[OTP8] Report on lab session #3 - Part I

BADILLO, Jose - MARTINEZ, Gabriela
jose.badillo@student-cs.fr - gabriela.martinez@student-cs.fr

Université Paris-Saclay, CentraleSupélec — December 5, 2019

1 Objective

The objective of this practical session is to build and train a multi-layer perceptron (MLP) classifier with Pytorch. This classifier has been built with the following features (see joint code for details):

- Parametrizable number of layers
- Parametrizable dropout ratio
- Parametrizable activation function

For the training phase, we will consider specific experiments to perform digit recognition for the MNIST data set. so we can conclude on the performance (measured by accuracy and loss error) different network architectures, hyper-parameters tuning and regularization methods.

The data was previously accessed from http://deeplearning.net/data/mnist/mnist.pkl.gz. and divided into training, dev, and test sets.

2 Network architectures

Deep networks with hidden layers of size 128-128, 128-64-32-16, 256-128-64-32-16, 512-256-128-64-32-16, 800-800 wanted to be tested. The following code allowed us to build these different architectures:

```
def getActivationFunction( activation ):
    if activation == "ReLU":
        return nn.ReLU()
    if activation == "Tanh":
        return nn.Tanh()
    if activation == "Sigmoid":
        return nn.Sigmoid()
    else:
        raise Exception('Wrong name for the activation function')
```

```
class MultiLayerPerceptron(nn.Module):
    def __init__(self, input_dim, output_dim, hidden_layers, hidden_dim, activation,
        dropout):
        super( MultiLayerPerceptron, self ).__init__()

        self.activation = activation

        self.functions = OrderedDict()
        self.functions["affine_0"] = nn.Linear(dim_input,hidden_dim[0])
        self.functions["activation_0"] = getActivationFunction(activation)
        self.functions["dropout_0"] = nn.Dropout(dropout)

        for i in range(1,hidden_layers):

            self.functions["affine_"+str(i)] = nn.Linear(hidden_dim[i-1], hidden_dim[i])
            self.functions["activation_"+str(i)] = getActivationFunction(activation)
            self.functions["dropout_"+str(i)] = nn.Dropout(dropout)

        self.functions["affine_"+str(hidden_layers+1)] =
            nn.Linear(hidden_dim[hidden_layers-1], output_dim)

        self.seq = nn.Sequential( self.functions )

        self.init_parameters()

    def init_parameters(self):

        if self.activation == "Tanh":
            for idx, m in enumerate( self.seq.modules() ):
                if type(m) == nn.Linear:
                    torch.nn.init.xavier_uniform_(m.weight.data)

    def forward(self,x):
        return self.seq(x)
```

3 Network training

3.1 The training loop

A major difference of our training loop for this exercise when compared to the lab session #2 is that now we will allow the deep network to be trained with batches of images. Instead of passing one image every time for training, as we did before, we can select the batch size as a parameter of the training. The same applies for the evaluation of the *dev* test. See the details of this code in the corresponding Jupyter notebook.

3.2 Gradient clipping

Also, we are implementing gradient clipping in the training process, so we can deal with numerical overflow or underflow (exploding gradients, more likely to happen if we use, for instance, the *sigmoid* activation function).

3.3 Activation functions

Three activation functions will be considered for training our deep networks: ***sigmoid***, ***ReLU*** and ***tanh***.

3.4 Regularization

In some of the experiments performed, which will be shown in the following section, we aimed to test the effects of two regularization techniques: ***dropout*** and ***weight decay***.

3.5 Performance measures

Our main goal is to study how accurate different network architectures can be on the classification of the MNIST data. Specifically, accuracy will be measured on the three data sets: training, dev and test, to evaluate possible overfitting. Also, we will keep track of the negative log likelihood behavior (loss function) over the epochs on training and dev data.

4 Experiments

4.1 Testing different activation functions

This experiment aims to test which activation function leads to better results in terms of accuracy on the test set. We trained two different architectures of deep networks (128-128 and 128-64-32-16) with the three activation functions we mentioned before. For this first experiment, we considered a batch size of 100, a learning rate of 0.01, 10 epochs and no regularization/dropout.

