Mini Deep Learning Framework

Deep Learning – Project 2

Marie Biolková **Minh Tri Pham**
Department of Computer Science, EPFL, Switzerland
{marie.biolkova, minh.pham}@epfl.ch

***Abstract*—In this report we present a deep learning framework with a few basic functionalities built using only tensor operations available in Pytorch and the standard Python mathematics library. The framework allows to build a neural network with fully connected layers and provides multiple activation functions (ReLU, Leaky ReLU and Tanh). All components can be combined in a sequential manner and subsequently optimized using SGD or Adam and the MSE or Cross Entropy loss. The framework therefore provides all core features to perform basic classification tasks. The report evaluates the performance of the framework on a simple simulated dataset.**

I. Introduction

The recent surge of popularity of deep learning has led to the development of a plethora of frameworks that enable users to easily construct and train neural networks (NNs) with complex and deep architectures. Their users are given a rich deep learning toolkit, without being required to go into the mechanisms that power them, hence making deep learning accessible for anyone. Despite that, one should still understand how backpropagation works – for instance, this can help identify any potential issues encountered in training. The aim of this project is therefore to implement a small-scale framework similar to PyTorch [1] or TensorFlow [2].

II. Methods

Suppose that we have a set of labelled data (x_i, y_i), $i = 1, \ldots, n$, and weights w with biases b that correspond to the parameters of a multi-layer perceptron (MLP). The training of an MLP consists of learning a function f by finding w, b such that the loss function

$$\mathcal{L}(w,b) = \sum_{i}^{n} \ell\left(f\left(x_i; w, b\right), y_i\right) \tag{1}$$

is minimized. In the context of NNs, this is done using the backpropagation algorithm, which has two key components: the forward and backward pass. The forward pass requires that the activations at each layer l be computed. If x^0 are the input data and x^{l-1}, $(w^{(l)}, b^{(l)})$ the input and the parameters of the l^{th}, respectively, we find

$$\begin{aligned} s^{(l)} &= w^{(l)} x^{(l-1)} + b^{(l)} \\ x^{(l)} &= \sigma\left(s^{(l)}\right) \end{aligned} \tag{2}$$

where σ is a non-linear activation function.

In the backward pass, derivatives and the chain rule are used to quantify how changing a certain parameter affects the loss. In particular, we compute the derivative of the loss with respect to the parameters

$$\frac{\partial \ell}{\partial w^{(l)}} = \frac{\partial \ell}{\partial s^{(l)}} \left(x^{(l-1)}\right)^{\top}, \qquad \frac{\partial \ell}{\partial b^{(l)}} = \frac{\partial \ell}{\partial s^{(l)}} \tag{3}$$

which also requires the gradients with respect to the activations

$$\begin{aligned} \tfrac{\partial \ell}{\partial s^{(l)}} &= \tfrac{\partial \ell}{\partial x^{(l)}} \odot \sigma'\left(s^{(l)}\right) \\ \tfrac{\partial \ell}{\partial x^{(l)}} &= \left(w^{(l+1)}\right)^{\top} \tfrac{\partial \ell}{\partial s^{(l+1)}}. \end{aligned} \tag{4}$$

The gradient in the last layer $\frac{\partial \ell}{\partial x^{(L)}}$ is determined by the loss function ℓ. Finally, to minimize the loss, we update each parameter by taking a step of size η in the negative direction of its gradient.

III. Modules

In our implementation, each component of the MLP is defined as a module that inherits from a common class `Module` that requires `forward` and `backward` to be specified for the two passes of backpropagation. The former computes the forward pass given the input, as in (2) while the latter computes the gradient w.r.t. the module's input, given the gradient w.r.t. its output, as given by (4). For parametrized modules (fully connected layers), the `param` and `zero_grad` methods return the parameters and set their gradients to zero, respectively.

A. Linear

The `Linear` module implements a fully connected layer. Given the output from the previous layer, it provides the computation of $s^{(l)}$ in (2) in the forward pass and $\frac{\partial \ell}{\partial w^{(l)}}, \frac{\partial \ell}{\partial b^{(l)}}$ in `backward` (which also accumulates the gradients).

