UNIVERSITÉ DE GENÈVE

FACULTÉ DES SCIENCES

UNIVERSITY OF GENEVA

Classification, weight sharing, auxiliary losses
Mini deep-learning framework

Authors
Arnaud SAVARY
Jérémie GUY

Professor:
François FLEURET

December 17, 2021

Contents

Chapter 1

Classification, weight sharing, auxiliary losses

1.1 Presentation

In this project, the objective was to test 3 different architectures to compare two digits given in two-channel images. We used PyTorch as the only framework for the neural networks[1] and wanted to show the impact of using weight sharing and auxiliary losses. We created a Jupyter notebook to have a comprehensive structure and to allow the execution of specific parts of the code.

1.2 Implementation

1.2.1 CNN

The first network we implemented is a Convolutional Neural Network. The shape of its inputs is $N * 2 * 14 * 14$, where N is the size of the mini-batch, 2 comes from the fact that we use pairs of images, and $14 * 14$ is the size of each image.

The figure 1.1 and 1.2 shows the structure of the network.

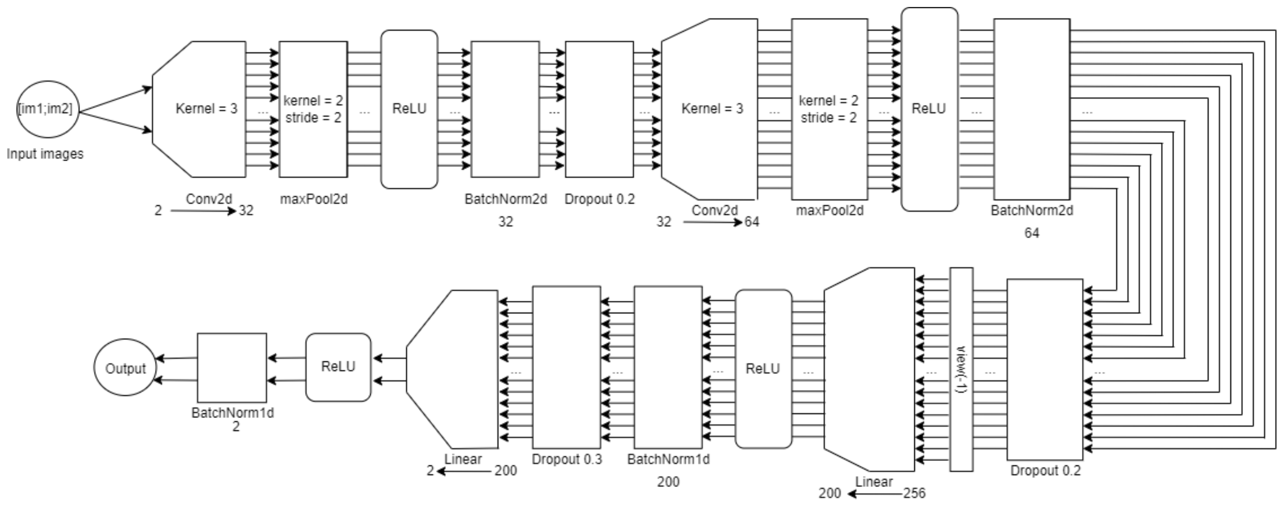


Figure 1.1: Graphical representation of the CNN network structure

[1]numpy is also imported but only to compute mean and standard deviation on lists

Conv2d(2, 32, kernel_size=3)
max_pool2d(kernel_size=2, stride=2)
ReLu()
BatchNorm2d(32)
Dropout(0.2)
Conv2d(32, 64, kernel_size=3)
max_pool2d(kernel_size=2, stride=2)
ReLu()
BatchNorm2d(64)
Dropout(0.2)
Linear(256, 200)
ReLu()
BatchNorm1d(200)
Dropout(0.3)
Linear(200, 2)
ReLu()
BatchNorm1d(2)
Dropout(0.3)

Figure 1.2: CNN

To enhance the results, we used two methods shown in the course: batch normalization and dropout. The first allows the code to run faster and to be more stable by normalizing the inputs. The second one removes a random part of the parameters to prevent over fitting. We tested multiple structure and obtained the best results when we used two different dropouts: one with a 0.2 probability and one with 0.3.

All the layers are created when the object is first created, and they are linked together and called once the "forward" method is called.

