into accounts the labels for training (a.k.a without auxiliary loss) is equivalent to having just one big model going from 14*14 images to comparison with only training on the comparison. Indeed the layer that was supposed to represent the 10 labels is, but in some case, it is simply an hidden layer f size 10 without a real sense humanly speaking. All those models are wrapped in *model_full.py*.

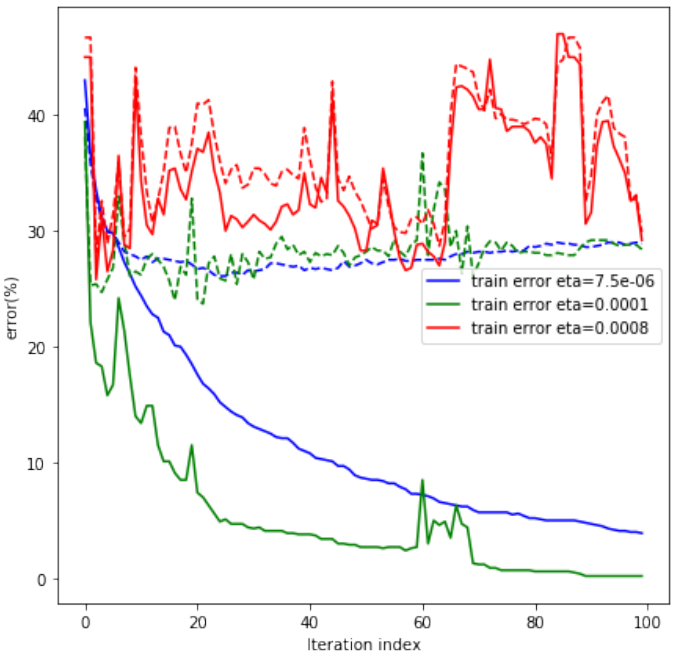
We use either a jupyter notebook for practicality or a *test.py* as a front-end where we train, test and analyses our models.

B. Analysis protocol

In order to have comparable and correct results we had to develop a robust method to train our models and analyse the performances of them. All experiment use the binary cross-entropy loss as primary loss to train our model.

We first operate multiples experiments where we train the model with different learning rate (η). In figure 1 you can see a typical examples. The dot-curve represent the testing error. By looking at the red curve, we see that for a relatively high learning rate the model is unstable and does not learn well on the data. However on smaller learning rate, the learning curve becomes a lot smoother and the model decrease its loss after each epochs. However the model is clearly over-fitting after 20 epochs as the training error approach 0% while the testing error stays around 28%. We also see that an higher η (green curve) makes the model trains faster but still result in an unstable testing error so the ideal η here is $7.5e-06$.

Fig. 1: Several train (full) and tests errors (dots) for a simple model and different η



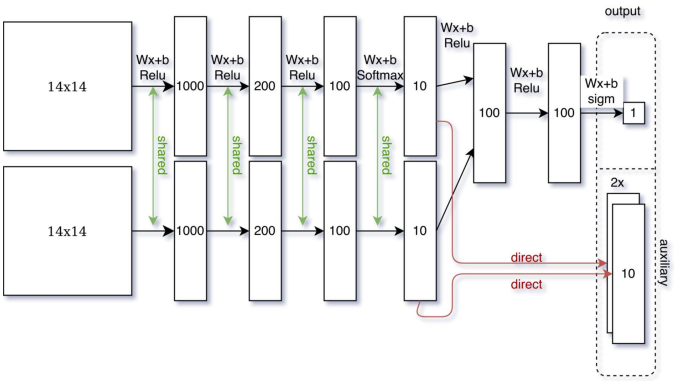
After this first training, in order to validate our model and results, we launch ten training and testing with randomized weights, this way we ensure that we didn't simply get lucky on our training and get relevant results. Those randomized initial weights are already implemented in *torch.nn* [3] and we just had to launch the experiment several times, in all the following figures, the dot lines are all the tests done and the full lines the mean values obtained.

III. Results

A. A first basic approach

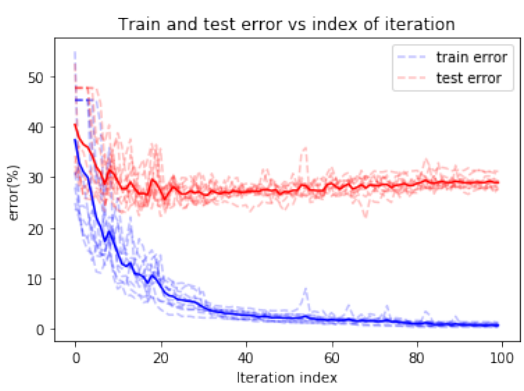
We started with a simple baseline. Our first model is simply a fully connected architecture going from the 14*14 images to the comparison. This architecture (graph 2) is composed of 7 linear layers with ReLUs, one softmax and one sigmoid for a total of 482101 parameters. For now we are not using weight sharing and auxiliary loss, so we can ignore the green and red arrows.

