UNIVERSITÉ DE GENÈVE
FACULTÉ DES SCIENCES
Département d'informatique

DEEP LEARNING PROJECTS

REPORT

Emile Fontanet, Alexander Safi, Ludovic Fol

December 18, 2022

ABSTRACT

Aims. Mini-project 1 : Test different architectures and methods to solve a deep learning task using Pytorch. The task is, given a pair of 14x14 gray-scale images representing numbers from 0 to 9, to determine whether the first number is larger or smaller than the second one. Mini-project 2 : Create a deep learning framework "from scratch" without the help of Pytorch or other deep learning libraries. The framework should implement Sequential and allow to train using MLE and SGD, combining layers of Linear layers and ReLU/tanh.
Methods. Mini-project 1 : We create eight different models for the task. Each model is a combination of the following parameters : using a CNN or an MLP, using weight sharing or not and using auxiliary loss or not. Mini-project 2 : We implement the framework by following the theory seen in class about MLPs and backward propagation.
Results. Mini-project 1 : All models achieve approximately 80% of correct classification. Overall, auxiliary loss is the parameter that increases the most the accuracy. Weight sharing also improves the results. CNNs tend to give better results than MLPs for this task. Mini-project 2 : The custom framework that we built gives results in the same order of magnitude than that of Pytorch for the same task.

1 Mini project 1

1.1 Introduction

In this mini-project, we aim to test different architectures and methods for solving a deep learning task using PyTorch. The task is to determine whether the first number in a pair of 14x14 grayscale images representing numbers from 0 to 9 is larger or smaller than the second number. To solve this task, we experiment with different combinations of convolutional neural networks (CNNs) and multi-layer perceptrons (MLPs), with or without weight sharing and auxiliary loss.

1.2 Methods

To train the deep learning models, we use the PyTorch library. We create eight different models for the task, each of which is a combination of the following parameters:

- Using a CNN or an MLP
- Using weight sharing or not
- Using auxiliary loss or not

For each model, we initialize the model and the loss function(s) used for training, and then loop through a number of training epochs. In each epoch, we perform the following steps:

1. Zero the gradients of all model parameters
2. Compute the output of the model on the input data
3. Compute the loss using the output and the target data
4. Backpropagate the loss to compute the gradients of the model parameters
5. Update the model parameters using the computed gradients

After training, we evaluate the performance of the trained model on the test dataset and return the accuracy of the model.

1.3 Results

The results for all different configurations are summarized in Table 1 and 2.

CNN	With Aux. loss	Without Aux. loss
With weight sharing	0.82 ± 0.01	0.80 ± 0.01
Without weight sharing	0.80 ± 0.01	0.78 ± 0.02

Table 1: MLP Accuracy results (obtained as an average of 10 runs on the test set)

MLP	With Aux. loss	Without Aux. loss
With weight sharing	0.82 ± 0.02	0.81 ± 0.02
Without weight sharing	0.79 ± 0.02	0.79 ± 0.02

Table 2: CNN Accuracy results (obtained as an average of 10 runs on the test set)

For the CNN data, the overall average of the values is 0.81, and the standard deviation is 0.02. This tells us that there is relatively little variation in performance across the different runs of the model. For the MLP data, the average of the values is 0.80 , and the standard deviation is 0.02. This tells us that the average performance of the MLP model is slightly lower than that of the CNN model, but there is still relatively little variation in performance across the different runs of the model. Overall, these results suggest that both the CNN and MLP models perform well on this task, with relatively consistent performance across different runs of each model. However, the CNN model appears to perform slightly better on average than the MLP model.

Weight sharing can help a model learn more efficiently and improve its performance on a given task by allowing the network to learn a set of weights that can be shared across multiple layers. This can reduce the number of parameters that the model has to learn, which can make training faster and more stable. It can also help the model learn more generalizable features that can be applied to different parts of the input, which can improve its performance on the main task.

Auxiliary loss can help a model learn more complex or hierarchical representations of the data, which can improve its performance on the main task. By adding additional loss functions to the main objective function of the model, the network can learn to extract more useful and discriminative features from the input data. This can improve the model's performance on the main task by allowing it to better capture the underlying structure and regularities in the data.

