Deep Learning Project 2 – Mini deep-learning framework

Orest Gkini, Theofilos Belmpas
EPFL, Switzerland

***Abstract*—In this report, we present our mini deep-learning framework, which we built using the basic tensor operations of PyTorch and the standard math library. Similar to the approach of PyTorch, we have built several modules that allow users to build neural networks along with our own *automatic differentiation engine* to optimize its parameters during the training process.**

I. Introduction

Neural networks (NNs) consist of a series of layers that are characterised by their parameters: the weights W and the biases b. Training a neural network consists of two phases:

- the *forward propagation*, during which each layer performs some computations on the data and gives its output as input to the next layer. More specifically each layer performs a linear transformation on its input $Wx+b$ followed by a non-linear activation function σ.
- the *backward propagation*, during which the parameters of the network are adjusted accordingly in order to improve its future predictions. The derivatives of the error with respect to the parameters of the network are computed, and then, then parameters are updated using gradient descent.

Modern deep learning libraries are equipped with *automatic differentiation* engines that perform the backpropagation phase. In our framework, we take a similar approach to PyTorch's `autograd` package [1], meaning that during the forward pass we build a **Directed Acyclic Graph (DAG)** of tensor operations, so that in the backward pass we can compute the gradient of the loss automatically with respect to any tensor. This approach offer the benefits of simpler syntax, since the user only needs to specify the operations of the forward pass, and flexibility as the graph is dynamically created.

II. Implementation Details

In this section, we describe the main components and features of our framework.

A. Directed Acyclic Graph (DAG)

Our framework builds a DAG of tensor operations **dynamically during the forward pass** of the network. Our `Tensor` class is a wrapper that contains a `torch.tensor` that stores its data and provides several useful functions for initialization and computations. A `Tensor` object constitutes a node in the computation graph, and thus, it keeps a list of the `Tensors` that were used to create it (`backward_nodes`) and a pointer to the `backward()` method of the operation that produced it (`grad_fn`) in order to compute its gradient during the backward pass. A `Tensor` also has a boolean `with_grad` field that indicates whether its gradient should be computed or not. Similar to PyTorch, if at least one of the inputs that were used to produce a `Tensor` have `with_grad==True` then the result also has that field set to `True`.

B. Core Functions

The core functionality of our framework is implemented in a set of core functions located inside the `autog.functional` package. These functions operate on objects of our `Tensor` class and work for inputs of different shapes, namely scalars, vectors, and matrices.

We have defined functions for the element-wise addition of different input shapes, by broadcasting the values when needed (e.g. when adding a vector to a matrix). We have also implemented functions for the multiplication of a vector with a scalar, element-wise multiplications between vectors, and also multiplications of a matrix with a vector, or between two matrices. Other functions include raising the components of a tensor to a power, the rectified linear unit (ReLU) and the hyperbolic tangent (Tanh).

Each of the above functions, when called during the forward pass perform two tasks. First, they compute the resulting `Tensor` given the input(s). Second, they fill the list of backward nodes to point to the `Tensors` that produced it (only if `with_grad==True`), essentially adding the resulting node to the computation graph for the backward pass, and also, set the `grad_fn` field to point to the `backward()` method of the operation that produced it which specifies how to compute its gradient.

C. Modules

In this section, we present the modules we have implemented, which can be used to build neural networks. Under the hood, they all use the core set of functions presented in the previous section. All subsequent modules inherit from the superclass `Module` and must implement the following methods:

- `forward()`: defines the forward pass of the module.
- `param()`: returns a list containing the parameters of the module.

Linear. This module represent a fully connected linear layer. It has two parameters that define the number of

input and output features (`in_features` and `out_features`, respectively), and the boolean parameter `bias` that indicates whether to add a bias vector or not.

- `forward()`: computes and returns $XW + b$, with $X \in \mathbb{R}^{n \times in}$, $W \in \mathbb{R}^{in \times out}$, $b \in \mathbb{R}^{out}$, where n is the input batch size, in is the number of input features, and out is the number of output features. The result belongs in $\mathbb{R}^{n \times out}$.
- `param()`: returns a list containing the weights and biases.

There some important points worth noting here.

First, our framework works for input *batches* of size n. Each single input vector has in dimensions, so if we stack them vertically we get a matrix $X \in \mathbb{R}^{n \times in}$ (each row is a data point). To compute XW we call our implementation of `matmul()` that performs the multiplication of two matrices.

Second, our module initializes the weights using the Xavier initialization rule [2] as follows:

$$W \sim U\left[-\frac{\sqrt{6}}{\sqrt{n_{in}+n_{out}}}, \frac{\sqrt{6}}{\sqrt{n_{in}+n_{out}}}\right]$$

where n_{in} and n_{out} are the input and output sizes of the module.

Sequential. This module enables the combination of multiple modules sequentially, in order to build deep neural networks.

- `forward()`: calls the `forward()` method of the modules iteratively, in the order they were passed to it, and passes the output of each one as input to the next.
- `param()`: returns a list containing the parameters of all modules.

ReLU. This module defines the rectified linear unit activation function.

- `forward()`: applies the $relu(x) = max(0, x)$ function element-wise.
- `param()`: returns an empty list.

Tanh. This module defines the hyperbolic tangent activation function.

- `forward()`: applies the $tanh(x) = \frac{e^{2x}-1}{e^{2x}+1}$ function element-wise.
- `param()`: returns an empty list.

Sigmoid. This module defines the sigmoid activation function.

- `forward()`: applies the $sigmoid(x) = \frac{1}{1+e^{-x}}$ function element-wise.
- `param()`: returns an empty list.

LossMSE. This module implements the mean squared error loss function, given two `Tensors` as input.