The network ends with 2 nodes, 2 possibles classes where the first one represent the case where the first digit is bigger than the second one and the opposite for the second class. It is important to note that the numbers are never "recognized", the network only learns which image represent a bigger digit than the other.

1.2.2 Weight sharing

The second network is a CNN with weight sharing. It uses the same as inputs as the CNN but divides the pairs of images in the dataset to obtain two $N * 1 * 14 * 14$ sets. They are then both called on the previously defined CNN but with only 1 node instead of 2 as input and 10 nodes instead of 2 as output. The two outputs are then merged together and we add 3 Linear layers before proceeding with an output of 2 classes.

The figure 1.3 shows the structure of the network.

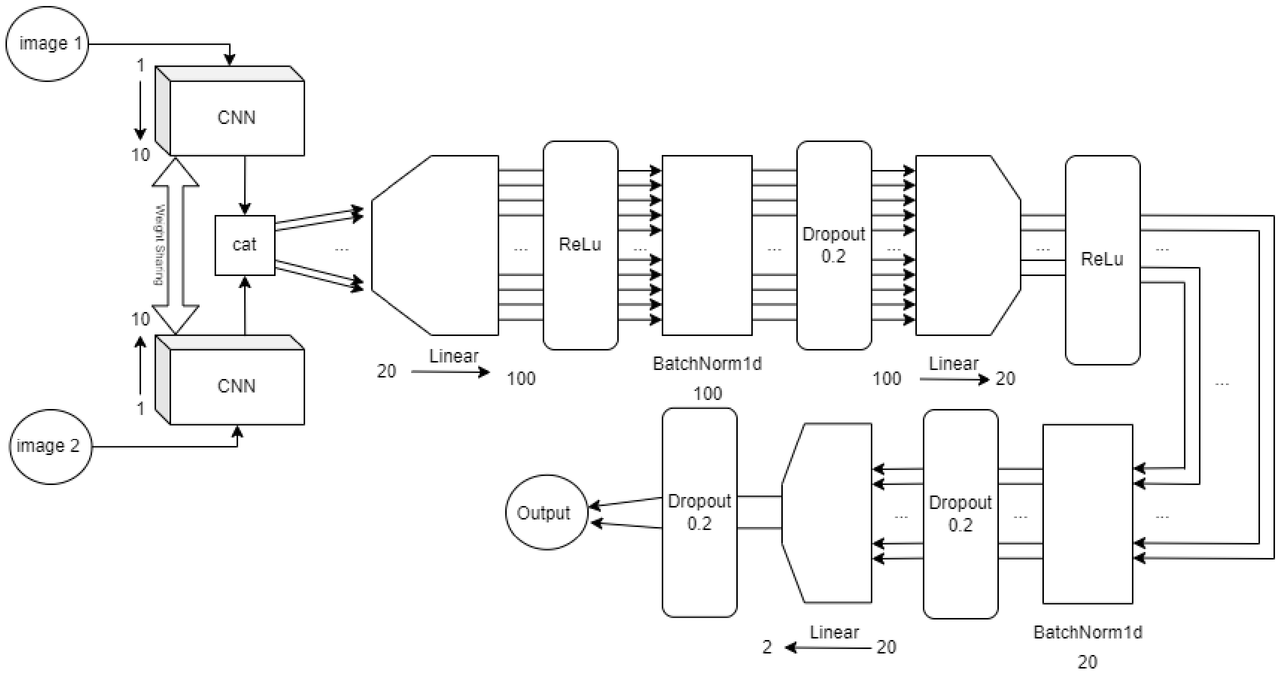


Figure 1.3: Graphical representation of the network structure with weight sharing

2 * Previously defined CNN(1,10)
Linear(20, 100)
ReLu()
BatchNorm1d(100)
Dropout(0.2)
Linear(100, 20)
ReLu()
BatchNorm1d(20)
Dropout(0.2)
Linear(20, 2)
ReLu()

Figure 1.4: Weight Sharing

1.2.3 Auxiliary losses

In this final model we use auxiliary losses to enhance the result. We use exactly the same model as the previous one, the only difference being that we output the losses of the 2 CNN. The main difference lays in the implementation of the training function. Instead of computing the loss directly on the output of the model, we compute the auxiliary losses on the outputs of the CNN called with 1 image and the main loss on the output of the MLP[2] that was called with the concatenated outputs of the CNNs. Those 3 losses are then merged with this weighted sum:

$$loss = \alpha * auxiliaryLoss1 + \alpha * auxiliaryLoss2 + \gamma * mainLoss$$

α and γ are two constants that we fix in the model tuning.

