Project Two

Authors: Alec Flowers, Ali Falsafi, Rania Islambouli
Deep Learning EE-559, EPFL, Switzerland

May 28, 2021

I. Introduction

Deep learning, as a powerful and flexible tool for supervised learning, has evolved to where it is used in almost any software dealing with complex structured real world signals [B1]. The goal of this project was to implement a mini-deep learning framework using only pytorch's tensor operations and the standard math library [B2]. The main objective of this project is to learn the nuts and bolts of deep learning, from the distinct parts of a deep learning model to the math behind back propagation. In this project, we built the neural network building blocks from scratch and chained it together to build a deep learning model. The building blocks of our model are layers of perceptrons made up of fully connected linear layers followed by nonlinear activation functions and finally a loss function that quantifies the prediction error. Backpropagation based on the chain rule is implemented for our modules for calculating gradients based on key network parameters used for learning. Our implemented solver can perform gradient descent, stochastic gradient descent, and mini-batch gradient descent to train and update model parameters, enabling the model to learn. Lastly, we created training and testing functions which manage the entire training and testing process.

We use our framework to build a model and train on a classification task. The goal in classification is to best approximate some function f^* where $y = f^*(x)$ maps the input x to a label y. Our deep learning model $y = f(x; \theta)$ learns (adjusts) the parameters θ which best approximates f^*. [B3] The training and testing data are generated independently by sampling 1000 points uniformly on $[0, 1]^2$ with a label of 0 if outside the disk centered at (0.5, 0.5) of radius $1/\sqrt{(2\pi)}$ and 1 inside [B2] as shown in Figure 1.

Our framework is organized as follows:

- *module.py* - contains every class inheriting from the module class. These modules are the core of our framework and include linear layers, non-linear activation functions, loss functions, and a sequential class for stacking the layers.
- *solvers.py* - contains one solver that can perform gradient descent, mini-batch gradient descent and stochastic gradient descent depending on the input parameters.
- *train.py* - contains functions that manage the training and testing of the deep learning models.
- *test.py* - run file built to the project specifications.
- *utils.py* - utility file that contains functions to generate training and testing data, calculate errors and save models.

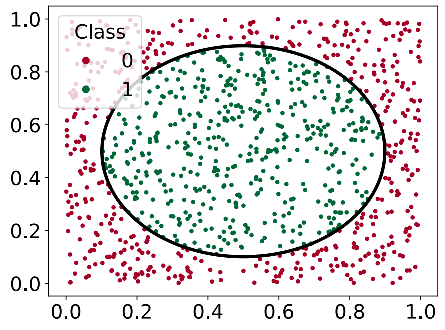

Fig. 1: Distribution of the generated data, color-coded with the class label. The black line shows the boundary between the classes.

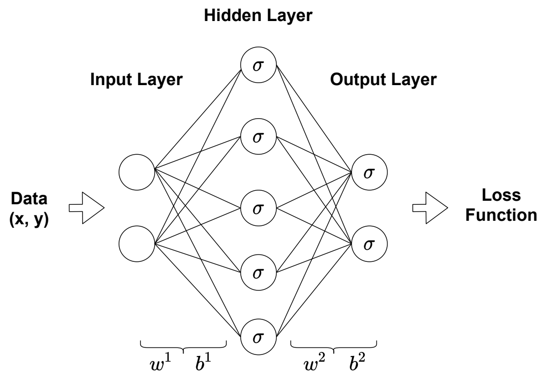

Fig. 2: Visualization of the example fully connected network [Linear(2,5), Relu(), Linear(5,2), Relu()]. σ represents the ReLU activation function.

II. Modules

Modules allow us to chain together operations necessary to build an entire neural network. Each module has a function for managing the forward pass - forward(), backward pass - backward() and listing parameters - get_param(). In our framework we have 4 types of modules:

a) Sequential: To create a network, you create a list of modules in sequence. The aptly named Sequential module is passed this list of modules and is in charge of iterating through the list for the forward pass, iterating in reverse for the backward pass, and returning all the parameters for updating.

For example Sequential([Linear(2, 5), Relu(), Linear(5, 2), Relu()]). (See Figure 2 for a visual.) This follows the standard format for a multi-layer perceptron which is:

$$X^{(l)} = \sigma(W^{(l)}X^{(l-1)} + b^{(l)}) \qquad \forall l \in \{1, 2, ..., L\} \qquad (1)$$

Where L is then number of layers, $W^{(l)} \in \mathbb{R}^{d_l \times d_{l-1}}$ $X^{(l-1)} \in \mathbb{R}^{d_{l-1}}$ and $b \in \mathbb{R}^{d_l}$. In the forward pass, each module applies a transformation defined in the forward() function and returns an output which can then be passed to the next module.

b) Linear: The Linear module is very important in our framework as it is the only one with learning parameters, i.e. parameters that can be updated. A forward pass for the linear module in matrix notation is:

$$S^{(l)} = X^{(l-1)}(W^{(l)})^T + b^{(l)} \qquad \forall l \in \{1, 2, ..., L\}. \tag{2}$$