		Accuracy		
Architecture	**Activation function**	**Training**	**Dev**	**Test**
128-128	ReLU	91.33%	92.20%	91.70%
	tanh	92.17%	92.57%	92.66%
	Sigmoid	33.28%	40.28%	40.99%
128-64-32-16	ReLU	91.23%	92.10%	91.74%
	tanh	92.36%	92.85%	92.66%
	Sigmoid	33.29%	38.71%	37.74%

Table 1. Accuracy on training, dev and test sets for two network architectures and three activation functions

From the previous table, it is evident that the performance of the *sigmoid* activation function is much worse, as it never went up over 50% in accuracy for any data set. This was expected to happen, as this function suits more for binary classification problems. Regarding *ReLU* and *tanh*, they tend to behave similarly and both offer high performance. However, for these two different architectures, *tanh* ended up with a slightly better accuracy in all the sets when compared to *ReLU*.

We also looked at the loss behavior over epochs for each architecture and activation function. The figure below shows in the left side the error diminish general **curvature** that all the architectures followed when using either *ReLU* or *tanh* as activation functions. In the right side, the curvature that the error followed when using the *sigmoid* function is shown. As can be seen, the evolution of the error when using this latter function is not converging to its minimal, which means the optimization problem is not being approached correctly by these architectures, and therefore they are not learning properly.

Given the previous, **we will only consider *ReLU* and *tanh* activation functions for future experiments.**

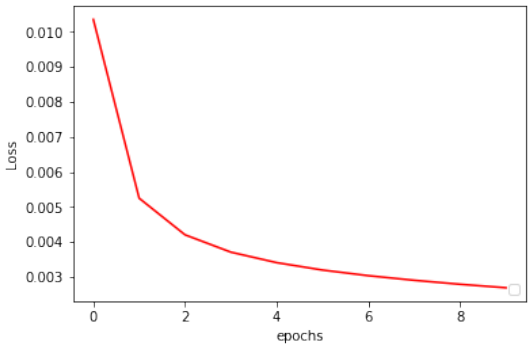

Figure 1. Using *ReLU* or *tanh*

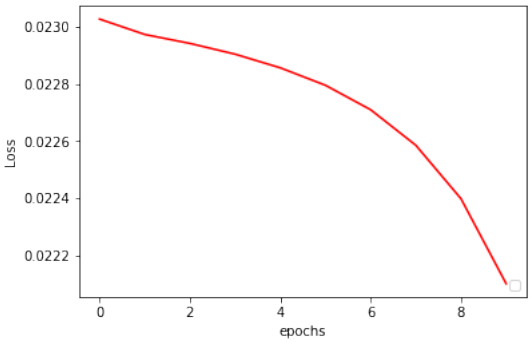

Figure 2. Using *sigmoid*

Figure 3. Loss behavior for the architectures and activation functions considered in this experiment

Info: Note that for the architectures trained in this experiment, we always got higher accuracies in the testing set when compared to the training set. This means that our networks are generalizing well. However, we cannot rely on training accuracy to make conclusions.

4.2 Testing different batch sizes

This experiment aims to explore how the batch size affects the performance of a deep network. We will explore in the simplest architecture (128-128), how the dev set accuracy behaves when we train with batches of 1, 10, 40, 70, 100, 130 and 160 images. Still, we will stick to a learning rate of 0.01, but this time we will consider only 5 epochs (to accelerate the training loop for small batches) and no regularization/dropout. As we want to evaluate the impact of the batch size, we will use the *tanh* activation function for all the trainings. The following graphic illustrates the results obtained:

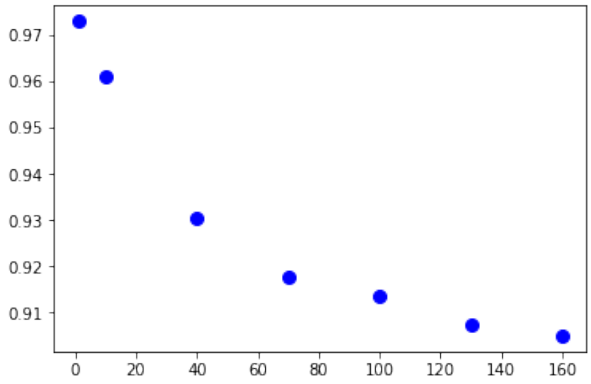

Figure 4. Accuracies on dev test for 128-128 architecture using different batch sizes

