EE-559 Project 2 – Mini deep-learning framework

Barış Sevilmiş Furkan Karakaş Sena Necla Çetin
School of Computer and Communication Sciences, EPFL, Switzerland
{baris.sevilmis, furkan.karakas, sena.cetin}@epfl.ch

***Abstract*—Our main aim is to design a mini deep learning framework using only `PyTorch`'s tensor operations and the standard math library, in particular not using any autograd or neural network modules. The best validation accuracy was obtained from ReLU activation function with Adam optimizer, yielding an accuracy of $95.57 \pm 1.00\%$ after hyperparameter tuning.**

I. Introduction

The aim of this project is to design a mini deep learning framework using only `PyTorch`'s tensor operations and the standard math library, in particular, without using autograd or the neural-network modules. The goal is to determine whether a point $(x, y) \in [0, 1]^2$ lies inside the disc centered at $(0.5, 0.5)$ with radius $\frac{1}{2\pi}$. We used a fully connected layer with two input units, three hidden layers of 25 units, and one output unit. The last activation function is sigmoid in every network, and we tested different networks with activation functions ReLU, Sigmoid and Tanh in the intermediate layers. As optimizers, we tested the networks with SGD and Adam [1], which we also implemented from scratch.

In the following, we start by describing the dataset in Section II. In Section III, we explain the methods, i.e. models, activation functions, loss functions, optimizers, and evalution steps that we used in our architectures. In Section IV, we discuss the results that we obtain from hyperparameter tuning in different models. Finally, we finish the report with concluding remarks and discussions.

II. Data description

We generate a synthetic dataset of 1,000 training and test samples. The sample consists of two numbers (x, y) uniformly sampled in the interval $[0, 1] \times [0, 1]$. If the point is outside of the disc centered at $(0.5, 0.5)$ with radius $\frac{1}{\sqrt{2\pi}}$, then we set the label to 0. Otherwise, we set the label to 1. We note that the area of the disc is $\pi r^2 = \pi \frac{1}{2\pi} = \frac{1}{2}$, which is half of the area 1. Hence, we will have a balanced dataset with approximately 500 samples from each class.

III. Methodology

A. Models

In this subsection, fully connected layer implementation and sequential module to build networks out of activations and layers are defined. Models created by implementations, are provided within the project documentation, to be more specific input layer with 2 neurons, 3 hidden layers with 25 neurons and an output layer of 1 neuron.

1) Linear: This module implements the fully connected/dense layers. Number of input neurons and number of output neurons to create necessary weights and biases for a specific dense layer are kept within. Following functions are the main building blocks:

- zero_grad(): Reset all weight and bias parameters to 0.
- forward(): $z_i = x_{i-1} \cdot W + b$ is computed.
- backward(): $Grad(x_{i-1}) = Grad(z_i) \cdot W^T$

To accumulate the gradients inside of the linear layers, we calculate the quantity `self.grad_weights += self.data.t() @ self.grad_data`. Grad data is what is coming from the next layer. In order to calculate this quantity, we need to store the tensor in the forward pass in the memory.

2) Sequential: This module uses linear layers and activation functions to build the sequential network. The methods such as zero grad, reset, forward and backward are defined iteratively from the same methods of linear layers and activation functions.

B. Activations

This module is used to define the activation function, i.e. the output of a node usually given after each layer of the network. In forward-propagation, $g(x)$, the activation function receives the output of the respective layer as input. In back-propagation, $g'(x)$, the derivative of the activation function receives the input to the activation function as in the forward pass as its input, and the gradient with respect to the output is multiplied with this quantity as per the *chain rule*. In order to backward the gradients to the previous layer, we need to store the input tensor from the forward pass in the memory as we will be using this data to compute the gradient. Our framework uses the following activation functions:

1) ReLU:

$$g(x) = \max(0, x)$$

$$g'(x) = \begin{cases} 1, & if x > 0 \\ 0, & otherwise \end{cases}$$

2) Sigmoid:

$$g(x) = \frac{1}{1 + e^{-x}}$$

$$g'(x) = g(x) \cdot (1 - g(x))$$

3) Tanh:

$$g(x) = \tanh(x)$$

$$g'(x) = 1 - g^2(x)$$

Note that in the backward pass, the computed derivatives are multiplied by the gradient with respect to the output by using the *chain rule*.