Where L is then number of layers, $W \in \mathbb{R}^{d_l \times d_{l-1}}$ is a matrix of weights $b \in \mathbb{R}^{d_l}$ is a vector of biases and $X \in \mathbb{R}^{n \times d_{l-1}}$ where n is the number of samples in the mini-batch. In this case b has to be broadcast (copied) in order for the math to work out. In order to make the matrix multiplication compatible with mini-batches a tweak is applied in Equation 2 compared to Equation 1. This makes linear modules able to handle multiple batches of input at once which manifests itself as multiple rows in the X matrix. The weights and the biases are randomly initialized by sampling from a Gaussian distribution of mean 0 and standard deviation 1.

c) Non-Linear activation: The non-linear activation functions are typically applied element wise to the output of the linear module. It is important that they are non-linear, otherwise the network would just be an affine mapping and equate to running a simple linear regression. The most common activation functions used in the hidden layers are ReLU and Tanh. ReLU applies the formula

$$ReLU(x) = max(0, x) \tag{3}$$

and helps promote sparsity and deal with the vanishing gradient problem. However, sometimes ReLU can cause too many neurons to output 0 which is where LeakyReLU comes in with the formula:

$$LeakyReLU(x) = max(-x \cdot 0.01, x) \tag{4}$$

as doesn't set all negative values to 0. Tanh applies the formula

$$Tanh(x) = \frac{(-1 - e^{-2x})}{(1 + e^{-2x})} \tag{5}$$

If there is a single output for a classification task, it is common to use the Sigmoid activation function on the final layer which pushes the output to be between [0,1] and can be thought of as a probability. For multi-class classification Softmax is used on the final layer instead and applied over the entire output vector.

$$\text{Softmax}(x_i) = \frac{e^{x_i}}{\sum_{j=1}^{n} e^{x_j}} \tag{6}$$

It forces the entire output vector to sum to 1 and can also be thought of a class probability.

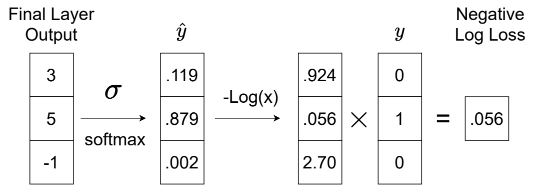

Fig. 3: Simple example of cross entropy. Notice that the elements of $\hat{y}$ sum to 1 and the final output only depends on the correct class.

d) Losses: The goal of optimizing a neural net is to minimize the error of the cost or objective function. Mean squared error is the average of the squared distance between the networks predictions and the true labels. As a formula:

$$\frac{1}{N}\sum_{i=1}^{N}(\hat{y}_i - y_i)^2 \tag{7}$$

Where $\hat{y}$ represents the prediction, y represents the true labels and N the number of samples. The further away our output is from the truth, the larger the error. Cross entropy loss (an example depicted in Figure 3) is the combination of Softmax on the final output layer with negative log likelihood loss whose formula is:

$$\sum_{i=1}^{N} -y_i \cdot log(\hat{y}_i + \epsilon) \tag{8}$$

where ϵ is a very small value that avoids taking the log of 0. Cross entropy loss is often used when working on multi-class classification tasks where the true labels are one-hot encoded. It only rewards or penalizes the network for its output on the correct class.

In our framework we built it so that the losses are not passed as part of the module list into the Sequential module. The first reason is because the losses need two pieces of information, the output of the neural network as well as the true labels, while all the other modules just need the output from the previous module. It makes more sense to have the loss separate where it is easy to pass in the target labels and the output. The second reason is so that the losses are modular and can be quickly swapped in and out. They are not inherently tied to the structure of the neural net and so it makes sense to keep the losses separate. However, this means to begin the backward pass we must first call backward() on the loss and pass this result to the backward loop of the Sequential module.

III. Backpropogation

In order to train the neural net we are trying to minimize the loss over the training set. To minimize the loss we need to calculate the gradient of the loss with respect to our parameters and update the parameters. Because a neural net can be seen as a composition of functions we use the chain rule to calculate these gradients. Every module in the backward pass is built to accept a *gradwrtoutput* parameter, which is the gradient information from the previous module -in the backward pass-, apply its own gradient, and return the output. The Sequential module handles the backwards pass by iterating through the modules in reverse and passing the gradient information back

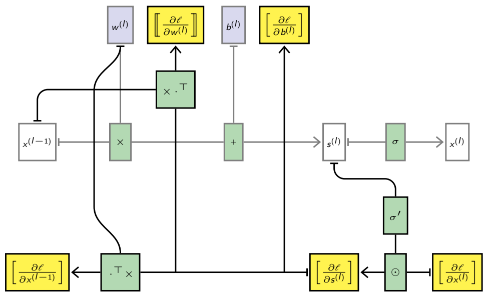
$w^{(l)}$
$\left[\left[\frac{\partial \ell}{\partial w^{(l)}}\right]\right]$
$b^{(l)}$
$\left[\frac{\partial \ell}{\partial b^{(l)}}\right]$
$\times \cdot^{\top}$
$x^{(l-1)}$
$\times$
$+$
$s^{(l)}$
σ
$x^{(l)}$
σ'
$\left[\frac{\partial \ell}{\partial x^{(l-1)}}\right]$
$\cdot^{\top} \times$
$\left[\frac{\partial \ell}{\partial s^{(l)}}\right]$
$\odot$
$\left[\frac{\partial \ell}{\partial x^{(l)}}\right]$