Consistently with these findings, higher training times were registered when batch sizes were smaller. The network performance on the dev set is defined by a trade-off between speed of computation and accuracy. As a result, **we have decided to run future experiments with a batch size of 10 images**, as accuracy on dev is stil not that compromised and the training time is way less when compared to batches of 1 image.

4.3 Training an overfitted network

This experiment aims to train a network that is overfitted, so we can explore later the effects of regularization techniques to alleviate this undesired behavior. To get overfitting, we tried the following architectures without regularization/dropout.

Architecture	epochs	Batch size	Activation function	Learning rate
800-800-800	300	8000	ReLU	0.01
512-256-128-64-32-16				
800-512-256-800-1200				

Table 2. Architectures expected to generate overfitting on the MNIST training data set

Accuracy and loss behaviors were found to be similar amongst the three architectures. For this reason, we will only show the results for the 800-800-800 architecture:

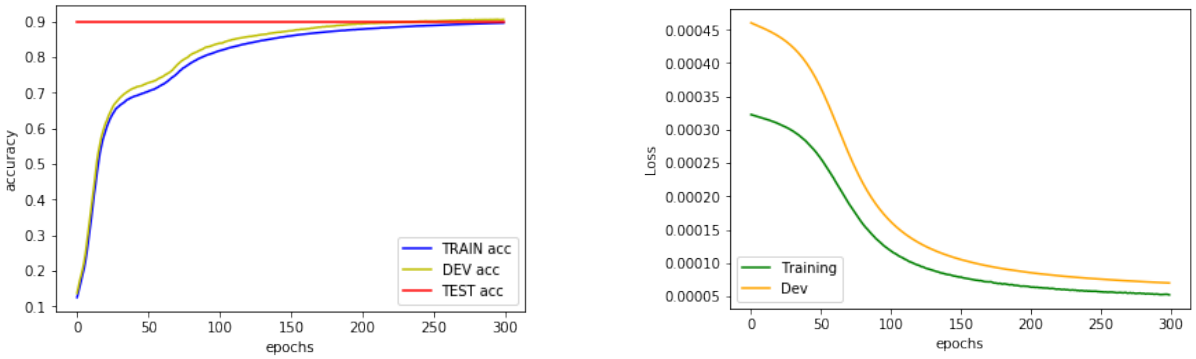

Figure 5. Accuracy and loss behavior for the 800-800-800 architecture considered in this experiment

From the previous graphics, we could not evidence overfitting in this network, as both the train and dev accuracies grow consistently over epochs, while their corresponding errors diminished in a parallel way.

We also trained this same network architecture for 500 epochs (instead of 300) and with a batch size of 2000 images (using the same activation function and learning rate as in the previous experiment). The results are summarized below, in which case observing overfitting was still impossible, even after involving more training epochs.

