Deep Learning: Homework 2

Federico Betti, Gauthier Mueller

May 17, 2018

Contents

1 Introduction **1**

2 Core Implementation Details **1**

2.1 Backpropagation Algorithm Implementation 2

2.2 Dense Layer 3

2.3 Optimizer 4

2.4 Loss 4

3 Extra-Feature **4**

4 Test & Results **5**

1 Introduction

The aim of this project is to create a freamwork for the development and training of Artificial Neural Networks. We decided to import, as requested, only the FloatTensor and LongTensor classes from Torch and therefore we decided to base all the calculation within the freamwork on the functions that were available in those classes. We tried to use the numpy library as little as possible in the computational phase, trying to apply the concepts learned in the first part of the course on the use of the Torch library and all the methods about tensors that it holds.
The starting model used was the one recommended by the assignments, but changes and improvements were made during development. The final version of the framework gives the possibility to create a multylayer perceptron of any size and depth. We have tried to create a modular structure by using the **Sequential** container to provide maximum flexibility to the programmer. Each Sequential layer must consist of a **Dense** layer and an activation function; the linear function can be used in case you want to propagate the output of the Dense layer without any modification.
The activation functions implemented are:

- Linear
- Tanh
- Sigmoid
- Relu
- Softmax

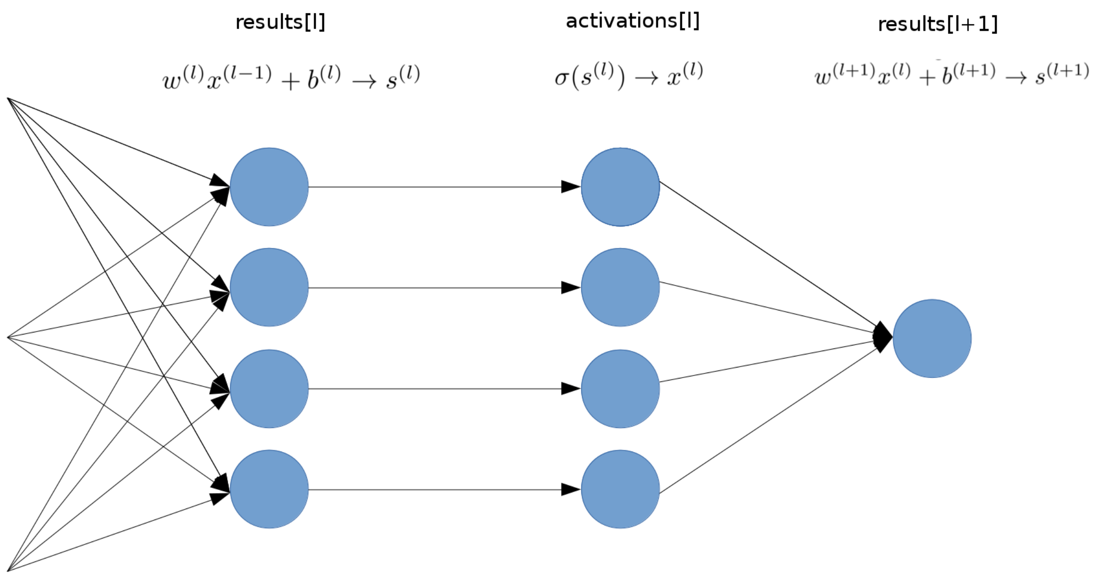
In this embryonic phase of the project only the **Mean Squared Error** (MSE) loss has been implemented and is sufficient to satisfy the requests. However, especially if classification problems are considered, a future implementation of the Cross Entropy loss would be desired.
We chose to implement, as required, only **Stochastic Gradient Descent** (SGD) as optimizer, in a variant that uses mini-batches.
As will be explained in more detail in the next section we have implemented the framework in order to have the ability to use mini-batches and not be forced to use only one sample at a time; in fact thanks to the use of matrix calculation provided by the Torch library, everything is optimized and executed very quickly.
During the development of this first framework we decided to implement some additional features that we think will be very useful for the end user. These new functionalities will be presented in more detail in the third section.