- `forward()`: the MSE loss for one vector x and target y is defined as $mse(x, y) = \frac{1}{d}\sum_{i=1}^{d}(x_i - y_i)^2$, where d is the size of the vectors x and y and the output is a scalar $\in \mathbb{R}$. In our case, however, we have a matrix X of n data points, so the output is a vector of losses of size n and we need to average over those, too, to get a scalar loss.
- `param()`: returns an empty list.

D. Optimizers

For the optimization of the network parameters during the training process we have defined the `Optimizer` superclass that all optimizers inherit from. The constructor accepts the `parameters` that will be updated and the learning rate `lr`, which is by default set to 0.01. The following two methods need to be implemented:

- `step()`: updates all the parameters in the negative direction of their gradients.
- `zero_grad()`: sets the gradients of all parameters to 0.

SGD. We have implemented the stochastic gradient descent optimizer whose `step()` method updates the parameters according the following formula:

$$p_t = p_{t-1} - lr * \nabla p$$

E. Automatic Differentiation

Calling our `backward()` method on a scalar loss value triggers the backpropagation pass of the training phase. The algorithm traverses the computation graph backwards, computing the gradient of the loss with respect to each `Tensor` (recall that a `Tensor` is a node in the graph) and propagating them using the chain rule.

There are several cases that we need to handle based on the shapes of the input `Tensors` of each operation and with respect to which one we are computing the gradient. A lot of useful identities are provided in the following article [3].

The most challenging part was the computation of the gradient of the multiplication between two matrices with respect to one of them, which is a 4-th order tensor. This is needed since our framework allows the input X of a `Linear` module to be a matrix in $\mathbb{R}^{n \times in}$, where n is the batch size and in is the dimension of each data vector. Therefore, during the backward pass we need to compute the gradient of XW with respect to the matrix W. However, similar to the approach followed in Section 3 paragraph (5) of [3], where they compute the gradient of a matrix-vector multiplication with respect to the matrix, the computations can be simplified given the fact that we have a scalar loss.

III. Experiments

To evaluate our framework we generated a dataset of 1,000 points sampled uniformly in $[0, 1]^2$, each with a label 0 if it is located outside the disk centered at $(0.5, 0.5)$ of radius $1/\sqrt{2\pi}$ and 1 if it is inside. We constructed a network with two input units, one output unit, and three hidden layers of 25 units. Listing 1 shows the code needed to build such a model using our framework and Listing 2 shows the training phase.

Table I: Effect of learning rate and batch size on the accuracy across 10 iterations with 100 epochs.

Learning rate	Batch size	Accuracy	Standard deviation
0.001	50	0.73	± 0.14
	100	0.66	± 0.14
	200	0.65	± 0.10
	1000	0.57	± 0.13
0.005	50	0.92	± 0.04
	100	0.86	± 0.08
	200	0.80	± 0.13
	1000	0.56	± 0.09
0.01	50	**0.96**	± 0.01
	100	0.91	± 0.05
	200	0.82	± 0.13
	1000	0.72	± 0.09
0.1	50	**0.96**	± 0.02
	100	0.95	± 0.02
	200	0.91	± 0.04
	1000	0.79	± 0.12

Listing 1: Defining the network.

```
model = nn.Sequential(
  nn.Linear(2, 25),
  nn.ReLU(),
  nn.Linear(25, 25),
  nn.ReLU(),
  nn.Linear(25, 25),
  nn.ReLU(),
  nn.Linear(25, 1),
  nn.Sigmoid()
)
```

Listing 2: Training the network.

```
EPOCHS = 100
mse = nn.LossMSE()
optimizer = optim.SGDStepper(model.params(), lr=0.01)
for epoch in range(EPOCHS):
  # Shuffle data in every epoch
  train_input, train_target = shuffle(train_input, train_target)

  # Break to batches
  for b in range(0, train_input.shape[0], mini_batch_size):
      # Calculate output and loss
      out = model(train_input.narrow(0, b, mini_batch_size))
      loss = mse(out, train_target.narrow(0, b, mini_batch_size))

      # Training
      optimizer.zero_grad() # Clear gradients before backprop
      loss.backward()       # Backprop
      optimizer.step()      # Update parameters
```

Table I shows the effect of the learning rate and the batch size in the mean accuracy of our model across 10 iterations and 100 epochs. We see that for a learning rate of 0.01 and 0.1 and a batch size equal to 50 we achieve the best mean accuracy with a really low standard deviation. In general, we see that larger learning rates yield a higher mean accuracy. Also, another pattern is that smaller batch sizes are better, with a batch size equal to the size of the full train set (1000) being the worst.

IV. Conclusion

In this project, we built a mini deep-learning framework from scratch using only the basic tensor operation of PyTorch and the standard math library. Our framework allows users to design, train, and test neural networks with several built-in modules. It is equipped with an automatic differentiation engine, which computes the gradients of the loss with respect to any tensor during the backward pass of the network training process and propagates them across the computation graph.

References

[1] A. Paszke, S. Gross, S. Chintala, G. Chanan, E. Yang, Z. DeVito, Z. Lin, A. Desmaison, L. Antiga, and A. Lerer, "Automatic differentiation in pytorch," in *NIPS 2017 Workshop on Autodiff*, 2017.

[2] X. Glorot and Y. Bengio, "Understanding the difficulty of training deep feedforward neural networks," in *Proceedings of the Thirteenth International Conference on Artificial Intelligence and Statistics*, ser. Proceedings of Machine Learning Research, vol. 9. Chia Laguna Resort, Sardinia, Italy: PMLR, 13–15 May 2010, pp. 249–256.

[3] K. Clark, "Computing neural network gradients." [Online]. Available: https://web.stanford.edu/class/archive/cs/cs224n/cs224n.1184/readings/gradient-notes.pdf