Mini-project II

Ada Pozo Pérez, Lucía Montero Sanchis, Milica Novakovic
Deep Learning 2018, EPFL Lausanne, Switzerland

***Abstract*—This report describes the implementation of a simple deep learning framework. Several different optimizers, activation functions, layers and loss functions are implemented, including fully connected, dropout, *ReLU*, *Tanh*, sequential and *MSE* loss among others. This framework is used for testing different architectures on a toy dataset, using different optimizers, in order to determine the effects that different activation functions and optimizers have in the training and accuracy.**

I. Introduction

In this project we implement a deep learning framework using only *Pytorch* tensors and *numpy*.

The framework is built with two base classes: *Module* and *Optimizer*. The following containers are implemented as subclasses for them:

- Layers: *Linear*, *Sequential* and *Dropout*.
- Activation functions: *ReLU*, *Leaky ReLU*, *Sigmoid* and *Tanh*.
- Loss functions: *Mean Squared Error (MSE)* and *Cross-entropy loss*.
- Optimizers: *Stochastic Gradient Descent* and *Adam*.

To test our framework we generate a toy dataset and run several experiments on it, comparing the performance of different combinations of modules, activations, optimizers and loss functions.

The rest of this report is organized as follows: Section II describes how the different containers were implemented while Section III shows the results of the experiment to test the framework. Conclusions are drawn in Section IV.

II. Implementation

We have defined two generic classes, *Module* and *Optimizer*, that are used as parent classes for the implementations. The optimizers are based on the structure of *Optimizer*, whereas the remaining implementations are based on *Module*.

Optimizer has two methods:

- *step*: implements the update of the parameters.
- *adaptive_lr*: returns a generator that decays the learning rate by a factor with every optimization step.

In *Module* we can find the methods:

- *forward* and *backward*: implement the forward and backward passes of backpropagation, respectively.
- *update*: updates the values of the parameters and resets the computed gradients to zero.
- *params*: returns the values and the last computed gradients of the parameters of the layer.

A. Layers

1) Linear: The fully connected layer is implemented in the *Linear* module. It is created specifying the number of input and output units. It is also possible to specify the default learning rate, which will be used to update the parameters. Nevertheless, this learning rate is only used when no learning rate is specified for the *update* function.

Unless a specific standard deviation (std) is specified during initialization, the weights are initialized with the Xavier [1] initialization adapted for ReLU [2] – following a zero-mean Gaussian distribution with standard deviation given by $\text{std} = \sqrt{2/n}$, where n is the number of input units. If a value for parameter *std_w* is specified, then the weights are initialized according to the zero-mean Gaussian distribution whose std value is *std_w*. For instance, when using $\tanh$ or sigmoid activation functions we recommend to specify a value for the parameter equal to $\text{std} = 1\sqrt{n}$.

In addition to this, in the initialization it is also specified whether there is a bias – in which case the bias values would be initialized following a Gaussian distribution with zero mean and standard deviation given by the parameter *std_b*. If the value of parameter *std_b* is not specified, they are initialized to 0. If there is no bias, in the backpropagation this parameter is ignored.

During the forward pass, the input to the layer is saved for the subsequent gradient computation. The output of the layer $s^{(l)}$ is computed with the equation 1, where $w^{(l)}$ and $b^{(l)}$ are the values of the weights and the bias respectively and $x^{(l-1)}$ is the input to the layer. The bias term is considered if it was specified to do so, otherwise it is ignored.

$$s^{(l)} = w^{(l)} x^{(l-1)} + b^{(l)} \tag{1}$$

The backward pass function receives as an argument the gradient of the loss with respect to the output of the layer ($\partial l / \partial s^{(l)}$) and returns the gradient of the loss with respect to the input of the layer, which is computed as given by equation 2. The gradient of the loss with respect to the weights and with respect to the bias are computed with equations 3 and 4, respectively:

$$\frac{\partial l}{\partial x^{(l-1)}} = \frac{\partial l}{\partial s^{(l)}} w^{(l)} \tag{2}$$

$$\frac{\partial l}{\partial w^{(l)}} = \left(\frac{\partial l}{\partial s^{(l)}} \right)^T x^{(l-1)} \tag{3}$$

$$\frac{\partial l}{\partial b^{(l)}} = \frac{\partial l}{\partial s^{(l)}} \tag{4}$$