[2]Multilayer perceptron

1.2.4 Model training, tuning and testing

All the models are trained, tuned and tested with the same functions. The only difference relies in the computation of the loss for the auxiliary losses model. Each model is trained with 25 epochs and batches of size 50. We chose the cross entropy for the criterion and the SGD optimizer since it's a classification task. Once the model is trained, we can call the "compute_nb_errors" functions to compute the number of errors made by the model on the training and/or testing set. This functions runs the given model with batchs of size 50 and compare the prediction of the model with the target. The errors are then summed up, returned and used to evaluate the model, during the tuning or the final testing.

To find the best parameters, we first tried by hand and then implemented a model tuning function with the grid search method. This functions creates a model with the given parameters and trains it on a set of 500 pairs of images, computes the number of errors made on a validation set of 500 pairs of images and stores it in a list. This process is repeated 5 times and the mean of the list is compared with the best mean found so far. If it's better, the parameters are saved and returned once all the combinations of parameters have been tested.

We tried 4 values for the learning rate: [5e-2, 1e-1, 5e-1, 1] and 4 for alpha and gamma: [0.2, 0.5, 0.8, 1].

The models are then tested with the best parameters and run 15 times. The number of errors on the training and testing set are stored in lists and their mean and standard deviation are printed.

1.3 Results

The means and standard deviations we got on the models can be seen in the figure 1.5

	CNN	WS	AL
Train error mean	$6.67 * 10^{-4}$	$2.60 * 10^{-3}$	$1.84 * 10^{-2}$
Test error mean	$1.70 * 10^{-1}$	$1.23 * 10^{-1}$	$1.52 * 10^{-1}$
Train error std	$1.01 * 10^{-3}$	$1.85 * 10^{-3}$	$1.13 * 10^{-2}$
Test error std	$1.63 * 10^{-2}$	$1.30 * 10^{-2}$	$8.98 * 10^{-3}$

Figure 1.5: Results

On the one hand we observe that the weight sharing greatly improves the average error, from 17% to 12.3%. The standard deviation is also smaller, which means that the results are more stables. On the other hand, the auxiliary losses give better results than CNN but worse than Weight Sharing. The standard deviation is bit lower for AL than WS, making it the most stable model, but it has very poor interest since the worst result for WS is better than the best result for AL.

1.4 Improvements

The first improvement that we could work on is the Auxiliary Losses, which should give better results than Weight Sharing. It may come from the alphas and gammas chosen, but we think it comes from the number of layers of the the CNNs. The task to discriminate between 10 classes is a lot harder than the binary greater/smaller and would therefore need more layers. We can imagine that the class predictions made by those CNN are actually wrong and influence badly the final loss.

The second improvement is the time of execution. While the models run pretty quickly (around 1 minute for the testing), it is still too slow for the grid search used in the model tuning. We test $3 * 4 * 4$ combinations of parameters for AL, each one on 5 runs, and it ends up costing us a lot of time.

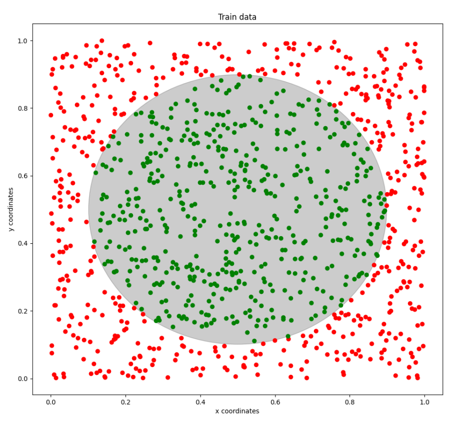
Chapter 2

Mini deep-learning framework

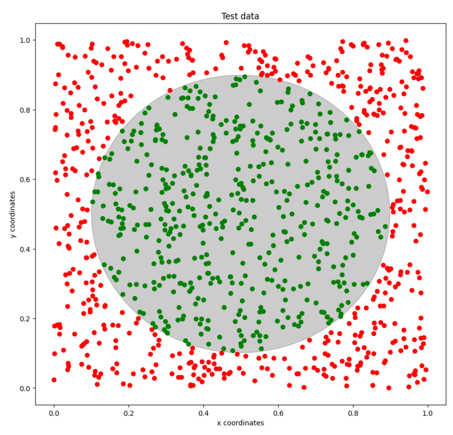
2.1 Presentation

The objective of the second part of the project is to generate a deep-learning framework. The principle is the same as any neural network : we will get a set of data points which have been classified using a criteria, train the model on it and finally make predictions using said trained model.