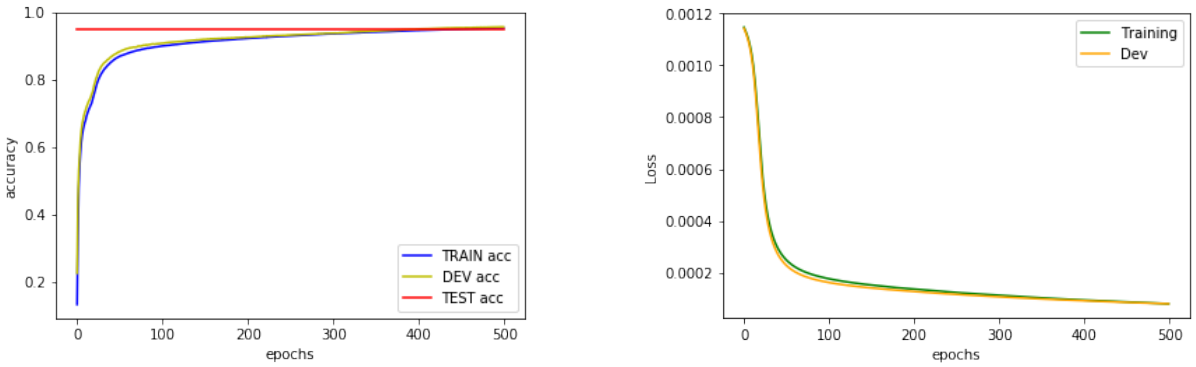

Figure 6. Accuracy and loss behavior for the 800-800-800 architecture, 500 epochs and batch size = 2000 images

4.4 Testing the effect of regularization through weight decay

In this section, even when we did not get to overfit a network, we will use one of the architectures of the previous experiment to see any potential effect of regularization on accuracy. We will train again a 800-800-800 deep net considering a batch size of 10 images, 20 epochs, *ReLU* as activation function and a weight decay of 0.001. Results are presented below:

		Accuracy			
Architecture	**Activation function**	**Training**	**Dev**	**Test**	**Variation training vs test**
800-800-800	Weight decay	99.18%	97.94%	97.88%	-1%
	No regularization	99.89%	98.11%	98.05%	-2%

Table 3. Accuracy for training, dev and test sets for two 800-800-800 deep networks trained with and without regularization

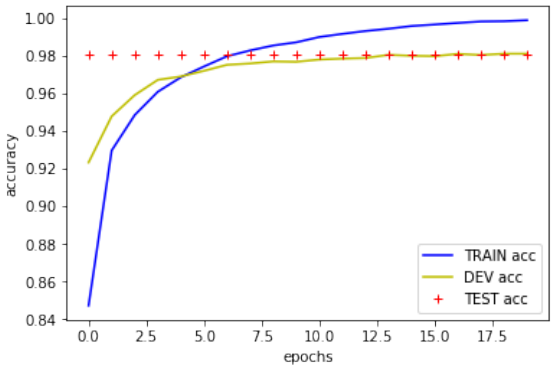

Figure 7. Network without regularization

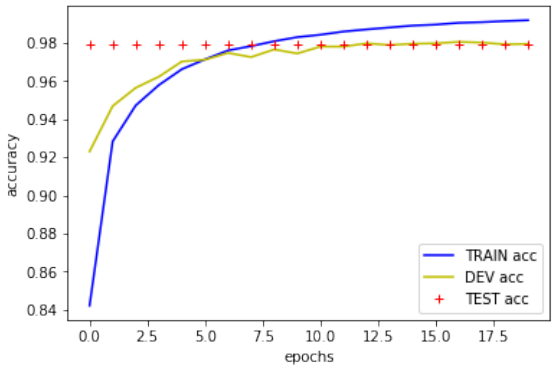

Figure 8. Network with weight decay

Figure 9. Accuracy for training, dev and test sets for two 800-800-800 deep networks trained with and without regularization

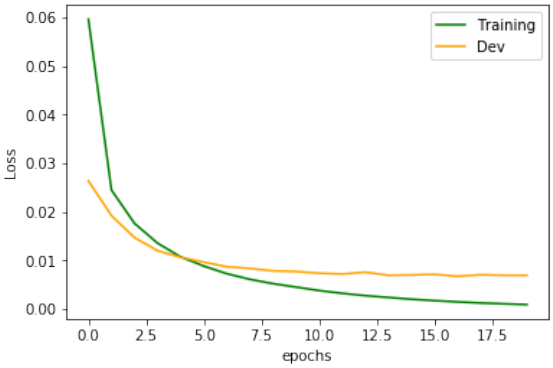

Figure 10. Network without regularization

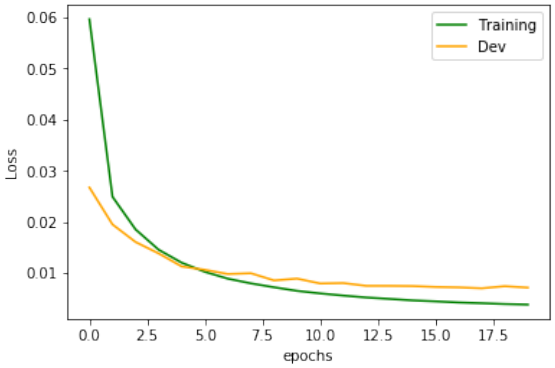

Figure 11. Network with weight decay

Figure 12. Training and dev loss for two 800-800-800 deep networks trained with and without regularization

