EE-559 Project 2 – Mini deep-learning framework

Alexandre Reynaud (258402), Victor Taburet (261845), Vincent Rinaldi (239759)

I. Introduction

In this project, the objective is to implement a simplified version of pytorch which can use, at least, linear layers and a way to combine them sequentially, the ReLU and Tanh activations, the MSE loss, the SGD optimization algorithm. This library has then to prove it's operativeness through the implementation of a simple $2 \times 25^3 \times 2$ test model that uses SGD as optimization algorithm and MSE as loss function. As advised by the project guidelines, we decided to implement our "Mini-Torch" framework using modules taking the form of classes and supporting each at least a forward, backward and param function. Also, in order to have additional leeway when optimizing the test model and further the possibilities of our library, we decided to implement some extra components such as the MAE loss and the Sigmoid activation.

II. Implementation

Here follows a description of the implemented components and the functionalities they respectively support.

A. Main Components

This section describes the implementation of the modules forming our minimal viable product.

1) Layer Modules:

Linear: Simulates a fully-connected layer:

- __init__: Initializes the weight matrix W and bias vector b using the given layer input and output sizes.
- Forward: Given the input batch x, saves x (for further use) and returns $W \cdot x + b$.
- Backward: Computes the loss gradient with respect to W, b and x, $gradW$, $gradb$ and $gradx$, uses these $gradW$ and $gradb$ to apply a gradient step on w and b (SGD) then, returns $gradx$. N.B.: $gradx$ is computed using the x saved by the forward function.
- Param: Returns a list containing the current weight matrix W paired with the corresponding loss gradient $gradW$ and the bias vector b paired with the corresponding loss gradient $gradb$.

Sequential: Simulates a sequential concatenation of modules:

- __init __: Initializes the ordered list of layers of the sequence.
- Forward: For each layer, following the order of the layer list, applies the forward function of the layer to the current input in order to get the next one.
- Backward: For each layer, following the inverse order of the layer list, applies the backward function of the layer to the current gradient in order to get the next one.
- Param: Returns an ordered list containing the parameter/gradient pairs all the layers of the represented sequence.

2) Activation Modules:

ReLU: Simulates a rectified linear unit activation layer:

- Forward: Given an input batch s, saves s (for further use) and returns $max(s, 0)$.
- Backward: Using the loss gradient with respect to the output, grad, returns $grad \odot \nabla s \, max(s, 0) = grad \odot (s > 0)$. N.B.: $\nabla s \, max(s, 0)$ is computed using the s saved by the forward function.
- Param: Returns an empty list since there are no parameters.

Tanh: Simulates an hyperbolic tangent activation layer:

- Forward: Given an input batch s, saves s (for further use) and returns $tanh(s) = 1 - 2/(1 + e^{2 \times s})$. N.B.: $Tanh$ is computed using this form of the function in order to prevent numerical issues.
- Backward: Using the loss gradient with respect to the output, grad, returns $grad \odot \nabla s \, tanh(s) = grad \odot 4/(e^s + e^{-s})$. N.B.: $\nabla s \, tanh(s)$ is computed using the s saved by the forward function and the form $4/(e^s + e^{-s})$ in order to avoid numerical issues.
- Param: Returns an empty list since there are no parameters.

3) Loss Modules:

MSE: Simulates a mean squared error loss layer:

- Forward: Given an input batch s and a target batch y, saves s and y (for further use) and returns the squared Frobenius norm of $s - y$.
- Backward: Returns $\nabla s\, MSE(s, y) = 2 \times (s - y)/N$, where N is the number of elements in s. N.B.: $\nabla s\, MSE(s, y)$ is computed using the s and y saved by the forward function.
- Param: Returns an empty list since there are no parameters.

B. Bonus Components

This sections describes the implementation of additional modules introduced to give the library more flexibility and modeling power.

1) Layer Modules:

No layer module has been implemented as a bonus component.

2) Activation Modules:

Sigmoid: Simulates a sigmoid activation layer:

- Forward: Given an input batch s, saves s for use in the backward function then returns $sigmoid(s) = 1/(1 + e^{-s})$. N.B.: Sigmoid is computed using the form $1/(1 + e^{-s})$ of the function in order to avoid numerical issues.
- Backward: Using the loss gradient with respect to the output, $grad$, returns $grad \odot \nabla s\, sigmoid(s) = grad \odot sigmoid \odot (1 - sigmoid(s))$. N.B.: $\nabla s\, sigmoid(s)$ is computed using the s saved by the forward function.
- Param: Returns an empty list since there are no parameters.

3) Loss Modules:

MAE: Simulates a mean absolute error loss layer:

- Forward: Given an input batch s and a target batch y, saves s and y (for further use) and returns the l1 norm of $s - y$.
- Backward: Returns $\nabla s\, MAE(s, y) = sign(s - y)/N$, where N is the number of elements in s. N.B.: $\nabla s\, MAE(s, y)$ is computed using the s and y saved by the forward function.
- Param: Returns an empty list since there are no parameters.