Fig. 2: Data Flow of our first architecture



For this architecture, we fixed $\eta = 1e{-}4$ and the results are visible on figure 3. We obtain a test error of 28.92% .

Fig. 3: Train and test error vs iterations



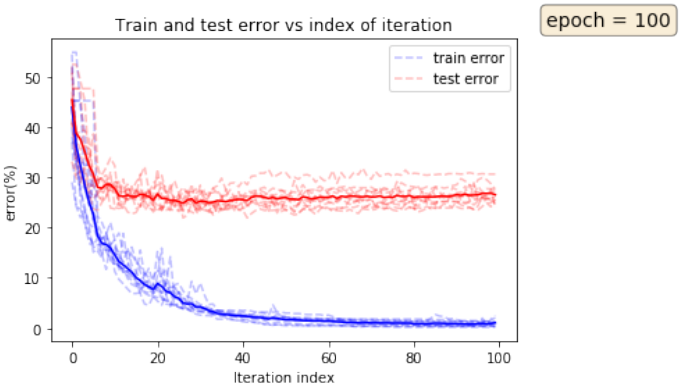
We can obviously do better, a first approach would be to increase depth, by doing so, we increase the number of parameters of our model to get better performances. Nevertheless this approach is very costly in terms of computation and is not a viable approach.

B. Introduction of weight sharing

We are in a perfect case to use weight sharing : we want to compare two images that have the same size and the same characteristics. The idea is to use the same exact model for both digits with the same weights. This way we first decrease drastically the number of parameters of our system (we now have 246711 parameters in our model), but furthermore we also increase performances by using this kind of siamese network. In the graph 2, this equivalent to allowing the green arrows.

The figure 4 shows the results of the same previous architecture but the weights of the two layers taking the digits are the same. We can see clearly here that the performances are better with a test error of 26.53 %.

Fig. 4: Train and test error vs iterations with weight sharing



Nevertheless we can do better, indeed we are not using all the information available in our training data: the classes of the two digits $\in \{0,...,9\}$.

C. Introduction of auxiliary loss

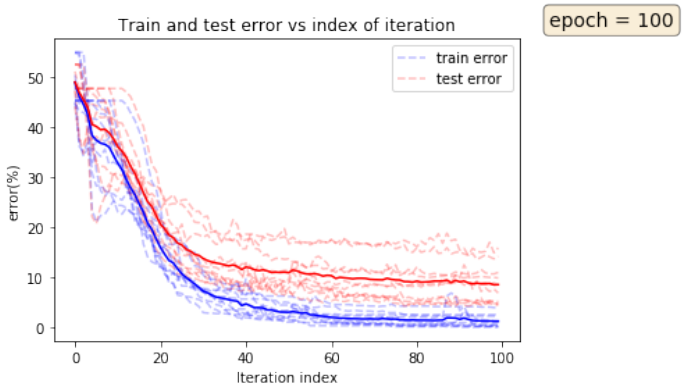
Intuitively, a human trying to solve this task would decompose it into two parts : the recognition of the digits and the comparison of them. The idea of auxiliary loss is to introduce a second loss that forces the model to correlate its network to the second information that it is producing. This way we clearly take advantage of the information of the real values of digits. We use the exact the same models as previously but we now compute our loss as a weighted sum of two cross-entropy losses for the two digit stages and the previous loss. In the graph 2, this equivalent to allowing the red arrows. Hence let α be the auxiliary-loss sharing factor , y be the final output, d_1, d_2 the first and second digit output and $\hat{y}, \hat{d}_1, \hat{d}_2$ the corresponding target,then the loss becomes:

$$(1-\alpha)BCE(y,\hat{y}) + \alpha(CE(d_1,\hat{d}_1) + CE(d_2,\hat{d}_2))$$

The figure 5 shows the results of this experiment for an auxiliary loss sharing factor of 0.9. We improved our performances and obtained a test error of 8.57%. This time disparity is quite more important than without auxiliary loss, the initial point of our weights influences more our training, the minimum test error we got was 4.5% and the maximum 15.9% which is a factor 3 between our best and our worst simply due to our randomized initialisation procedure. Also the over-fitting is reduced drastically since the testing-error follows a lot more the training curve.