2 Core Implementation Details

As already mentioned, we have used a structure of classes and methods similar to the one recommended in the assignments with some little modifications trying to reflect the structure we had set ourselves. Our model is based on the succession of layers within a Sequential container, which consist respectively of a Dense module and an activation function. We made the implementation choice to save the weights and biases inside the layer object, while gradients will be stored in the container. This is because during the backpropagation algorithm, which in our case takes place in the Sequential container, it is necessary to refer to all gradients in the network, while the subsequent weights update, performed by the optimizer, can be done separately.
The core of our framework, as you can see from these lines, is the Sequential module. When it is instantiated, the container receives as input all the Dense layers of the network. It add them in an ordered list and consequently initialize lists to store weight (dw) and bias (db) gradients.

```
def __init__(self, *args):
        super(Sequential, self).__init__()
        self.layers = [e for e in args if e.is_layer()]
        self.num_layers = len(self.layers)
        self.db = [FloatTensor(layer.bias.shape).zero_() for layer in self.layers]
        self.dw = [FloatTensor(layer.weigths.shape).zero_() for layer in self.layers]
        self.results = []
        self.activation = []
```

Results and Activation lists will play an important role in the algorithm's evolution, as will be seen below. Result list will contain the outputs of the dense layers during the forward pass, while Activation will save the results of the corresponding activation functions. For an implementation issue the input layer during the backpropagation algorithm will be treated as a result of activation function, hence stored in the Activation list. Thinking about it, it sounds normal since it is what enter as input in the first layer of the net and it should be treated exactly the same as other input layers.


results[l]
$w^{(l)}x^{(l-1)} + b^{(l)} \to s^{(l)}$
activations[l]
$\sigma(s^{(l)}) \to x^{(l)}$
results[l+1]
$w^{(l+1)}x^{(l)} + b^{(l+1)} \to s^{(l+1)}$

Thanks to the figure above our model structure can be easily understood. In fact we have separated the Dense layer that computes results from the activation layer that computes activations.

2.1 Backpropagation Algorithm Implementation

```python
def backward(self, loss, target, mini_batch):

    db = self.db
    dw = self.dw

    x_lt = self.activations[-1]
    x_lb = self.activations[-2]
    s_lt = self.results[-1]
    dsigma = self.layers[-1].activation.backward(s_lt)

    dldx = loss.prime(x_lt, target)
    dlds = dsigma * dldx

    db[-1].add_(dlds.sum(0))
    dw[-1].add_(x_lb.t().mm(dlds))

    for i in range(2, self.num_layers+1):
            x_lt = self.activations[-i]
            x_lb = self.activations[-(i + 1)]
            s_lt = self.results[-i]

            dsigma = self.layers[-i].activation.backward(s_lt)
            w = self.layers[-i + 1].weigths

            dldx = (dlds).mm(w.t())
            dlds = dldx * dsigma

            db[-i].add_(dlds.sum(0))
            dw[-i].add_(x_lb.t().mm(dlds))
    return dw, db
```

During the algorithm implementation we decided to keep the computation of gradients of the output layer separate from those for the internal layers because they are slightly different. To fully understand and correctly implement the backpropagation algorithm we relied mainly on the slides provided by the course.

$\sigma'\left(s^{(l)}\right)$	$\left[\frac{\partial \ell}{\partial x^{(L)}}\right] = \nabla_1 \ell\left(x^{(L)}\right)$	$\left[\frac{\partial \ell}{\partial x^{(l)}}\right] = \left(w^{(l+1)}\right)^T \left[\frac{\partial \ell}{\partial s^{(l+1)}}\right]$	$\left[\frac{\partial \ell}{\partial s^{(l)}}\right] = \left[\frac{\partial \ell}{\partial x^{(l)}}\right] \odot \sigma'\left(s^{(l)}\right)$
dsigma	dldx output layer	dldx inner layer	dlds