III. Test Model

In order to assess the correctness of our framework, we were asked to implement and optimize a $2 \times 25^3 \times 2$ neural network using our library at its core. As for the data, training and testing data sets, each of size 1000, were generated. In it, each data point is sampled uniformly in $[0, 1]^2$ and labeled 1 if inside the circle of radius $1/\sqrt{2\pi}$ and center $(0.5, 0.5)$ and 0 otherwise. The model having two output units, we turned each label into a pair of entries, one hot encoding which label is correct (for label 1, [0,1], for label 0, [1,0]).Since optimizing our model using MSE, we considered encoding our labels using -1 instead of 0 (for label 1, [-1,1], for label 0, [1,-1]) to increase the distance between the two values, potentially reducing interference brought by their proximity, but it didn't seem to improve the performance.

A. Implementation Validity

The test model seems to always converge, reducing it's training loss consistently across SGD iterations and resulting in good final training loss, test loss and test accuracy. We take this as evidence of the correctness of the implemented modules. The evolution of the losses and accuracies on a single run of the training can be observed on respectively Figure 1 and 2.

B. Fine-Tuning

Our test model has hyperparameters, the batch size, SGD's learning rate, the number of epochs and the activation functions. Each has been fine-tuned using grid-search:

- The batch size: We found that using 25 samples par batch gave a good trade-off between the convergence rate and running time per epoch.
- SGD's learning rate: The learning rate appeared to behave differently depending on the activation used. In the end, for the combination of activations we chose, a learning rate of 10^{-2} was the highest that didn't cause divergence.

- The number of epochs: Convergence appeared to be quite slow, significant increases in performance were observed for any number of epochs lower than 4000. Since our training iterations are fast run, we kept this number as our number of epochs.
- The activation functions: Many combinations, most involving the $Tanh$ activation, gave satisfying results. That being said, the combination of 3 $Tanh$ activations seemed to outperform the others.

N.B.: The number and kind of layers, the loss and the optimization algorithm were not considered since immutable.

C. Final Performance

In order to evaluate rigorously the real performance of our model, we decided to run it 110 times, use bootstrap re-sampling to generate 10^6 100-sample estimators and finally use these to construct solid confidence intervals. Despite the obvious handicap of using the MSE loss in the context of a classification, our test model performs remarkably with an average test accuracy in $[98.55, 98.63]$ % and a test accuracy standard deviation in $[0.53, 0.59]$ with probability 99%.

IV. Conclusion

We implemented a simple library, 'Mini-Torch', which can be used to build fully connected neural networks. It provides the Sigmoid, Tanh and ReLU activations, the MSE and MAE loss function and supports SGD optimizations. Based on the empirical results, it is fair to assess that the library performs as desired.

The test model built using this library, for a batch size of 25, 4000 epochs, 3 Tanh activations and a learning rate of 0.01, does a very respectable job at modeling our simple database with an average performance shown to be tightly and consistently around 99% test accuracy.

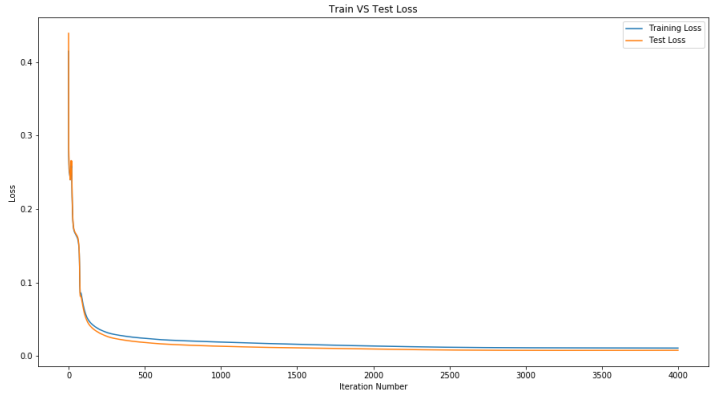

Figure 1. Train/Test Loss VS Iteration Number for batch-size 25, epochs 4000, learning rate 0.01, on a single run of the final model

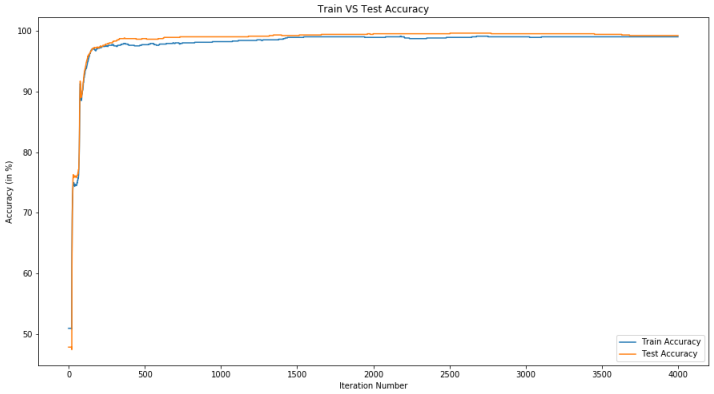

Figure 2. Train/Test Accuracy VS Iteration Number for batch-size 25, epochs 4000, learning rate 0.01, on a single run of the final model