Results from the previous experiment show that the weight decay technique closed a bit the gap between the training and test accuracies when compared to the network without regularization, in which case both the training and the test accuracies are slightly higher but more separated (having a gap of 2% between them, vs 1% when using weight decay). This behavior can also be evidenced in the loss functions of both networks, as the errors of training and test sets tend to be closer when regularization is considered. This was expected to happen, as the weight decay will aim to reduce the general variance of the model.

4.5 Testing the effect of regularization using dropout

In this experiment, we aimed to observe the effect of the dropout technique as a *regularizator* for a deep network. We will train the same architecture as in the past experiment (800-800-800) with a dropout of 0.5 in all the fully connected layers (as suggested by the original paper (Hinton, 2012)).

From the results obtained, which are summarized below, we can observe almost no difference in the

performance of a 800-800-800 network trained without regularization and using dropout at 0.5. Even when the test accuracy was slightly higher in the dropout implementation (and maybe with less lumps), the graphics of the accuracy/loss over time seem very similar.

		Accuracy			
Architecture	**Activation function**	**Training**	**Dev**	**Test**	**Variation training vs test**
800-800-800	Dropout	99.89%	98.07%	98.16%	**-1.7%**
	Weight decay	99.18%	97.94%	97.88%	**-1.3%**
	No regularization	99.89%	98.11%	98.05%	**-1.8%**

Table 4. Accuracy for training, dev and test sets for a 800-800-800 deep network trained with and without regularization (weight decay and dropout)

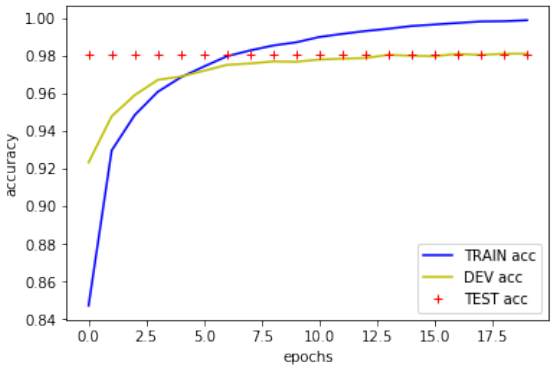

Figure 13. Network without regularization

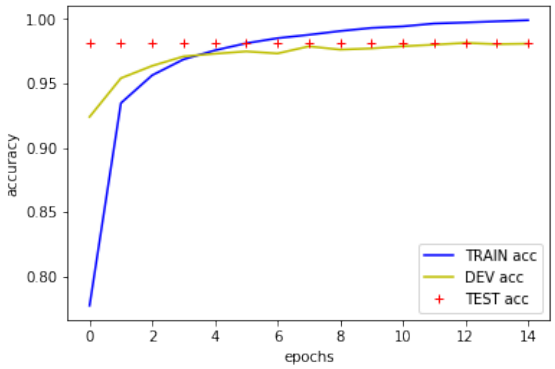

Figure 14. Network with dropout

Figure 15. Accuracy for training, dev and test sets for two 800-800-800 deep networks trained with and without regularization

