Deep Learning - Lab Exercise 2

Authors : Maxime Jauryon, Gabriel Becquet, Mahdi Ranjbar
Date : 24th March 2023

I. Goal of the lab exercise

In this Deep Learning lab, we worked on building an automatic differentiation library using Python. Automatic differentiation is a key concept in Deep Learning. The gradients are used to adjust the weights of the network using the backpropagation algorithm to minimize the loss function. In the first exercise, we built a simple linear classifier, and in this one we will build an automatic differentiation library to train deeper neural networks. This automatic differentiation library will be based on PyTorch's autograd mechanism, simplified for the purpose of this exercise.

II. Neural network architecture description

A. Computation nodes

We made an abstraction for computational nodes to do automatic backpropagation. By storing gradients and operations in objects, we can easily compute complex function gradients without explicit derivation. We made a generic computational node (Tensor) and a parameter-specific one (Parameter) with a name and velocity (which cannot be backpropagated). This is important for deep neural networks, which have many layers and parameters that would be hard to handle manually without backpropagation.

We wrote functions to work with tensors (excluding softmax) and create backward information. The output tensor must have require _grad = True if any of the inputs require gradient. These functions will be used to build the neural network and calculate backpropagation.

- "*ReLU*" and "*Tanh*".They are commonly used in deep learning to introduce nonlinearity in neural networks. In our case, we have implemented these functions to be used in the framework of non-linear activation function.

- The affine transformation operation involves computing the dot product between input and weight matrix, then adding a bias term. We implemented this similarly to the first exercise, but with an additional step to calculate the gradient with respect to x.

- Finally, we have implemented functions that are basic mathematical operations needed to build the end of the deep neural networks. The "*_softmax*" function is an activation function used to normalize a vector of scores into a probability distribution. The "*nll*" (negative log-likelihood) function is used to calculate the loss of a multi-class classification model that predicts a probability distribution over several classes.

B. Module

In this part, we have implemented two important classes: "*Module*" and "*ModuleList*".They have a parameters() function that retrieves all the parameters of the network, including those stored in the submodules (usefull to give them to the SGD for exemple).

This is crucial for updating parameter values during neural network training. We store these parameters in Module or ModuleList objects, allowing us to easily navigate the network and retrieve parameters for updates. This simplifies the management of complex neural networks, which may have multiple layers and nested sub-modules.

C. Initialization and Optimization

Three weight initialization methods are proposed: zero initialization ("*zero_init*"), Glorot initialization ("*glorot_init*"), and Kaiming initialization ("*kaiming_init*"). These methods set the initial values of the weights according to specific rules, which have been shown empirically to promote model convergence and performance. The initialization of the weights is an important step when training neural networks because it can have a significant impact on the performance of the model.

Next, a simple optimization class is implemented: stochastic gradient descent (SGD) :

$$\theta_{t+1} = \theta_t - \epsilon_t \nabla_\theta L$$

This class contains two functions: "*step()*" and "*zero_grad()*". The "*step()*" function performs a gradient descent step by updating each parameter of the model with its gradient and the defined learning rate. The "*zero_grad()*" function sets the gradients of all model parameters to zero before the next gradient descent step. The choice of optimizer determines how the weights are updated at each training step, like the momentum SGD : (implemented at the end of the notebook)

$$\gamma_{t+1} = \mu\gamma_t + \nabla_\theta L$$
$$\theta_{t+1} = \theta_t - \epsilon\gamma_{t+1}$$

III. Networks and Training Loop

A. Simple linear classifier

In this first step, we create a simple linear neural network, which consists of a linear layer with an input of dimension "*dim_input*" and an output of dimension "*dim_output*". To create the parameters of the network, we initialize the weight matrix W and the bias vector b using the functions "*glorot_init*" and "*zero_init*", respectively. The forward method takes an input x and computes the output of the linear layer using the "*affine_transform*" function.

B. Training loop

This step is crucial for training neural networks on the training data. The loop iterates through the data multiple times (epochs), calculating the loss for each data point. Then, it uses backpropagation to calculate gradients and the optimizer to update the network weights.

In addition, our code computes the validation set accuracy every time a certain amount of training data is processed (5 times per epoch). After training, it also calculates the accuracy on the test set to evaluate the model's overall performance.

At each epoch, it shuffles the training data, and then for each training example, it performs the following steps:

- Computes the network output for the current example (*network.forward(x)*).
- Computes the cost (or loss) associated with the network output for this example (*nll(z, _gold)*).
- Reinitialize the gradient of the parameters (optimizer.*zero_grad()*).
- Performs gradient backpropagation to compute the gradients with respect to the network parameters (*l.backward(1.)*).
- Uses the optimizer to update the network parameters (*optimizer.step()*).

C. Linear network to deep network

The last step consists in passing from a linear neural network to a deep neural network. To do this, we create a new *DeepNetwork* class and which takes as parameters the number of layers n_layers, the input dimension *dim_input*, the output dimension *dim_output*, the hidden dimension *hidden_dim* as well as a boolean tan which indicates whether we want to use the hyperbolic tangent activation function or the relu activation function.

In this new class, we use a *self.W* parameter list to store the weight matrices of each layer (except last), and a *self.b* parameter list to store the corresponding bias vectors (except last). The first hidden layer has as input the input data x and as output the value obtained after an affine transformation applied to the input data, followed by an activation function. The output of each hidden layer is then used as input for the next layer until the output layer is reached. This last layer uses an affine transformation to give the final output of the network.