C. MSE loss

As the underlying loss function, we use *mean-squared error* (MSE). Here, $0 \leq y_i \leq 1$ is the predicted label at the output neuron, and $\hat{y}_i \in \{0, 1\}$ is the true label of the sample point, and N is the batch size in the training phase. The loss is calculated as the mean score of the samples in the batch.

$$L = \frac{1}{N}\sum_{i=1}^{N}(y_i - \hat{y}_i)^2$$

$$\frac{\partial L}{\partial y_i} = \frac{2}{N}(y_i - \hat{y}_i)$$

D. Optimizers

Optimizers are the main tool required to begin the training and enable model learning. Different optimizers could be created easily as sub classes given the main optimizer class is implemented. Base class for optimizer (**_Optimizer**) implements train() method only, in which optimization is completed. It could be depicted as in Algorithm 1:

Algorithm 1 _Optimizer_

procedure TRAIN(TRAINSET, VALIDSET)
  for $e \in epochs$ **do**
    for $b \in trainset.batches$ **do**
      $model.zero\_grad()$
      $preds \leftarrow model(b.input)$
      $loss \leftarrow criterion(preds, b.target)$
      $grad \leftarrow criterion.backward()$
      $model.backward(grad)$
      optimizer.step()
    $train\_acc \leftarrow compute\_nb\_errors(model, trainset)$
    $val\_acc \leftarrow compute\_nb\_errors(model, validset)$

1) SGD: SGD is one of the most simple optimizers, in which batch size can be varied such that implementation leads to Mini-batch-SGD. Operations in SGD's step() function are demonstrated in Algorithm 2, where weights and biases are decreased by the multiplication of learning rate and the corresponding gradients.

Algorithm 2 SGD(_Optimizer_)

procedure STEP()
  for $w, b, gradW, gradB \in model.params()$ **do**
    $w \leftarrow w - lr * gradW$
    $b \leftarrow b - lr * gradB$

2) Adam: Adam is an advanced optimization technique, which can be considered as a mixture of Gradient Descent, Momentum, and RMSProp optimization techniques. We provide three extra parameters for simplicity, namely $beta1$, $beta2$, and $epsilon$. $Beta1$ is utilized as the exponential decay rate for first moment estimates (mw, mb) and $beta2$ is used as the exponential decay rate of second moment estimates (vw, vb). These moments are updated as in Algorithm 3, and moment corrections are computed, namely $t1, t2, t3, t4$. These moment corrections are used in the ending weight and bias updates as in Algorithm 3, which provides a faster and more efficient training procedure.

Algorithm 3 Adam(_Optimizer_)

procedure STEP()
  for $w, b, gradW, gradB \in model.params()$ **do**
    $mw \leftarrow beta1 * mw + (1 - beta1) * gradW$
    $t1 \leftarrow mw/(1 - beta1^{step})$
    $mb \leftarrow beta1 * mb + (1 - beta1) * gradB$
    $t2 \leftarrow mb/(1 - beta1^{step})$
    $vw \leftarrow beta2 * vw + (1 - beta2) * gradW^2$
    $t3 \leftarrow vw/(1 - beta2^{step})$
    $vb \leftarrow beta2 * vb + (1 - beta2) * gradB^2$
    $t4 \leftarrow vb/(1 - beta2^{step})$
    $w \leftarrow w - lr * \frac{t1}{\sqrt{t3}+\epsilon}$
    $b \leftarrow b - lr * \frac{t2}{\sqrt{t4}+\epsilon}$
  $step \leftarrow step + 1$

IV. RESULTS

We ran different architectures in 10 rounds and we calculated the mean and standard deviations of the recorded accuracy in the rounds. We had hypothesized that Adam optimizer would perform the best with ReLU activation function. Our expectations were realized since we got the best performance with Adam and ReLU. The validation accuracy scores for different setups are illustrated in Table I.

TABLE I
THE BEST FINE-TUNED HYPERPARAMETERS FOR DIFFERENT ACTIVATION FUNCTIONS AND OPTIMIZERS

	Learning rate	Beta1	Beta2	Accuracy
ReLU (SGD)	0.5	N/A	N/A	$93.08 \pm 2.93\%$
Sigmoid (SGD)	0.1	N/A	N/A	$49.62 \pm 1.74\%$
Tanh (SGD)	0.9	N/A	N/A	$91.32 \pm 3.41\%$
ReLU (Adam)	1e-3	0.9	0.999	$95.57 \pm 1.00\%$
Sigmoid (Adam)	1e-3	0.9	0.999	$49.21 \pm 1.15\%$
Tanh (Adam)	0.1	0.5	0.999	$49.41 \pm 1.19\%$

We note that some results are sub-optimal. For instance, sigmoid activation function performed poorly with SGD as well as Adam optimizers, obtaining an accuracy level less than 50%. On the other hand, Tanh activation function performed well with the SGD optimizer, but performed poorly with the Adam optimizer.