A careful parameter initialization is needed to avoid the problem of vanishing gradients, especially in the presence of activations that saturate (such as $\tanh(x)$ as x tends to $\pm\infty$). The `param_init` method implements the Xavier initialization [3] which is designed to control the variances of the parameters across layers. The parameters of layer l are therefore initialized depending on the dimension of the input, $d_{\text{in}}^{(l)}$, and the output, $d_{\text{out}}^{(l)}$.

$$w^{(l)}, b^{(l)} \sim \mathcal{N}\left(0, g(\sigma^{(l)}) \times \sqrt{\frac{2}{d_{\text{in}}^{(l)} + d_{\text{out}}^{(l)}}}\right) \tag{5}$$

In the above, we also multiply the standard deviation by a corrective gain g to account for applying different activations, as given in He et al. (2015). For instance $g(\text{ReLU}) = \sqrt{2}$. If no activation is applied, we sample from the standard normal distribution.

B. Activations

Activation functions allow us to introduce non-linearity into the network and therefore elevate the representation power of the otherwise affine transformations of the input. Our framework offers three different activations. First, probably the most popular, is the *Rectified Linear Unit (ReLU)* which encounters fewer vanishing gradient problems than some other activations that saturate (e.g. $\tanh$). It also provides sparse activation.

$$\mathrm{ReLU}(x) = \max(0, x)$$

Closely related is the *Leaky ReLU* which is parametrized by α. It is equivalent to ReLU if $\alpha = 0$. The gradient is equal to α for negative values, as opposed to zero in the case of ReLU. In our experiments, we set $\alpha = 0.05$.

$$\mathrm{LeakyReLU}(x) = \left(\begin{cases} \alpha x & \text{if} \quad x < 0, \\ x & \text{if} \quad x \geq 0 \end{cases} \right)$$

The third activation that was implemented is the *hyperbolic tangent*:

$$\tanh(x) = \frac{e^{2x} - 1}{e^{2x} + 1}$$

Although it can suffer from the vanishing gradient problem, it is differentiable everywhere, unlike the other activations we consider here.

C. Sequential

This module allows to stack multiple modules and in particular combine multiple fully connected layers with activation layers. The module keeps track of these layers and applies the forward pass in a sequential manner. For the backward pass, it iterates over the layers in reverse order, back propagating the gradients. The `param` function return a list of all parameters and `zero_grad` sets gradients to zero across all layers.

D. Loss Functions

The framework provides the following two loss functions: *Mean Squared Error (MSE)*

$$\mathcal{L}(w, b) = \frac{1}{n} \sum_{i=1}^{n} \left(f\left(x_i | w, b\right) - y_i \right)^2 \tag{6}$$

and *Cross-Entropy*

$$\mathcal{L}(w, b) = -\sum_{i=1}^{n} p(x_i) \log q(x_i), \quad q(x_i) = \frac{e^{f(x_i|w,b)}}{\sum_j e^{f(x_j|w,b)}} \tag{7}$$

While MSE is frequently used for its desirable properties (differentiability, convexity, symmetry), it is not very suitable for classification. One of the reasons is that it strongly penalizes confident predictions. Cross-entropy is a better choice because it increases as the predicted probability $q(x)$ diverges from the target distribution $p(x)$. To implement these losses, the labels (y_i and $p(x)$) were converted to one-hot encoding.

E. Optimizers

We chose to explore two different optimization rules. Firstly, the *Stochastic Gradient Descent (SGD)*, which performs a parameter updates after computing the gradient based on a single randomly sampled training example (x_i, y_i):

$$w_{t+1}^{(l)} = w_t^{(l)} - \eta \nabla \ell \left(f\left(x_i; w_t, b_t\right), y_i \right).$$

Secondly, we implemented *Adam* [5]:

$$\begin{aligned} m_t &= \beta_1 m_{t-1} + (1 - \beta_1) G_t \\ v_t &= \beta_2 v_{t-1} + (1 - \beta_2) G_t^2 \\ \hat{m}_t &= \tfrac{m_t}{1-\beta_1^t}, \quad \hat{v}_t = \tfrac{v_t}{1-\beta_2^t} \\ w_{t+1} &= w_t - \tfrac{\eta}{\sqrt{\hat{v}_t} + \varepsilon} \hat{m}_t \end{aligned} \tag{8}$$

where G_t is the stochastic gradient at time t, $\ell\left(f\left(x_i; w_t, b_t\right), y_i\right)$. Unlike SGD, Adam adapts the learning rate for each parameter separately based on the first two moments of the gradients. We set the decay parameter for the first moment (β_1) to 0.9 and for the second moment (β_2) to 0.999, and $\varepsilon = 10^{-8}$ (prevents division by zero), which are standard values.