The update of the parameters is performed when the method *update* is called. If a learning rate is passed as a parameter

it is used for the update, otherwise the one specified upon initialization is used. As will be explained in more detail in the optimizers section II-D, the update can be done directly on the derivatives of the bias and weights (for instance, when using SGD), or in the case when the parameter *values* is specified it will use these values instead (if Adam is used, the optimizer computes the parameter *values* that should be used). The equations used for the updates are therefore given by 5 and 6, where γ is the learning rate:

$$w^{(l)} \leftarrow \begin{cases} w^{(l)} - \gamma \cdot \partial l / \partial w^{(l)}, & \text{if SGD is used} \\ w^{(l)} - \gamma \cdot \text{value}_w, & \text{if Adam is used} \end{cases} \tag{5}$$

$$b^{(l)} \leftarrow \begin{cases} b^{(l)} - \gamma \cdot \partial l / \partial w^{(l)}, & \text{if SGD is used} \\ b^{(l)} - \gamma \cdot \text{value}_b, & \text{if Adam is used} \end{cases} \tag{6}$$

When using *Adam*, value_w and value_b are computed as given by equation 16.

Lastly, the *Linear* module's method *params* returns a list of tuples containing a tuple with the weights and the gradient and, if bias is activated, a second tuple with the values of the bias and the gradient.

2) Sequential: This class implements a container for a list of layers. Its parameters are the list of layers and a variable that indicates whether the model is in training or testing mode. In the forward pass, it simply calls the *forward* method of each layer, propagating the output through the layers. In the backward pass, it propagates the gradient calling the corresponding method of each layer iterating through them in reverse order.

Likewise, the *params* method calls the *params* method of each layer and combines their outputs, returning a list of tuples containing the values and gradient for each parameter.

The *update* method iterates over all the layers calling each layer's corresponding method. If a learning rate γ is specified, then it is used to update all parameters. It is also possible to specify the *values* parameter that should be used for the update, instead of using the gradient computed during the forward pass. As previously mentioned, this is useful for implementing the Adam optimizer and will be explained later.

3) Dropout: We implement a dropout layer by defining a class *Dropout*. This layer is created taking as parameters the probability of dropping a hidden unit (p_{drop}) and whether the model is training. If it is not training, the forward and backward passes simply return their respective inputs.

The forward pass is implemented in the *forward* method. It creates a mask of hidden units to drop and saves it for the backward pass. The mask is used to shut down the corresponding units of the input data. Lastly, the result is normalized by $1 - p_{drop}$. The *backward* method carries out the backward pass, using the mask saved in the forward pass to shut down the same units of the gradient of the next layer. The result is again normalized by $1 - p_{drop}$.

B. Activation functions

The following activation functions are implemented in our framework:

1) ReLU: In the forward pass this activation layer applies the non-linear function $f(x) = max(0, x)$ to the input [3]. The backward pass is computed with the derivative:

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

2) Leaky ReLU: We implement a parametric Leaky ReLU, which takes takes a parameter α and computes the forward pass as:

$$f(x) = \begin{cases} x, & \text{if } x > 0 \\ \alpha \cdot x, & \text{otherwise} \end{cases} \tag{7}$$

In the backward pass, the gradient is given by:

$$f'(x) = \begin{cases} 1, & \text{if } x > 0 \\ \alpha, & \text{otherwise} \end{cases} \tag{8}$$

3) Tanh: We implement the hyperbolic tangent activation function in the class *Tanh*. This class computes in the *forward* method the $\tanh$ of the input, which is saved for the next step. In the backward pass step, we compute the gradient as $1 - \tanh^2 x$

4) Sigmoid: We implement the forward pass of this function using the $\tanh$ function to ensure that there is no overflow, according to the formula $\sigma(x) = 0.5 + 0.5 \cdot \tanh(x/2)$. We store the obtained value and then compute the derivative as $\sigma(x) \cdot (1 - \sigma(x))$.