The data in this case will be a set of 1000 points in a 2D space with each coordinate being generated using a random normal distribution on [0,1]. The classification criteria is a circle of diameter $\frac{1}{\sqrt{(2\pi)}}$ centered on [0.5,0.5] : if a point is inside the vicinity of the circle then we classify it as '1' else as '0'. The generated data to process is represented in Figure 2.1.



(a) Training Data



(b) Testing Data

Figure 2.1: The red dots are outside the circle (class '0') and the green ones are in (class '1')

2.2 Implementation

The implemented network will be working using a basic sequential structure of modules. Each module implements two methods : forward and backward. They will be used to make the forward pass and back-propagation steps while training the model. We first defined a base module (Sequential) that will chain all the sub modules, then our main layer module (Linear) as well as 2 activation functions modules (ReLU, TanH). We finally implemented a criterion: MSE loss and an optimizer: SGD.

We implemented every module and model structure using only the *torch.empty* tensor from the library *torch*. We import as well *math* and *numpy* to compute means, square roots and get pi. Every plot has been made using *Matplotlib*.

2.2.1 Structure

The model is composed of 4 layers: 1 input layer and 3 hidden layers. Every layer is connected to one another using one of the implemented activation functions. The whole structure is then encapsulated inside the Sequential module. We added the option to create and run batches of data instead of just one point to speed up the computation.

2.2.2 Sequential

Sequential is the module that links the whole network together. It links the classes together as well as create the sequence of the layers and activation functions. This allows in turn to apply the **forward()** and **backward()** methods: **forward()** saves the sequence to propagate the signal in the correct order in the network with respect to the dimensions of each layer by calling the **forward()** method of the current module. **backward()** works in the same way as forward but in the reverse order to back-propagate. It will also save and update the newly computed parameters.

2.2.3 Linear

Each of the layers (hidden and visible) is made using a Linear module with some input and output sizes. Using weights and a bias, it will compute the asked number of output given a certain input number. We defined the layers ins and outs as such:

- The first layer (visible) has 2 input (the two coordinates of a point) and 25 outputs.
- The two first hidden layers both have 25 inputs and output
- The last hidden layer has 25 outputs and only 1 output which corresponds to the classification of the point (between 0 and 1)

Figure 2.2 illustrates the layers structure.

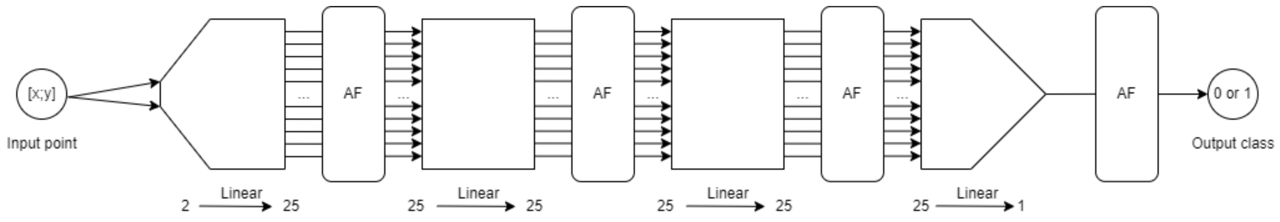


Figure 2.2: Graphical representation of the network structure

Each of the Linear layers has a set of parameters (weights and bias) that needs to be initialized. Depending on the activation function of the layer we used two different initialization technique based on the activation function used. In both cases the weights and bias are initialized using a normal distribution of mean 0 and a standard deviation σ. The value of σ will change according to the function:

- if the activation function is TanH : we used the Xavier initialization which computes the standard deviation as $\sigma = \sqrt{\frac{1}{N}}$ with $N = LayerInputSize + LayerOuputSize$
- if the activation function is ReLU : we used the He initialization which computes the standard deviation as $\sigma = \sqrt{\frac{2}{N}}$ with $N = LayerInputSize + LayerOuputSize$

2.2.4 Criterion

To evaluate the performances of our model with used the Mean Squared Error criterion: after finishing the forward pass, the model is evaluated with the loss MSE. The evolution of the loss MSE is illustrated on Figure 2.3.