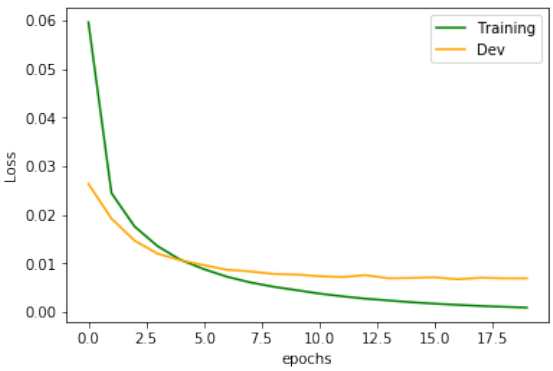

Figure 16. Network without regularization

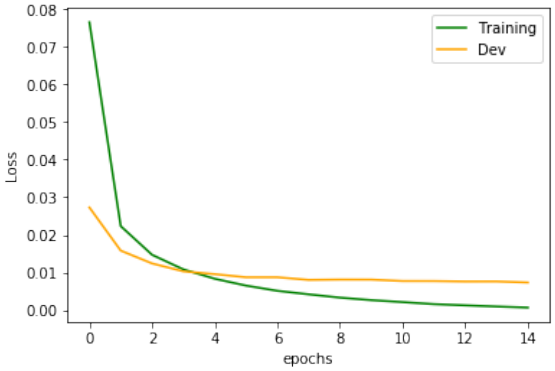

Figure 17. Network with dropout

Figure 18. Training and dev loss for two 800-800-800 deep networks trained with and without regularization

5 Final notes

- For the MNIST data set, almost all the deep network architectures that we trained performed well in terms of the desired level of fitting to the training data (no overfitting evidenced) and also in their accuracies, which were always above 95%.
- Nevertheless, we found that the batch size has a profound effect on the accuracy, so the larger it was, the more disturbed the accuracy ended up. Although it is well-known that big batches of data usually lead to poor generalization, still we cannot know why this happens[1].
- The convergence speed of almost all the networks suggested that we could end up with a high accuracy (over 95%) just by reaching the fifth or sixth epoch.
- With regards to overfitting, we were not able to replicate a training scenario in which we could observe this phenomenon explicitly happening. We conclude this because the training and test accuracies were always very similar and close to each other, and they increased consistently and proportionally over time (while their losses decreased in the same manner). Also, for all of the experiments, the average loss of training and test sets were not that far from each other. Moreover, in no experiment we could get to train a network to reach 100% of training accuracy, even when the number of epochs was greater than 100.
- After some research, we found some online experiments that got to train overfitted neural networks on the MNIST data set, and one of the questions that have arisen is: *Why, despite over-fitting, does a neural network trained to the MNIST data set still generalize apparently well in terms of accuracy?* With regard to this, an answer offered by Neil Slater (a computer scientist in the United Kingdom) seems quite interesting:

  "*... MNIST examples have relatively smooth transitions between classes in feature space, and that neural networks can fit to these and interpolate between the classes in a "natural" manner given NN building blocks for function approximation - without adding high frequency components to the approximation that could cause issues in an overfit scenario.*

  It might be an interesting experiment to try with a "noisy MNIST" set where an amount of random noise or distortion is added to both training and test examples. Regularized models would be expected to perform OK on this dataset, but perhaps in that scenario the over-fitting would cause more obvious problems with accuracy[2]."

  From our perspective, the overfitting problem happens due to several factors, which may include for instance, the number of training samples, the consistency of those samples and their dimensions, the intrinsic dimensionality, amongst others. In this case, the MNIST data set is almost linearly separable (as explained by some experts, who say that the first 2 principal components or only one pixel can separate the digit classes well)[3], which would explain that the complexity of the problem is not that high so that we need to worry about fighting against overfitting that much.

  Intuitively, the concept of overfitting can be also understood as the mix of large models and unrelated features. Even when we tried to implement complex models on the MNIST data, we did not get to beat the easiness of the features relationship.
- The previous point can explain why we did not experience significant improvements in terms of accuracy and loss optimization when we implemented dropout and weight decay in a relatively complex network of three layers with 800 hidden neurons that was trained with a large batch size.

[1] More info at: https://medium.com/mini-distill/effect-of-batch-size-on-training-dynamics-21c14f7a716e

[2] More info at: https://datascience.stackexchange.com/questions/19874/why-doesnt-overfitting-devastate-neural-networks-for-mnist-classification

[3] More info at: https://twitter.com/drob/status/869991240099549185