Fig. 4: The chain-rule graph depicting back-propagation of fully connected layer. [B4]

up through the loop. For the Linear module we need to take the derivative of Equation 2 w.r.t the weights and the biases:

$$\left[\frac{dL}{dw^{(l)}}\right] = \left[\frac{dL}{dS^{(l)}}\right]^T x^{l-1} \qquad \left[\frac{dL}{db^{(l)}}\right] = \left[\frac{dL}{dS^{(l)}}\right] \tag{9}$$

We used Figure 4 when computing the derivatives for the backward() functions as it shows exactly which information needs to be passed where in a multi-layer perceptron.

IV. Solvers

The main algorithm for non-convex optimization is gradient descent and its variants. If batch size = 1 this is stochastic gradient descent, 1 < batch size < N is batch-gradient descent, and batch size = N is gradient descent. These algorithms control how our parameters are updated. The formula for gradient descent is:

$$w^{(l+1)} \longleftarrow w^{(l)} - \lambda \left[\frac{dL}{dw^{(l)}}\right] \qquad b^{(l+1)} \longleftarrow b^{(l)} - \lambda \left[\frac{dL}{db^{(l)}}\right] \tag{10}$$

where λ is the step size and we update using Equation 9. The solvers interact with the Sequential module as they need the parameters from all the modules in order to apply the update. Sequential provides this by calling get_param() and returning a list of all the parameters from all the modules. The solver then iterates through this list and applies the update according to Equation 10. Note that many of the modules do not have any parameters to update and therefore return None for the parameter list.

V. Training and Test

The training file provides an example of how to connect the various items in the framework we built in order to train and test a neural network.

1) Create a list modules to pass into the Sequential module.
2) Initialize the loss function you want to use.
3) Choose hyperparameters such as number of epochs, batch size, and learning rate.

For each batch we run the training data through a forward pass by calling forward() on our sequential module. Then we calculate the loss by comparing the output of the network with the true labels. We have to reset all the gradient parameters to avoid accumulating unwanted gradients from previous batches.

Table I: Table of results using our home-built framework to train neural networks on the given task. LRelu() stands for LeakyRelu.

	Structure	**Hyper parameters**	**Final Accuracy**
Vanilla Network	[Linear(2,25), Relu(), Linear(10,25), Relu(), Linear(25,25), Relu(), Linear(25,2)] Loss = MSE	batch size = 5 lr = 1e-4 epochs = 200	Train 83.8 % Test 80.3 %
Other Network	[Linear(2,10), LRelu(), Linear(10,50), LRelu(), Linear(50,100), LRelu(), Linear(100,100), LRelu(), Linear(100,50), LRelu(), Linear(50,10), LRelu(), Linear(10,2)] Loss = Cross Entropy	batch size = 5 lr = 1e-4 epochs = 200	Train 99.7 % Test 96.7 %

Next, we apply a backward pass to calculate and store the gradients w.r.t the new loss by first calling backward() on our loss and passing this into the backward() function in our sequential module. Finally, we take a step where the parameters are updated based on a step size. Once all the batches have been run this completes one epoch.

In order to observe the loss and accuracy while training we save losses as well as the accuracy and print a summary every 10 epochs. We calculate the accuracy using a utility function that returns the total number of errors made every batch divided by the total number of inputs given in that batch. Testing is quite simple as there is no backward pass necessary. We generate a new independent sample of data, apply a forward pass through our network and then calculate the error and accuracy.

VI. Results

We now take our deep learning framework and apply it to the given classification task we introduced in the introduction. We create two networks, one given in the project requirements of having two input units, 3 hidden layers of 25 units, 2 output units and using MSE loss. The other we created ourselves and used Cross Entropy loss. Refer to Table I for details.

VII. Conclusion

In this project, we successfully implemented a mini-deep learning framework that you can use to create a neural network, select a loss function and optimizer, and train then test on a dataset. We have also tested the implementation of our home-baked deep learning framework by successfully building a model to classify the generated points according to their position.

References

[B1] F. Fleuret, Deep learning 1.2 current applications and success, https://fleuret.org/dlc/materials/dlc-slides-1-2-current-success.pdf, accessed: 2021–05-27 (2021).

[B2] F. Fleuret, Deep learning mini-projects, https://fleuret.org/dlc/materials/dlc-miniprojects.pdf, accessed: 2021–05-27 (2021).

[B3] I. Goodfellow, Y. Bengio, A. Courville, Y. Bengio, Deep learning, Vol. 1, MIT press Cambridge, 2016.

[B4] F. Fleuret, Deep learning 3-5 gradient descent, https://fleuret.org/dlc/materials/dlc-slides-3-5-gradient-descent.pdf, accessed: 2021–05-27 (2021).