The data suggests that for the CNN model, using weight sharing and auxiliary loss generally improves performance compared to not using them. The average score for the block where weight sharing and auxiliary loss were both used is 0.82, which is higher than the average scores for the blocks where either weight sharing or auxiliary loss were not used (0.79 and 0.80, respectively). However, the standard deviation is also higher for the block where weight sharing and auxiliary loss were used, indicating that the scores in that block may be more variable.

In the MLP model, weight sharing and auxiliary loss improves the performance as well. The average score is higher when weight sharing is enabled 0.81 compared to 0.79

Similarly, it appears that enabling auxiliary loss slightly improves the performance of the mlp model, as evidenced by the higher average score and lower standard deviation. As we can see that the average score when auxiliary loss is enabled for mlp is 0.81 with a standard deviation of 0.02. When auxiliary loss is disabled, the average score is 0.79 with a standard deviation of 0.02.

All of the models achieve almost 80 percent accuracy on the task of determining whether the first number in a pair of 14x14 grayscale images is larger or smaller than the second number. Overall, the use of auxiliary loss improves the accuracy of the models, and weight sharing also leads to better performance.

1.4 Conclusion

In conclusion, this mini-project demonstrates the effectiveness of deep learning models in solving the task of comparing numbers in images, and highlights the importance of carefully designing and training the models to achieve good performance.

Moreover, we can conclude that the use of weight sharing and auxiliary loss can improve the performance of a model on a given task by helping the network learn more efficiently and extract more useful features from the data. However, the specific effect of these techniques will depend on the specific architecture of the model, the type of data being used, and the task being performed.

2 Mini project 2

2.1 Introduction

In this mini-project, the objective is to create a deep learning framework with only importing torch.empty and the standard math library. The framework will have the following implementations : ReLU, Tanh, forward pass, backward pass, Linear, Sequential, MSE and optimize the parameters with SGD. This will allow us to build a network with two input units, one output unit and three hidden layers of 25 units.

2.2 Implementation

For each module except MSE, we create a class that has 6 methods:

- *__init__()* that initialises all the necessary parameters
- *__call__()*, the forward call of the class.
- *backward()*, the backward pass, where we compute the backward of layer l given the gradient of layer $l+1$.
- *param()*, the parameter function that return the list of all the parameters.
- *update_weights()*, the function that updates the weights.
- *zero_grad()*, the function that resets the gradients to zero.

The activation functions RELu and TanH also have an additional method *gradient()* which returns their derivative. Even though RELu and Tanh do not have any parameters, they still have the weight updating and zero grad methods, that simply return nothing. This is done as a matter of simplicity when computing the backward on the Sequential object, which iteratively calls the backward of all the layers contained in it. The same applies to the zero grad method, which is called for all the layers when called from the Sequential class

With all theses classes created we can then create our framework. We start by setting set_grad_enabled to false to ensure we do not use any built in grad functions. We create the training set and the test set using the torch.empty() method. We create two sets of 1000 points uniformly distributed on $[0,1]^2$. We then label them 1 if they are in the disk centered at $(0.5, 0.5)$ of radius $\frac{1}{\sqrt{2\pi}}$ and 0 otherwise i.e.

$$\text{label=} \begin{cases} 1 \text{ if } (x_1, x_2) \in [0,1]^2 \text{ and } (x_1 - 0.5)^2 + (x_2 - 0.5)^2 \leq \frac{1}{2\pi} \\ 0 \text{ otherwise} \end{cases}$$

We then initialize our framework by calling our Sequential class with our defined layers to create the desired network. Here, as we want to have two input units, one output unit and three hidden layers of 25 units, we have :

