Deep Learning Project 2: Report

Raphael Bonatti
Lena Csomor
Marijn van der Meer
Department of Computer Science, EPF Lausanne, Switzerland

I. Introduction

In the scope of this project, we implemented several popular Machine Learning modules and activation functions using only PyTorch's standard tensor operations and the math library. All modules run with `autograd` globally off. Overall, these modules were combined into a model trained on points sampled uniformly from $[0, 1]^2$ with labels 1 if they are within a disk centered at $(0.5, 0.5)$ with radius $r = 1/\sqrt{2\pi}$ and 0 otherwise. We compared the performance of our architecture to the same model built with pre-implemented PyTorch modules.

II. Modules implementation

For our implementations we used the structure suggested in the project description which is to define a `Module` class and have all modules inherit from it.

```python
class Module(object):

    def forward(self, *input):
        raise NotImplementedError

    def backward(self, *gradwrtoutput):
        raise NotImplementedError

    def param(self):
        return []
```

A. Linear module

The `Linear` module applies a linear transformation of the form $y = A^T x + b$ to its input. For familiarity purposes, the signature of the `init` method is the same as the one in the PyTorch framework. It initialises all weights and bias vectors to values sampled uniformly between $[0, 1]$. Our version of the linear module also has a bias flag, which corresponds to enabling or disabling the bias term. To update the module weights during training, we store the input, the weights and the gradients with respect to the parameters i.e., weight and bias. The input is stored during the forward pass so that it can be used in the backward pass to compute the gradient with respect to the weights. We can access the module's parameters (weight and bias) and its gradients with respect to the parameters by calling the `param()` method. This method returns a dictionary, whose values can be easily accessed with the keys `weight` and `grad`. For the purpose of our task, this module can also take a tuple as an input and it will output a tuple of the same size. In that case, a tuple of gradients will also be stored.

B. Tanh module

The `Tanh` module is an activation function that applies the tanh function element-wise to its input. It can be used as a replacement for the sigmoid activation function, as it outputs numbers in $[-1, 1]$. This module does not have a `param()` method as it is an activation function and has no specific parameters. It also does not have weights that need to be updated. However, it still needs to store the input to compute the backward pass. Like the linear module, this module accepts tuples.

C. ReLU module

The `ReLU` module applies a piece-wise linear ReLU function element-wise to its input. ReLU has become a popular activation function for especially MLPs and CNNs because it is supposed to overcome the vanishing gradient problem [1].

$$f(x) = max(x, 0) \tag{1}$$

$$f'(x) = \begin{cases} 0 & x < 0 \\ 1 & 0 \leq x \end{cases}$$

Similarly to `Tanh`, this module accepts tuples as well and does not have a `param()` method as it is an activation function. `ReLU` also stores the input to compute the backward pass.

D. LeakyReLU module

The `LeakyReLU` applies a parametric ReLU function to the input, also known as Leaky ReLU, which does not set the gradient to 0 for inactive units, but allows it to stay at a small, positive value. Its parameter α is usually chosen between 0 and 1. The main idea of Leaky ReLU is to let the gradient be non zero and recover during training eventually, so that we ideally avoid a vanishing gradient problem. We decided to implement Leaky ReLU so that we could compare its effect on our model compared to standard ReLU.

$$f(x, \alpha) = max(x, \alpha * x) \tag{2}$$

$$f'(x) = \begin{cases} \alpha & x < 0 \\ 1 & 0 \leq x \end{cases}$$

E. Sequential module

The `Sequential` module is a container for sub-modules and allows us to assemble different modules and activation functions. It adds the modules passed to the constructor in the same order to the model. The `Sequential` module feeds the outputs of sub-modules as inputs to the next sub-modules during forward pass and passes the gradients in the reverse order during backward propagation. The module also works for tuples as inputs.

```python
class Sequential(Module):
    def __init__(self, modules):
        self.modules = modules

    def forward(self, *input):
        x = input
        n = len(self.modules)
        for i in range(n):
            x = self.modules[i].forward(*x)
        return x

    def backward(self, *gradwrtoutput):
        x = gradwrtoutput
        n = len(self.modules)
        for i in range(n):
            x = self.modules[n-i-1].backward(*x)
        return x

    def param(self):
        return [module.param() for  module in self
.modules if module.param() != []]
```

III. The Model

For simplicity reasons and to be able to use the modules we had at hand, we implemented our task as a regression problem rather than classification. This was mainly motivated by the fact that we had to train our model with Mean Squared Error (MSE), which is usually considered as a bad choice for classification. Furthermore, this way there was no additional function needed like a sigmoid or softmax to transform output probabilities into a class. For further work, we would have implemented Binary Cross Entropy to be able to transform the task into classification.

As instructed, we constructed a network with two input units, one output unit and three hidden layers of 25 units (Figure 1). As seen in Figure 1, we used ReLU (or Leaky ReLU) between the input layer and the first hidden layer. All other activation functions used were Tanh. We also tried another architecture where we changed the first ReLU to Tanh and a second architecture with more ReLU layers instead of Tanh. However, these models either gave worse performances or vanishing gradients.