C. Loss functions

Two loss functions are implemented in this framework: the Mean Squared Error (MSE) and the Cross Entropy Loss.

1) Mean Squared Error (MSE): In the forward pass, this module computes the difference between the output of the network $f(x, w)$ and the ground truth y and saves it for the backward pass. The loss returned is computed as:

$$Loss = \sum_{i=1}^{N} (y_i - f(x_i, w))^2 \tag{9}$$

The backward pass returns the gradient of the loss, which is computed using the stored difference between the output of the network and the ground truth:

$$\nabla_{Loss} = 2(y - f(x, w)) \tag{10}$$

2) Cross-entropy loss: This module computes the loss of the network in the forward pass as:

$$Loss = -\sum_{i=1}^{N} y_i \log(f(x_i, w)) + (1 - y_i) \log(1 - f(x_i, w)) \tag{11}$$

In the backward pass the gradient of the loss is computed with the following equation:

$$\nabla_{Loss} = y - f(x, w) \tag{12}$$

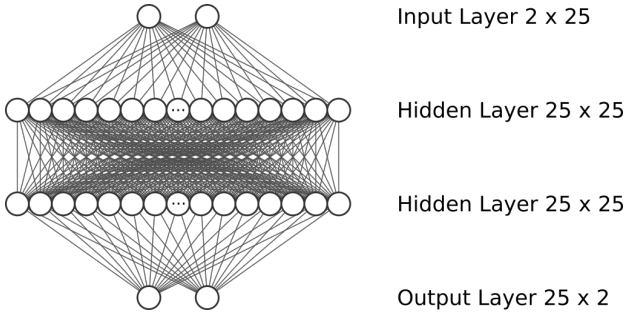


Fig. 1: Architecture of neural network

D. Optimizers

As explained in the beginning of Section II, the optimizers are defined using a class with a *step* method that updates the parameters and a method *adaptive_lr* that implements a generator that yields decaying learning rates. The decay is controlled by parameters κ and η according to:

$$\gamma^{(t)} = \eta \cdot t^{-\kappa} \tag{13}$$

where t is incremented every time the generator is called.

1) Stochastic Gradient Descent (SGD): The parameter update using Stochastic Gradient Descent calls the *update* method of the model (explained in Sections II-A1 and II-A2). This update can be done either with a fixed learning rate if κ and η are not passed as parameters when creating the optimizer, or with the described decaying learning rate if they are passed. In the second case, each time *step* is called the next learning rate is generated and passed to the *update* method of the model.

2) Adam: This class is implemented as described in [4]. Given the parameters of the optimizer settings β_1, β_2 and ϵ, at each timestep t we first calculate the gradient with respect to the stochastic objective g_t and then the first and second moment vectors are computed according to equations 14 and 15 respectively:

$$m_t = \beta_1 \cdot m_{t-1} + (1 - \beta_1) \cdot g_t \tag{14}$$

$$v_t = \beta_2 \cdot v_{t-1} + (1 - \beta_2) \cdot g_t^2 \tag{15}$$

With the previously obtained results we compute the *value* argument that is used to update the parameter, as given by:

$$\text{value} = \frac{m_t}{1 - \beta_1^t} \cdot \frac{1}{\epsilon + \sqrt{v_t/(1 - \beta_2^t)}} \tag{16}$$

Then, each parameter θ is updated as $\theta_t = \theta_{t-1} - lr \cdot \text{value}$. It is worth mentioning that this optimizer has a *restart* method that resets the value of t and the moments to 0 (which are also the values with which the optimizer is initialized).

The default values for the parameters are the ones used in [4], that is, a learning rate of 10^{-3}, and $\beta_1 = 0.9$ and $\beta_2 = 0.999$.

III. Experiments

A. Datasets

The train and test datasets are each composed of 1000 randomly generated samples. Each sample consists on two values chosen from a uniform distribution between 0 and 1, corresponding to a point in the 2D space. The samples that fall inside the disk of radius $1/\sqrt{2\pi}$ centered in $(0.5, 0.5)$ are labelled as 1's, whereas the rest are labelled as 0's.