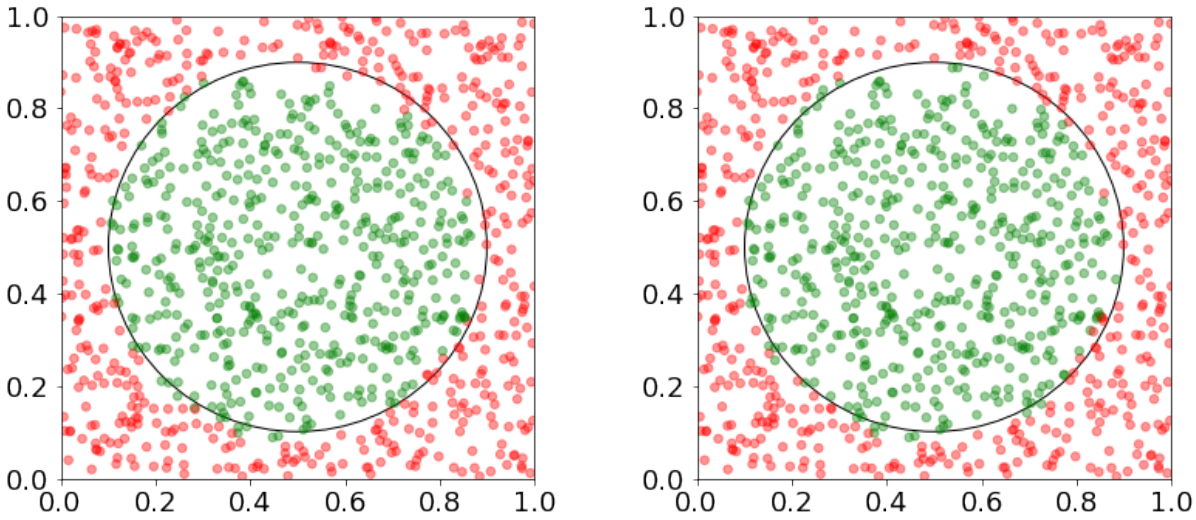
```
Sequential(Linear(2,25), relu(),
           Linear(25,25), relu(),
           Linear(25,25), relu(),
           Linear(25,25), relu(),
           Linear(25,1), tanh())
```

We train our model for 150 epochs and we log the running loss every 10 iterations. The evolution of the loss as a function of the number of epochs for our custom network and a similar one made using Pytorch can be seen on Figure 2. With this result we can check the accuracy of the trained model.

2.3 Results

With 150 epochs of training, our MLP systematically achieves more than 95% accurary on both the training and testing sets. This result is very satisfying and the performance are in the same order of magnitude than that of Pytorch for the same task. An example of a set of results on the training set can be seen on Figures 1a and 1b. We see that, in both cases, almost all the data points are correctly classified.

We can also look at the evolution of the loss with respect to the number of epoch elapsed. Results for our network and a comparison with the one from Pytorch are shown on Figure 2. For both network, we see a very sharp descent at the beginning, followed by a period of very slow decrease of the loss, before it starts decreasing again. While we do not



(a) Classification obtained with our network

(b) Classification obtained with Pytorch's network

Figure 1: Comparison of the results obtained with our network with the ones obtained using Pytorch. The green dots are the data points that are labelled as "inside the circle" by both networks. The red dots are the ones labelled as "outside the circle". We can see that both networks seem to be doing a pretty good job solving this task in 150 epochs.

know the exact reason for this behaviour, it is probable that it could be changed, by starting with a higher value of learning rate and reducing it as the epochs go.

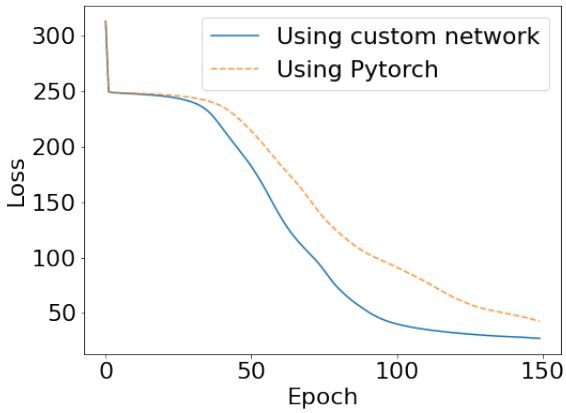


Figure 2: Comparison of the evolution of the losses for our network and the one from Pytorch. Results are an average of 10 runs of both algorithms.

2.4 Conclusion

In conclusion, we see that the custom deep learning framework that we implemented does a very good job at solving the problem we are given. It allows to build networks using any given number of linear layers interconnected with RELus or Tanh. The sequential method that we implemented allows to combine the different layers together and to perform the backward pass on all of them, as well as to update the weights. We see that the network that we built using our own

functions obtains an accuracy very similar to that of Pytorch for the same task, which allows us to be confident about the correctness of our implementation.