- **disgma**: It contains the activation function derivative made with respect to the output of the previous dense layer.
- **dldx (output layer)**: It contains the derivative of the loss function, starting point of the algorithm. This step compares the final target and the network output to calculate the network output gradient.
- **dldx (inner layers)**: It is the derivative of the loss made with respect to the output of a specific layer. To calculate it, multiply the loss derivative obtained with respect to the output and the weights of the next layer. The algorithm, as you can see from the for loop that cycles in the opposite direction, starts from the network output and propagates back to the first input. For this reason when we are in an inner layer the derivatives made with respect to the next layer have already been calculated and therefore they can be used to propagate backwards the gradients.
- **dlds**: It is the gradient of the output of the dense layer with respect to the final loss. It is calculated by multiplying the loss derivative of its own activation layer (the next one) and its derivative. This operation is the cause of vanishing gradient problem: if the network is too deep you risk that both factors become very close to zero, therefore dlds and all the gradients in previous layers would be stick to zero. This would lead to a dead net that doesn't change its weights and consequently stops learning.

The code has been properly commented in the source code; it can be useful to check also what support functions do, since they are not well investigated in the report.

2.2 Dense Layer

```
def __init__(self, in_neurons, out_neurons, activation):
    super(Dense, self).__init__()
    self.in_neurons = in_neurons
    self.out_neurons = out_neurons
    self.activation = activation
    self.weigths = FloatTensor(in_neurons, out_neurons).normal_() * math.sqrt(2.0 / in_neurons)
    self.bias = FloatTensor(out_neurons).zero_()
    self.error = 0

def forward(self, x):
    exceptions_check.checkFloatTensor(x)
    return x.mm(self.weigths).add(self.bias)

def backward(self, input):
    return input
```

The definition of a Dense Layer is quite standard. We made the implementation choice to save the activation, taken as a parameter, in a specific field. For this reason the backward pass of this layer replicates only the input, while the derivative of the activation function will be executed during the backward pass calling activation.backward(input).
The initialization of the weights was problematic: at first the initialization was only done with a

normal distribution, but we realized that with certain initial conditions the gradient lay down on a local minimum and blocked after a few epochs. At the end, after a detailed search on the web for weights initialization techniques, we found the solution to multiply by $\sqrt{\frac{2}{in\_neurons}}$. This made the initial weights smaller and eliminated the initial problem once and for all.

2.3 Optimizer

```
def step(self, model):
        for i, layer in enumerate(model.layers):
                layer.weigths -= self.lr * model.dw[i]
                layer.bias -= self.lr * model.db[i]
```

The step method is the core of the optimizer: since we have implemented only the SGD the updates on weights and biases are performed by subtracting the gradients multiplied by the learning rate, previously saved as the initial parameter of the optimizer.
We have given the possibility to update the learning rate during the training procedure. It can be very useful in more complicated problems when you want to decrease the learning rate after a specific number of epochs to increase precision and regularity.

2.4 Loss

```
def apply(self, v, t):
        return (v - t.resize_(v.size())).pow(2).sum()

def prime(self, v, t):
        return 2 * (v - t.resize_(v.size()))
```

Apply and Prime are the two core methods of a Loss class. The former one computes loss instead the last one works on the derivative of the loss.
$.resize(v.size())$ must be inserted to handle specific 1D target tensors that can create dimensional problems.

3 Extra-Feature

We have added some extra features to the framework that we think could be useful for an easier and more complete use, even in such a prototype version.

- **Softmax**: We decided to implement the Softmax activation function because it is the one that best fits a classification problem since it relates all the output layers, creating a probability vector. However, in the final solution it was not used as it achieves the best performance with the CrossEntropy Loss; with a MSE Loss a Sigmoid output function still has better results.
- **Compute History**: This framework feature allows you to keep track of epoch-wise loss and accuracy for both training datasets and validation datasets. It can be very useful to see the network trend, to study overfitting and to compare different models. Since at any time it has to calculate loss and accuracy for both datasets, it slightly slows down the training process so we have inserted the option to disable it.
- **Real-time Graphic Visualization**: We have provided the possibility of having an epoch-wise graphic display of the network trend, showing which are the correct and wrong predicted samples. This tool can be very useful because it allows you to see the evolution of the hyperplans that delimit the classes in each epoch and consequently it gives you a clear visual example of how the process evolves.