IV. Results

The framework was tested on a toy problem – we generated a training and a test set of 1000 points sampled uniformly in $[0, 1]^2$. The label was 1 if the point was within the disk centered at $(0.5, 0.5)$ with radius $1/\sqrt{2\pi}$ and 0 otherwise.

Prior to training, we standardized the datasets to have zero mean and standard deviation of 1.

The architecture we explored had two input units, two output units and 3 hidden layers of 25 units each. An activation function was applied after each hidden layer. The network was trained for 30 epochs with a batch size of 25 samples. We tuned the learning rate using a grid search and evaluated the performance of each model across 10 randomized runs. The results are shown in Table I.

TABLE I
Mean test accuracy and standard deviation of each model considered, computed across 10 runs.

Loss	Criterion	Activation	Accuracy (%)	SD (%)
MSE	SGD	ReLU	96.7	1.1
		Tanh	95.3	1.4
		Leaky ReLU	89.7	1.2
	Adam	ReLU	96.8	1.6
		Tanh	94.9	1.9
		Leaky ReLU	85.9	2.8
Cross-Entropy	**SGD**	**ReLU**	**97.4**	**0.9**
		Tanh	97.0	1.0
		Leaky ReLU	92.7	0.8
	Adam	ReLU	96.9	0.6
		Tanh	96.7	1.4
		Leaky ReLU	94.7	1.6

Overall, Cross-Entropy outperformed MSE, proving that it is the better choice for loss when it comes to classification problems. ReLU gave best results out of all activation functions while Leaky ReLU was the worst. It would however be worth exploring whether tuning α could significantly improve the accuracy. Adam was not as efficient as SGD – it only worked better with the combination Leaky ReLU + Cross-Entropy loss. In general, the accuracies had higher variance between runs with Adam than with SGD. The best result was achieved with ReLu and Cross-Entropy loss, trained with SGD using learning rate $\eta = 0.001$. This yielded average test accuracy of 97.4 ± 0.9 %.[1]

V. Conclusion

In this project, we built a small deep-learning framework from scratch that offers basic functionalities for training neural networks, including a fully connected layer, a sequential module, various activation functions, two different loss functions and two optimizers. It was shown to perform well when deployed on a simple binary classification problem.

[1]The test accuracy given by `test.py` when running only the best model overall is 98.1%, and when running the model required by the task (SGD and MSE), it is 97.4%. These are different than the results in Table I since it is not an average, but a value obtained by training and testing the model only once.

References

[1] Paszke, A., Gross, S., Massa, F., Lerer, A., Bradbury, J., Chanan, G.,..., Chintala, S. (2019). PyTorch: An Imperative Style, High-Performance Deep Learning Library. *In Advances in Neural Information Processing Systems 32*, pp. 8024–8035. Retrieved from http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf

[2] Abadi, M., Barham, P., Chen, J., Chen, Z., Davis, A., Dean, J., Devin, M., Ghemawat, S., Irving, G., Isard, M. & others (2016). TensorFlow: A System for Large-Scale Machine Learning *OSDI* , pp. 265-283.

[3] Glorot, X, Bengio, Y. (2010). Understanding the difficulty of training deep feedforward neural networks. *In International Conference on Artificial Intelligence and Statistics (AISTATS).*

[4] He, K., Zhang, X., Ren, S. and Sun, J. (2015). *Delving deep into rectifers: Surpassing human-level performance on imagenet classification.* CoRR, abs/1502.01852.

[5] Kingma, P. D., Ba, J. (2014). Adam: A Method for Stochastic Optimization, *CoRR*, abs/1412.6980.