Deep Learning - Assignment 1

Names: Daniel Szelepcsenyi, Marie De Mey, Theo Guegan, Pranav Panday

Title: Deep Learning Assignment 1

Course: SYDE 577

Professor: Dr. Bryan Tripp

1) Design Overview

The goal of this assignment was to develop an autodifferentiation package based on NumPy. Inspired by MiniGrad, the package has two main modules: the tensor module and the neural network module [1]. The tensor module creates the individual nodes, which allow the neural network module to connect these nodes, creating the full network.

1.a) Tensor Module

The tensor module, implemented in `tensor.py`, was inspired by the Tensor datatype in PyTorch. The `Tensor` class takes in a NumPy array as an argument and initializes attributes for the associated gradient, the child nodes that it was created by, and a backward function depending on the operation that created it. The `_backward` function defines how the gradient is passed back to child nodes of the tensor during backpropagation. This method allows the use of the chain rule efficiently by propagating the gradient through the network. Before backpropagation, all gradients are reset to zero to ensure proper accumulation of the gradients. Backpropagation is performed by creating a topological order of all the tensors and visiting them recursively in inverted order.

For example, if `c = a + b` and `d = ReLU(c)`, reversing the topology will result in an order of `d, c, b, a`.

The tensor class has functions to override the built-in operations for addition, multiplication, exponentiation, and matrix multiplication, to work with tensors and define the `_backward` function. The `_backward` function is overloaded for each operator to assign the gradient with respect to the parameter of the tensor and also to the child one.

The tensor class also defines two non-linear activation functions: ReLU and Sigmoid, and their derivatives.

Here is a sample of the ReLU gradient calculation used in its `_backward` function:

```
a_grad = c.gradient * (self.data > 0)  # compute derivative with respect to self (a)
self.gradient = self.gradient + a_grad  # accumulate the gradient
```

To illustrate this class's functionality, take the loss tensor as an example: $L = \frac{1}{2}(\hat{y} - y)^2$ with $\hat{y} = (h_1^2 w_1^3 + h_2^2 w_2^3) + b_1^3$. In this example, y and $\hat{y}$ are declared as child nodes of the square tensor and are saved in the `_previous_nodes` attribute as a set. Figure 1 illustrates the computational graph developed during forward propagation and backpropagation. As seen, every operation creates a new node. The gradient value "grad" or δ is calculated during the backward pass by reversing the depicted connections. The green boxes in Fig. 1 indicate the resulting value the weight will be updated by, and the δ values computed by hand in class that are required for earlier nodes in the network are shown in red boxes.

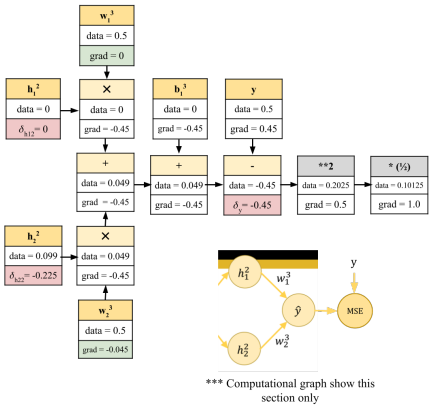

Figure 1: Computational graph of nodes

1.b) Neural Network Module

The neural network module, implemented in `nn.py`, is used to construct multi-layer fully connected networks using the Tensor class as building blocks. Inspired by PyTorch, it builds a neural network by stacking multiple linear modules in the sequential module.

The module defines a base class called `Module`, which defines the methods `zero_grad` and `parameters`. This base class is inherited by the rest of the classes defined in the module, including the `Linear`, `ReLU`, `Sigmoid`, and `Sequential` classes. Overloading the `__call__` function defines the behavior of the network during the forward pass by calling the object with the input as a parameter.

The `Linear` class takes the number of input and output features (`in_features`, `out_features`) and initializes a weight tensor of size `[in_features, out_features]` following the Kaiming initialization discussed in the lectures. The bias of the neurons is initialized as a tensor of size `[1, out_features]` and set to zero. The weights and biases are saved as parameters of the module. The `__call__` function for the Linear class implements the forward pass: (`output = X@W + B`).

The `ReLU` and `Sigmoid` classes override the `__call__` function to call the tensor module implementation for the ReLU (`x.relu()`) or sigmoid (`x.sigmoid()`) functions, respectively, during the forward pass in the network.