Although this is only a starting project we discussed what future implementations could be easily developed to add useful functionality to the framework:

- **CrossEntropy Loss**: This is another type of loss function that is suitable for classification problems and can be used easily and with excellent results with the softmax activation function. We believe that this is the first thing to develop in the future.

- **Dropout**: Adding dropout layer to the framework should not be difficult to implement and could be very useful to tackle more complicated problems, where the risk of overfitting increases significantly with the noise in the data.
- **Other optimizers**: Several methods of optimization alternative to SGD have been invented and often have better performance (Adam, Adadelta...). Therefore, they could be objects of future expansions of the framework.

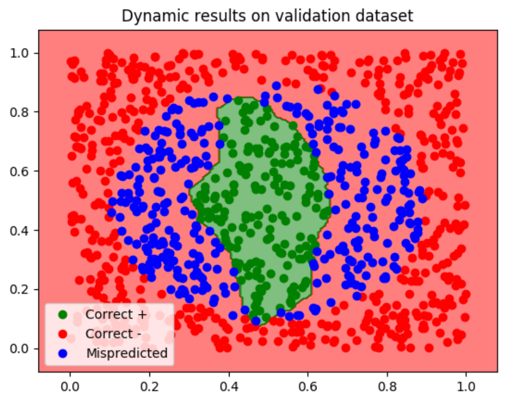
4 Test & Results

The framework test was done with the model indicated in the assignemtns. We decided to create three different datasets: training, validation and test. The first is used for the training procedure, the second to check improvements of the training results and to check for possible overfitting, the third to check the final results.
The test was performed with the model described below, using the compute history and real-time graphic visualization functionality.

```
model = nn.Sequential(
        nn.Dense(2, 25, F.ReLU()),
        nn.Dense(25, 25, F.ReLU()),
        nn.Dense(25, 25, F.ReLU()),
        nn.Dense(25, 2, F.Sigmoid())
)
```

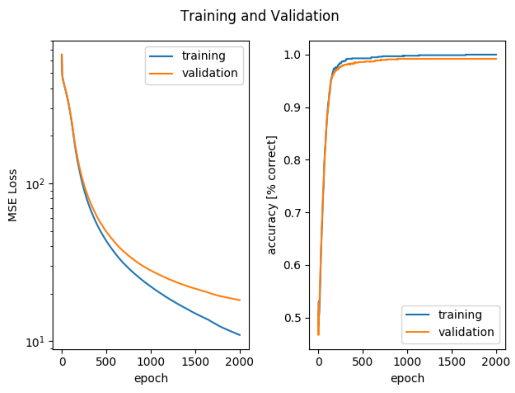
The model consists in a three-layer network with ReLU activation functions in the hidden layers and a Sigmoid as output function. Sigmoid function fits perfectly with a 2-classes classification problem because the result is within [0 - 1].

The 'Dynamic results on validation dataset' picture represents the performance of the network after 30 epochs on validation dataset. The green area represents the zone where the network predicts 1, the red area is where it predicts 0. Green dots are samples 1 that are correctly predicted by the net, red dots are the correct 0 samples, blue ones are mispredicted samples.



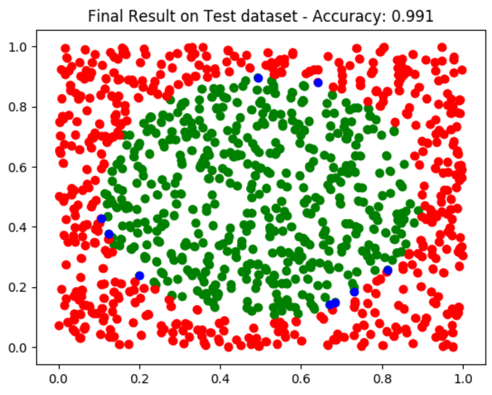
dlds

This figure shows the network trend in terms of loss and accuracy, both for training and validation dataset. The graph shows a divergence in the losses around the 2000th epoch; this is a first sign of overfitting.



dlds

The last image shows the network performances on the test dataset. As written in the caption, it perforates 99.1% on the test, in fact only the 9 blue dots on a dataset of 1000 samples are labeled incorrectly.



dlds