B. Architecture

The architecture of the system in the experiments can be found in Figure 1. It has 2 input and 2 output units and 3 fully connected layers with 25 units each. The fully connected layers are followed by a dropout layer, whose drop probability is set on the experiments. The two first hidden layers use the same activation, while the last one may use a different one. These activation functions vary depending on the experiment.

C. Results

To test our framework, we evaluate the performance with three different experiments:

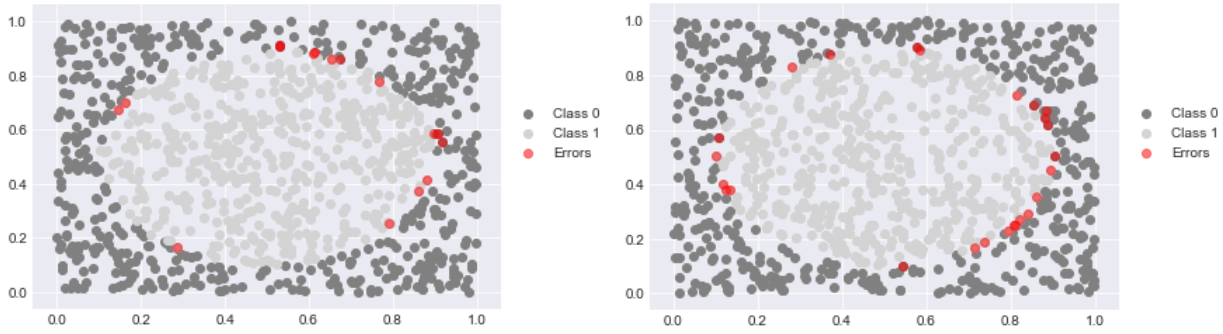
1) General performance: this experiment aims to compare the performance on the train and test sets, checking whether there is any overfitting and if dropout is needed.
2) Comparison of the optimizers: this experiment compares the performance in terms of speed and accuracy when using *Adam* or *SGD* as optimizers.
3) Comparison of the activation functions: similarly to the previous experiment, this one compares the performance with different activation functions (*Tanh*, *ReLU* and *Leaky ReLU*) for the first two layers.

All the experiments use a a batch size of 50 samples and train the model for 50 epochs.

General performance. For the first experiment, the model uses *ReLU* as the activation function for the first two fully connected layers and *Sigmoid* for the last one. No dropout is used - i.e. it has 0 drop probability. The optimizer used is *Adam* with a learning rate of 10^{-3}, and $\beta_1 = 0.9$ and $\beta_2 = 0.999$. The classification results obtained are shown in Figures 2a and 2b, for training and test respectively, with the misclassified samples in red.

The accuracy obtained for the test set is 97.5%, very similar to the 98.4% obtained for the training set. Hence, the model does not overfit, as could be expected since both sets are sampled from the same distribution. Therefore, we do not use dropout in any of the following experiments carried out. It is also worth noting that all misclassified points are located in the border area.

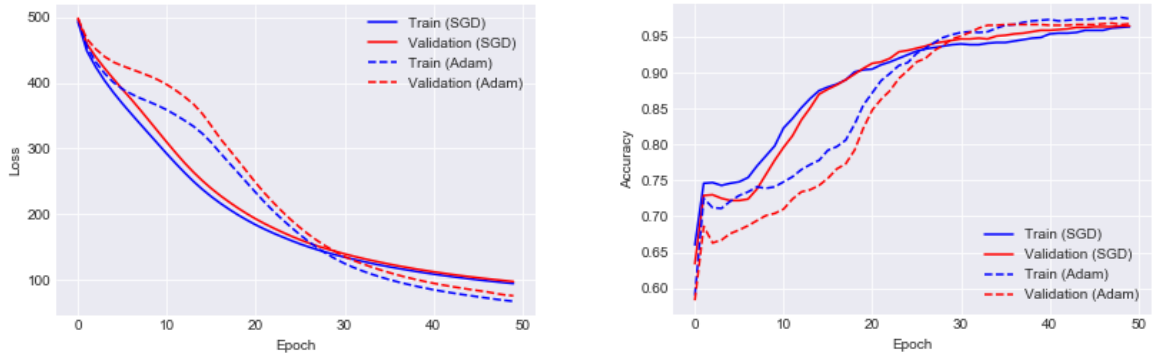
Comparison of the optimizers. In the second experiment we use the same architecture as in the previous one, but we train the model both with *Adam* and *SGD*. Adam uses the same parameters as before, while for *SGD* we use the adaptive learning rate given by equation 13, with $\kappa = 0.6$ and $\eta = 0.035$. In Figure 3 can be found the loss and accuracy for the optimizers per epoch. From these figures one can