2.2.5 Optimizer

We implemented an Optimizer to better tune the parameters when training the model based on the Stochastic Gradient Descent. The optimizer will update the weights using the learning rate. We added as well the option to use weight decay so that the further the model is tuned, the less the weights learn: this is a way to reduce oscillation in the results as well as decrease the risk of over-fitting the model.

2.3 Results

To yield the best results we fine tuned the parameters using the grid search method. We test 3 learning rate values [1e-3, 5e-3, 1e-2] and 2 weight decay [False, 1e-6]. We tune as well the activation functions: we test for each layer the two activation functions [TanH, ReLU]. The tuning runs the model 5 times for each set of parameters on 100 epochs with batches size of 50.

The best parameters obtained are : lr = 0.005, wd = $1 * 10^{-6}$, activation functions = ['TanH', 'ReLu', 'ReLu', 'TanH']

Using these parameters, we trained and tested the model with 250 epochs and batch sizes of 50. We obtain the loss shown on Figure 2.3:

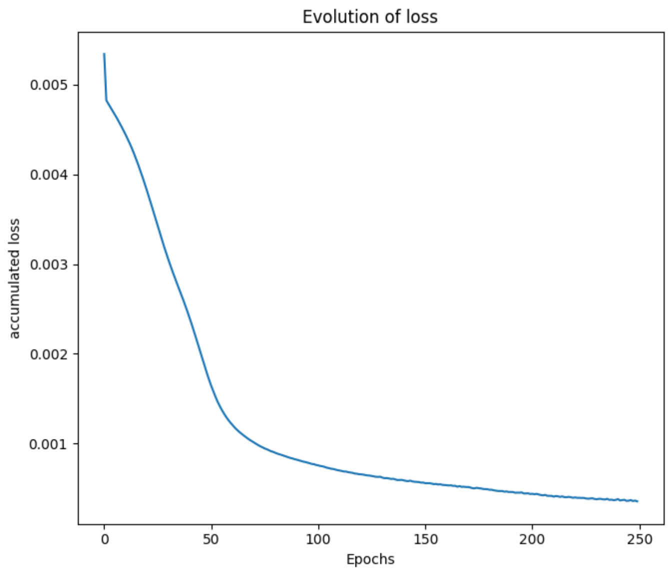


Figure 2.3: Evolution of the loss function on the number of epochs

The classification of the points inside of the circle is illustrated on the Figure 2.4.

We get approximately 10% of error at the end of the classification for the test set, and 1.5% on the train set.

2.4 Improvements

To get even better results, we could have implemented a few new modules to add to the grid search: cross entropy loss as error criterion, ADAM optimizer instead of the Stochastic Gradient Descent and some other activation functions such as softMAX or sigmoid.

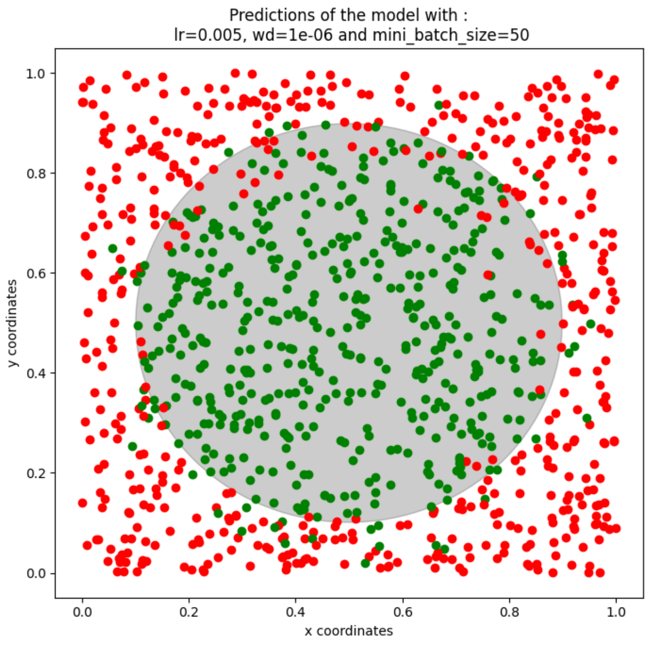


Figure 2.4: Predicted classification of the points. The red dots have been classified outside the circle and the green ones inside