The weights of the matrices of each hidden layer and the bias vector of the output layer are initialized using Glorot's method and by initializing the bias vectors to zero.

The value of moving to a deep neural network is to be able to learn more complex hierarchical representations from raw data. Each hidden layer can be seen as a transformation function of the input data, allowing the extraction of more and more abstract features as one progresses in the network. This allows to model more complex non-linear relationships between input and output variables (going from 91% to 97% accuracy).

IV. Results

A. Linear network

The network is initialized with an input layer of dimension 28x28 (for 28x28 pixel images) and an output layer of dimension 10 (for the 10 possible classes). The stochastic gradient optimizer is used with a learning rate of 0.01.

The results show that the network improves over the epochs, with a decrease in the average loss and an improvement in performance on the validation (Dev) dataset (see Table 1). The network achieves a test accuracy of 92.1%. However, there is still room for improvement, which justifies the use of a deeper neural network.

Epoch 0	Mean Loss : 1.632340 Best Dev Accuracy : 0.919900 Train Accuracy : 0.908740
Epoch 1	Mean Loss : 1.592537 Best Dev Accuracy : 0.919400 Train Accuracy : 0.918560
Epoch 2	Mean Loss : 1.585946 Best Dev Accuracy : 0.924600 Train Accuracy : 0.919080
Epoch 3	Mean Loss : 1.582328 Best Dev Accuracy : 0.922300 Train Accuracy : 0.922780
Epoch 4	Mean Loss : 1.579504 Best Dev Accuracy : 0.924100 Train Accuracy : 0.925040
Test Accuracy :	0.921000

Table I: Training results of linear network

B. Deep network

We created a deep network with 2 layers, with dimension 100 in hidden layer (but here there is none but it still

apply between the 2 layers : input_dim X hidden_dim to hidden_dim X output_dim). For training, we used the SGD algorithm. We performed two trainings by changing the option "tan=True".

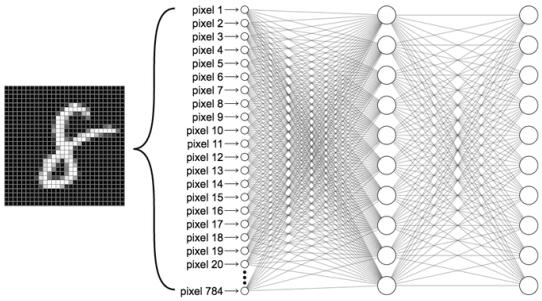

Figure 1: Neural Net Architecture

From the results obtained, the deep network model with the tanh activation function gave similar results to the model with ReLU, but required more iterations to converge to similar accuracy (see Table 2 and 3). The deep network achieved a test accuracy of 97.36%, this shows that using a deep neural network is obviously better.

Epoch 0	Mean Loss: 1.571504 Best Dev Accuracy: 0.962200 Train Accuracy: 0.959860
Epoch 1	Mean Loss : 1.512890 Best Dev Accuracy : 0.969000 Train Accuracy : 0.977940
Epoch 2	Mean Loss : 1.499349 Best Dev Accuracy : 0.972200 Train Accuracy : 0.977720
Epoch 3	Mean Loss : 1.491334 Best Dev Accuracy : 0.975900 Train Accuracy : 0.984520
Epoch 4	Mean Loss : 1.485290 Best Dev Accuracy : 0.976700 Train Accuracy : 0.986360
Test Accuracy :	0.971800

Table II: Training results of deep network (ReLU)

Epoch 0	Mean Loss: 1.583031 Best Dev Accuracy: 0.954400 Train Accuracy: 0.952500
Epoch 1	Mean Loss : 1.521072 Best Dev Accuracy : 0.967500 Train Accuracy : 0.973280
Epoch 2	Mean Loss : 1.504079 Best Dev Accuracy : 0.970600 Train Accuracy : 0.980100
Epoch 3	Mean Loss : 1.495193 Best Dev Accuracy : 0.972400 Train Accuracy : 0.984500
Epoch 4	Mean Loss : 1.488635 Best Dev Accuracy : 0.975900 Train Accuracy : 0.986520
Test Accuracy :	0.973600

Table III: Training results of deep network (tanh)

V. Conclusion

In this lab, we implemented two types of neural networks for MNIST image classification: a simple linear network and a deep network. This lab allowed us to learn the necessary steps to build a neural network, we have noticed the importance of network depth to obtain better classification results.

VI. Bonus

For the momentum SGD, we simply used the velocity attribute from the parameters to change the rule of the update function. As we expected, we were able to converge faster than the SGD as shown in the following table.

Epoch 0	Mean Loss: 1.570353 Best Dev Accuracy: 0.960500 Train Accuracy: 0.962080
Epoch 1	Mean Loss : 1.511513 Best Dev Accuracy : 0.968300 Train Accuracy : 0.979140
Epoch 2	Mean Loss : 1.498082 Best Dev Accuracy : 0.972500 Train Accuracy : 0.983440
Epoch 3	Mean Loss : 1.490860 Best Dev Accuracy : 0.975500 Train Accuracy : 0.984660
Epoch 4	Mean Loss : 1.484879 Best Dev Accuracy : 0.975200 Train Accuracy : 0.989980
Test Accuracy :	0.976100

Table IV: Training results of deep network (ReLU) with MomentumSGD