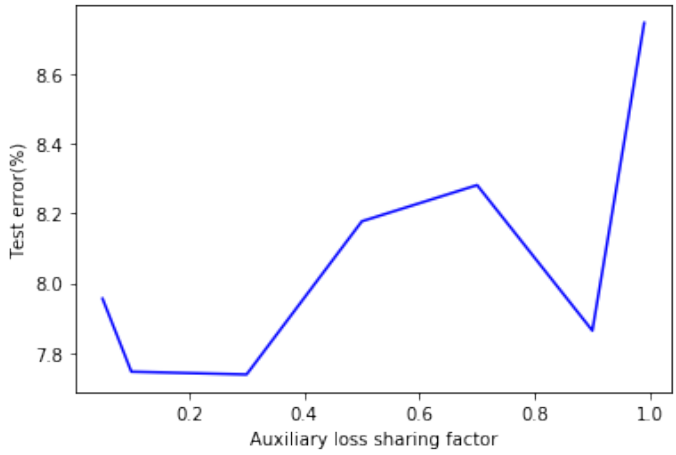
If we run the same experiment with different auxiliary loss sharing factor we obtain the figure 6. This time we repeat the experiment 50 times to have a good statistical power to overcome the high dispersion of data we saw. What we can conclude is that the auxiliary loss

Fig. 5: Train and test error vs iterations with weight sharing and auxiliary loss



sharing factor has no influence on the test error. The system manage to get rid of the auxiliary loss without too much influence on the other part of the architecture.

Fig. 6: Test error vs auxiliary loss sharing factor



D. Use of a more performing model

Now that we have studied the influence of weight sharing and auxiliary loss, we can try more performing models.

We will use a LeNet5 (adapted for our 14*14 input vectors) models for the first part of our architecture (until the layer of size 10) with the use of weight sharing and auxiliary loss and see if our performances increases. The graph 7 represents this architecture. This model uses 56711 parameters which is a factor 5 compared to our previous implementation.

The results can be seen on figure 8. Indeed we increased our performances and now have a test error of 7.45 % . Therefore we managed to increase slightly our performances while we reduced the number of parameters by a factor 5. This results was expected since LeNet5 is a very good architecture for digits recognition for the MNIST dataset. Furthermore, this model seems way more stable than our previous one with less dispersion on the final test error after our training.

Fig. 7: Data Flow of our new architecture

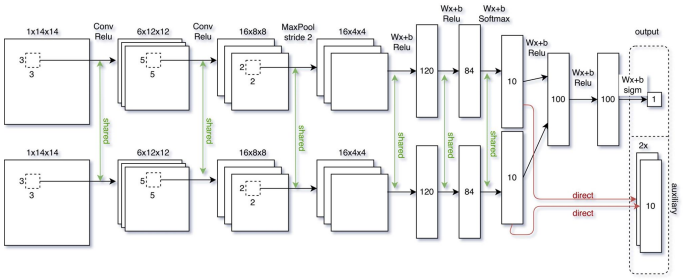
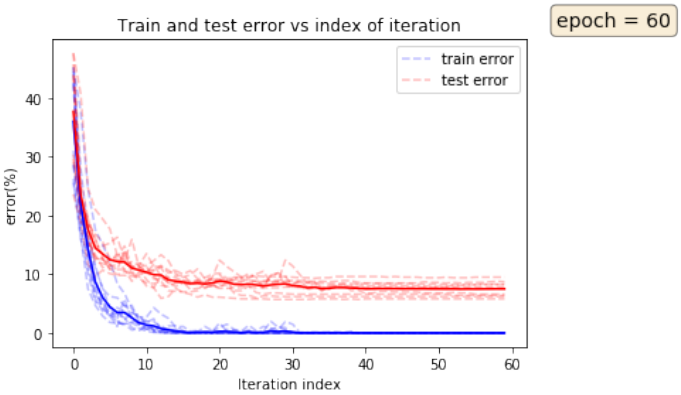


Fig. 8: Train and test error vs iterations for convolutional layers



IV. Conclusion

To conclude the use of weight sharing and auxiliary is a clever way of improving performances without having to pay a huge cost in terms of computation. Our test error with our first architecture went from 28.92% to 8.57%. Nevertheless the introduction of those tools also added some dispersion in the training and training has to be done carefully now. Using some fancy models like LeNet5 allowed us to drastically reduce the number of parameters for the same kind of performances.

Another interesting point is that we didn't yet try the most straight forward way of doing things. If we use our LeNet5 model to recognize digits and then directly with an argmax compares those two digit, we obtain better performances with test errors of 4.57%. In fact, performances for recognise digit is really good compare to the one for comparing them, therefore using a direct algorithm is way more efficient than using a deep net for the comparison. This shows clearly that everyone should think twice before using deep learning everywhere : when an analytical solution exists for a part of the problem, it is sometimes better to use it. Deep learning should be used only on the part where it is really needed. Maybe this problem could be overcome with a very deep net on the second part of the network, that would at least be as performant as simply compare the two most probable digits by taking into account the probabilities of the other digits, but with only 12301 parameters we didn't even touched the performance of the simple comparison. It's a trade-off between performances and number of parameters and here the use of deep learning for the second part of the network is not relevant if we want to keep the number of parameters quite reasonable.

REFERENCES

[1] "Imagenet," *https://fr.wikipedia.org/wiki/ImageNet*.

[2] Z. Li and X. Tang, "Bayesian face recognition using support vector machine and face clustering," 01 2004, pp. II–374.

[3] "torch.nn library," *https://pytorch.org/docs/stable/nn.html*.