We show the changes in train loss, validation loss, train accuracy and validation accuracy with respect to number of

epochs in Figures 1 and 2. We picked the best performing activation functions for each optimizer, i.e. the ReLU activation function.

In Figure 3, we illustrate two misclassified points. In the first one, our model predicts that the point lies inside of the circle. In the second one, our model predicts that the point lies outside of the circle. However, calculating the distances of the points to the center of the disc, we realize that they are not correctly classified. We see that these errors are close to the boundary region, and most of the misclassified samples occur around that region.

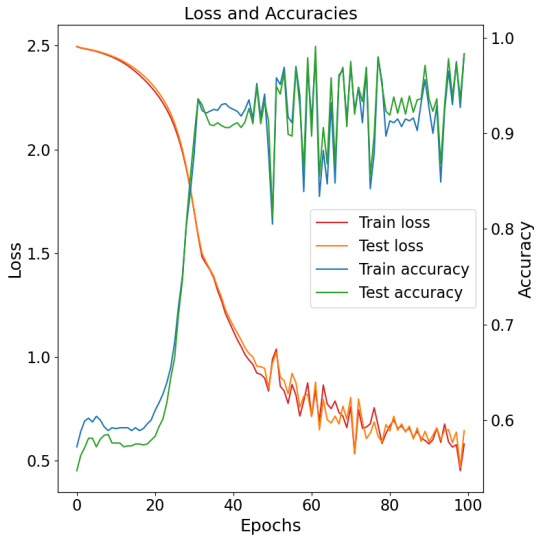

Fig. 1. Losses and accuracy values of the network with ReLU activation function and SGD optimizer

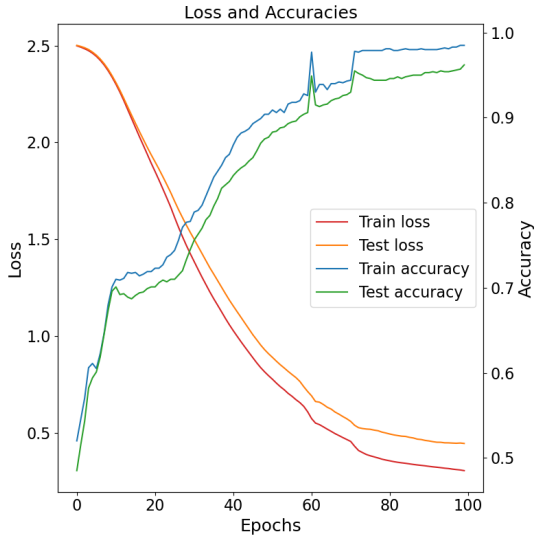

Fig. 2. Losses and accuracy values of the network with ReLU activation function and Adam optimizer

```
Misclassification #2:
Input point: (x,y)=(0.22690123319625854,0.20222771167755127)
Prediction: 1
Actual: 0
Distance to the (0.5,0.5): 0.16325128078460693
Radius 1/(2*pi): 0.15915494309189535
Radius is greater than distance (Inside): False

Misclassification #3:
Input point: (x,y)=(0.840284526348114,0.3032761216163635)
Prediction: 0
Actual: 1
Distance to the (0.5,0.5): 0.15449383854866028
Radius 1/(2*pi): 0.15915494309189535
Radius is greater than distance (Inside): True
```

Fig. 3. Two misclassified points

V. Discussion & Conclusion

Since our task is fairly simple, we expected our classifier to work well with such a small network size of 3 hidden layers with 25 neurons for each. Indeed, we easily reach an accuracy level around 95% for specific activation functions and optimizers. However, some combinations, in particular sigmoid with either optimizer and Tanh with Adam, have performances worse than random guessing. We are not sure why this happens. We are positive that we implemented the forward and backward passes of the activation functions correctly because there is always a sigmoid layer before the final output neuron in every network and we have a well-performing Tanh network with the SGD optimizer.

We realize in Figure 1 that there are a lot of oscillations after 30 epochs with the SGD optimizer. However, we do not observe those oscillations with the Adam optimizer in Figure 2. This is the reason why Adam optimizer usually performs better than the SGD optimizer – with the utilization of momentums, the Adam optimizer learns more steadily and smoothly. We realize that the accuracy scores increase rather more slowly in Adam than in SGD, but Adam is better in reaching the local minimum as the number of epochs increase by adaptive momentum calculations and "not being trapped" in a valley during learning.

References

[1] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," 2014, cite arxiv:1412.6980Comment: Published as a conference paper at the 3rd International Conference for Learning Representations, San Diego, 2015. [Online]. Available: http://arxiv.org/abs/1412.6980