A. Training

This model was trained using mini-batch gradient descent with mini-batches of size 4 and a learning rate of $1e^{-4}$. As an optimiser, we implemented stochastic gradient descent (SGD) and gradient descent (GD) but chose to use GD for the final implementation to take advantage of using tuples of tensors to create mini-batches. We experimented with bigger batch sizes but the performance was poor, so we kept a small batch size. The test and training set were both of size 1000 and normalised using z-score normalisation. Results of the training and test loss can be seen in Figure 2a. The final training and test loss are 0.25048. We see that both training and test loss are very close and that the model oscillates around a 0.25 loss for the last iterations.

We also trained a second model where we replaced the ReLU activation with a Leaky ReLU. But, as can be seen in Figure 2b, the performance is poorer for this activation function, with a final train and test loss of 3.32610. The amplitude of oscillations are also much bigger than for ReLU.

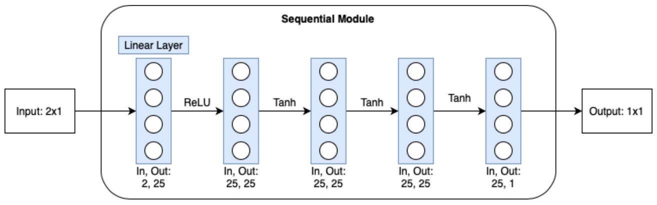
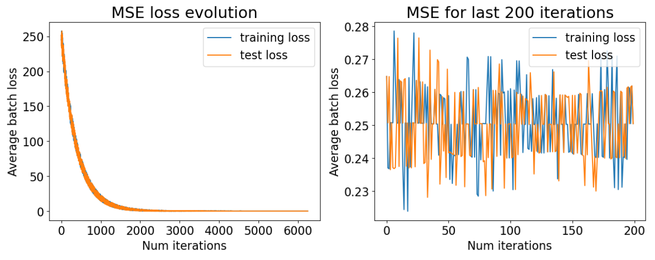
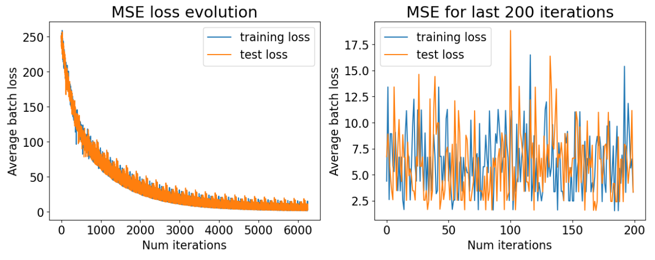


Figure 1: Visualisation of Model constructed of submodules. The network has two input units, one output unit and three hidden layers of 25 units



(a) Model with ReLU.



(b) Model with Leaky ReLU.

Figure 2: Visualisation the evolution of the training (blue) and test MSE loss (orange) of model constructed with our sub-modules (Figure 1). On the left is the MSE loss over all iterations and on the right for the last 200 iterations. The number of iterations is given by the number of epochs (25) multiplied by the number of mini-batches created from the dataset (250).

IV. Comparison to PyTorch

As a comparison, we created the same module with the pre-implemented PyTorch framework and trained it in the same way with MSE, a batch-size of 4 and learning rate of $1e^{-4}$. The only difference is that we had to use the SGD optimiser as normal GD, which is not implemented as an optimiser in PyTorch. But according to PyTorch documentation their SGD is actually a vanilla gradient descent.

Unsurprisingly, the performance of this implementation is significantly better than our model (Figure 3), with a final training and test loss of respectively 0.01765 and 0.01646. We also see that the model converges much faster and that there are almost no oscillations. We also notice that, like for our model, training and test losses are very close.

We suspect that, though SGD is described as a vanilla gradient descent in the documentation, it might still be differently implemented than a simple gradient descent and be one of the reasons that this model performs better. Furthermore, PyTorch being a very popular and professional framework, it is highly probable that other internal implementation differences make their modules faster and better.

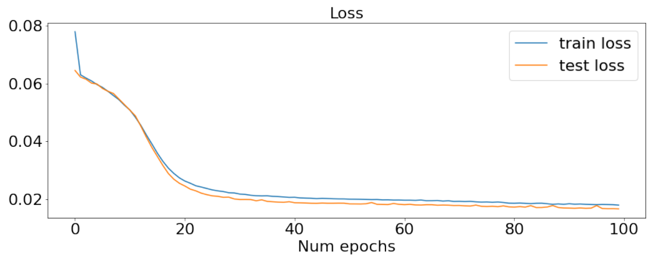


Figure 3: Visualisation of MSE training (blue) and test loss (orange) evolution over 100 epochs using an architecture constructed with standard PyTorch modules.

V. Summary

Overall, we created a model that performs a regression with a final loss of 0.25. To improve this and get closer to the loss of performance of the model that used official PyTorch modules, we could implement softmax and Binary Cross Entropy modules to formulate the task as classification. We could also implement different optimisers like Adam or SignSGD to aim to dampen oscillations and converge faster.

References

[1] J. Brownlee, "A gentle introduction to the rectified linear unit (relu)." [Online]. Available: https://machinelearningmastery.com/rectified-linear-activation-function-for-deep-learning-neural-networks/