(a) Training set - 98.4% accuracy.

(b) Test set - 97.5% accuracy.

Fig. 2: Classification results. Misclassified samples are shown in red, whereas correctly classified ones are shown in different gray levels depending on the class.



(a) Loss.

(b) Accuracy.

Fig. 3: Performance per epoch using SGD and Adam.

observe that during the first 20 epochs, the training with SGD is faster, with a steeper slope than the one found using *Adam*. However, this changes in the following epochs and by epoch 50 both accuracy and loss are slightly better when using *Adam*. Similarly to the previous experiment, there is no overfitting.

Comparison of the activation functions. Lastly, we compare the results obtained using different activation functions (*Tanh*, *ReLU* and *Leaky ReLU*) for the first two hidden layers using *Adam* as optimizer. The results obtained can be observed in Figure 4. In this plot we have omitted the test accuracy curves for simplicity since they were very close to their corresponding training accuracies. The accuracies obtained after 50 epochs are quite similar and do not seem to be strongly affected by the activation functions chosen for the first two activation layers. However, it is worth mentioning that *Tanh* learns the fastest at first, while *Leaky ReLU* seems to be the slowest.

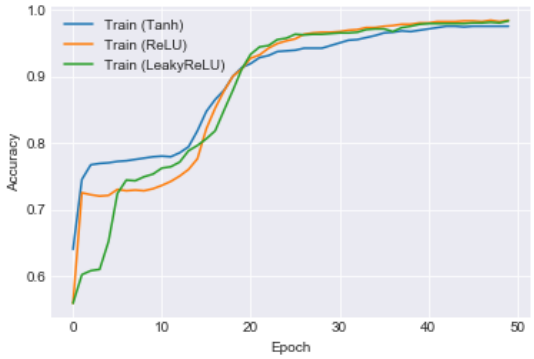


Fig. 4: Adam's accuracy with different activation functions.

IV. Conclusions

In this project we have implemented and used a simple deep learning framework for classifying the points that fall inside and outside a disk. The performance obtained with the chosen architecture was higher than 95% in both train and test sets after 50 epochs. Since both sets were sampled from the same distribution, there was no overfitting in the train data. The misclassified points are located in the border area, which was to be expected because a small variation in these points results in a change of their label – whereas small changes in points that are not in the border would not modify the labels. It should also be noted that by using a well-chosen learning rate – in this case a decaying one – *SGD* achieved a similar result to the one obtained with *Adam* width default parameters.

REFERENCES

[1] X. Glorot and Y. Bengio, "Understanding the difficulty of training deep feedforward neural networks," in *In Proceedings of the International Conference on Artificial Intelligence and Statistics (AISTATS10). Society for Artificial Intelligence and Statistics*, 2010.

[2] K. He, X. Zhang, S. Ren, and J. Sun, "Delving deep into rectifiers: Surpassing human-level performance on imagenet classification," *CoRR*, vol. abs/1502.01852, 2015. [Online]. Available: http://arxiv.org/abs/1502.01852

[3] V. Nair and G. E. Hinton, "Rectified linear units improve restricted boltzmann machines," in *Proceedings of the 27th International Conference on International Conference on Machine Learning*, ser. ICML'10. USA: Omnipress, 2010, pp. 807–814. [Online]. Available: http://dl.acm.org/citation.cfm?id=3104322.3104425

[4] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," *arXiv preprint arXiv:1412.6980*, 2014.