The `Sequential` module allows multiple layers to be stacked and takes in a tuple of desired layers. The `__call__` function loops through the tuple of objects and feeds each module with the output of the previous one.

2) Assignment Task Script (Model Training)

The script begins by importing our package, datasets, and pickle data file to set the weights, biases, inputs, and target values. The model is built using a fully connected feed-forward neural network made with the Sequential module as follows:

```
model = Sequential(
  Linear(2, 10),
  ReLU(),
  Linear(10, 10),
  ReLU(),
  Linear(10, 1)
)
```

The input layer accepts two features, and the output layer produces a single scalar prediction ($\hat{y}$). The forward pass is done by feeding the input to the model (`y_hat = model(first_input)`), and the loss, in tensor form, is then calculated using MSE:

```
loss = ((y_hat - first_target) ** 2) * 0.5
```

Afterward, the model's gradients are set to zero (`model.zero_grad()`) before a backward pass is performed on the loss tensor (`loss.backward()`). This process is first performed for only the first input-target pair to calculate the first-layer gradients and is then repeated in a training loop with the parameters adjusted each epoch according to the gradients and learning rate:

```
p.data -= learning_rate * mean_gradient
```

3) Results

```
Gradients of the first-layer weights:
 [[-0.0210 -0.0184  0.      0.0178  0.      0.
  -0.0097  0.      0.      0.    ]
 [-0.0926 -0.0815  0.      0.0783  0.      0.
  -0.0430  0.      0.      0.    ]]

Gradients of the first-layer biases:
 [-0.2151 -0.1893  0.      0.1820  0.      0.
  -0.0998  0.      0.      0.    ]
```

Figure 2: First-layer weights and biases of the untrained network for the first (input, target) pair in the training dataset

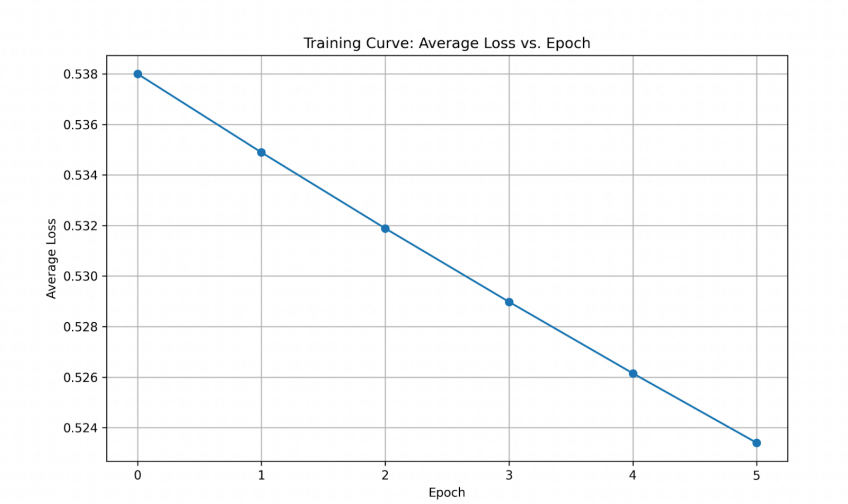

Figure 3: Training curve with average loss over the dataset before any updates (with initial parameters) and after each update

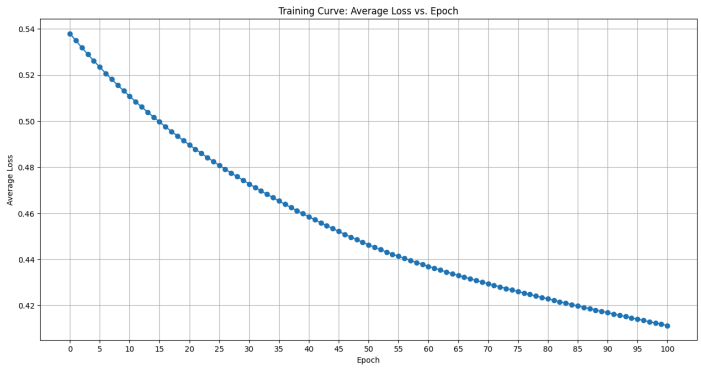

Figure 4: Training curve over a higher number of epochs to verify that the average loss decreases as the epoch number increases, thus verifying that the model is learning correctly

Bibliography

[1] kennysong, *MiniGrad*. [Online]. Available: https://github.com/kennysong/minigrad