Project
Learn OOP The Hard Way
Modeling and Implementing a Deep Learning Library

Important notes

This is an OOP project with a machine learning flavor. Therefore, particular attention will be given to the design/modeling and elegance of your solution, which has to leverage all the power of OOP principles and Python features (decorators, properties, static/class methods, etc.) seen during the course (and beyond). Note that to reach such a solution (cleanness of the design and elegance of the solution), it requires many iterations, so do not optimize for cleanness and elegance immediately.

Along the development iterations, you will make design decisions (trading between advantages and disadvantages) about your solution. We ask you to keep track of all these design choices and explain them in your report. It is also advised to keep track of the successive versions of your class (and object) diagram in a folder `/docs/diagrams/` with a naming convention like `class-diagram-v*.png` (where * corresponds to the version number starting from 0).

Regarding the report, it should provide at least the following elements, but you are free to be creative:

1. A class diagram that provides a high-level overview of your solution along with a section that provides the general principles (or philosophy) of your approach to modeling the library;
2. A series of subsections focusing on different parts of your model with a detailed description of these parts accompanied by **some portions** of code (of **particular interest**). You can also talk about the history (the successive iterations) that led you to that specific modeling;
3. A section dedicated to the evaluation of your library, including, the description of the different tests you performed on the library and the results that you obtained. You are able to perform comparisons with the performance that you would get with the original Pytorch library, so do not hesitate to draw comparative charts regarding computation time, convergence of the results, etc.;
4. A section dedicated to the limits of your model and implementation: what didn't work? why (e.g., which design choices led you to that)? and what can you do about it if you have a chance to continue working on this project?

Don't use ChatGPT to generate your report. Any abuse will not be tolerated. Be concise in your writing: we do not ask you to write a novel but simply what you did.

All developments will be done in the GitHub organization dedicated to this course. Make regular commits to your respective repositories. Recall that a commit should correspond to implementing/solving/fixing a very specific aspect in your code or model. Recall also that it is essential to provide clear messages for your commits. This way, one can retrieve commits related to a particular aspect and ease version control (e.g., reverting changes).

The repositories will be private. Do not share your models nor implementation with other groups. Chances are that you will end up with similar code. Again, any abuse will not be tolerated.

The project has to be done by groups of two to three students.

The Core Artificial Neural Network Library

Important: even if we already discussed one possible model for this part of the library, you are free not to stick to it and to provide yours (which fits your own approach or philosophy to modeling the library).

This series of YouTube videos Neural networks - YouTube is a very interesting entry-level resource to learn more about neural networks. In particular, the first and second videos show in a detailed manner the structure of neural networks and what it means for these neural networks to learn.

A linear layer applies an affine linear transformation to the incoming data: `y = xA^T + b`. It is characterized by the size of its inputs and outputs. It also contains a set of weights and, optionally, another set of biases, corresponding to the `A` and `b` of the affine linear transformation, respectively.
Weights and biases are learnable parameters. These parameters are a type of tensors, which are treated differently when added to the artificial neural network.

Tensors are multi-dimensional matrices containing elements of a single data type. They have special properties like `requires_grad`, `is_leaf`, etc., which are essential for the construction of the backward computational graph and gradient computation (we will see it later in the course!). Note that the data are also tensors.

An artificial neural network is composed of a set of linear layers. An artificial neural network can be in one of two modes: *training* or *evaluation*. The artificial neural network determines how the linear layers are applied to the incoming data.

A loss is computed between an input and a target value. The *mean square error* and the *negative log-likelihood loss* are examples of such a loss. The computed loss is used to compute a gradient for each learnable parameter.

An optimization algorithm or *Optimizer* is used to update the learnable parameters of an artificial neural network. It is characterized by a learning rate and momentum. Stochastic gradient descent (SGD) is an example of such an optimization algorithm. Each call to the Optimizer performs one step of the algorithm.

1. Analyze the provided specification and determine your system's perimeter (fences).
2. Identify the entities and model them in the form of classes.
3. Provide an object diagram. Do you notice any inconsistencies? If yes, refactor your model.
4. Now, link the different entities together to form a complete class diagram.
5. Provide an object diagram. Again, do you notice any inconsistencies? If yes, refactor your model.
6. Translate your model into Python classes.

Datasets, Dataloaders, and Transforms

Complete your model with the appropriate associations.
We now want to extend our library with *datasets*, *dataloaders*, and *transforms*:

- A *dataset* stores the samples and their corresponding labels. At creation, one can specify a set of transformations to apply to the dataset's examples. MNIST is a dataset that your library could provide;
- A *dataloader* wraps an iterable around the Dataset to enable easy access to the samples. The dataloader will return examples in the form of batches of a certain size;
- A *transform* is a data manipulation process, like normalization or standardization, that is applied to the features of the examples inside a dataset. Transforms can equivalently be applied to the labels.

Provide an extension of the class diagram for these new entities.

The Artificial Neural Network Library from the user's perspective

Now, we turn to how the end-users should use your artificial neural network library. Here is how a neural network is defined in a "*pythonic*" way and trained (note that this is also how PyTorch is roughly used to define a neural network):

```
class Model(Module):
      def __init__(self) -> None:
      (1)     self.linear1 = Linear(32*32, 20)
              self.linear2 = Linear(20, 10)

   (4) def forward(self, x):
              # apply the linear transformations of the neural
              # network to the incoming data, x.
              x = self.linear1(x)
              return self.linear2(x)

model = Model()
print(model) (6)
optimiser = SGD(model.parameters()(2), lr=0.01, momentum=0.9)
loss_fn = MSE()

dataset = MNIST()
dataloader = DataLoader(dataset, batchsize=32, transform=None)

for x, y in dataloader:(5)
      # Zero your gradients for every batch!
      optimizer.zero_grad()
      y_hat = model(x)(3)
      loss = loss_fn(y, y_hat)
      loss.backward()
      optimizer.step()
```

Extend your Python implementation of the artificial neural network library with appropriate magic (dunder) methods to be able to define a neural network in a similar way. In particular, we want:

- When you declare a new layer, e.g., `self.linear1= Linear(32*32, 20)` 1 , the parameters corresponding to this layer should be registered in a dictionary inside the model, which the optimizer can access later via the `.parameters()` method 2 .
- Applying the model to the data inputs 3 should call the method `.forward()` 4 .
- Iterating over a dataloader 5 should return a sequence of batches of examples.
- Redefine the behavior of print() 6 so that one can visualize the neural network's architecture.

The Automatic differentiation Engine

If you take a close look at the code provided in the previous section, you can see the for loop going through the entire dataset. This is where the magic … oops the learning is happening. In this part of the code, you can see in particular:

- ```
  y_hat = model(x)
  loss = loss_fn(y, y_hat)
  ```
  These two lines correspond to the forward pass where the data are passed through the model in order to compute the model's loss signal.

- ```
  loss.backward()
  optimizer.step()
  ```
  These two lines correspond to the backward pass where we compute the gradient of the loss with respect to the model's parameters (weights and biases) and use the gradient to adjust the model's parameters via gradient descent.

Together, these steps form what we call the backpropagation algorithm, which is the most frequently used algorithm for training neural networks.

In Pytorch, for example, the gradients are computed automatically using the built-in automatic differentiation engine called torch.autograd. The goal of this part of the project is to model and implement a minimal automatic differentiation engine that is similar to/inspired by (not the same!) the one provided by Pytorch.

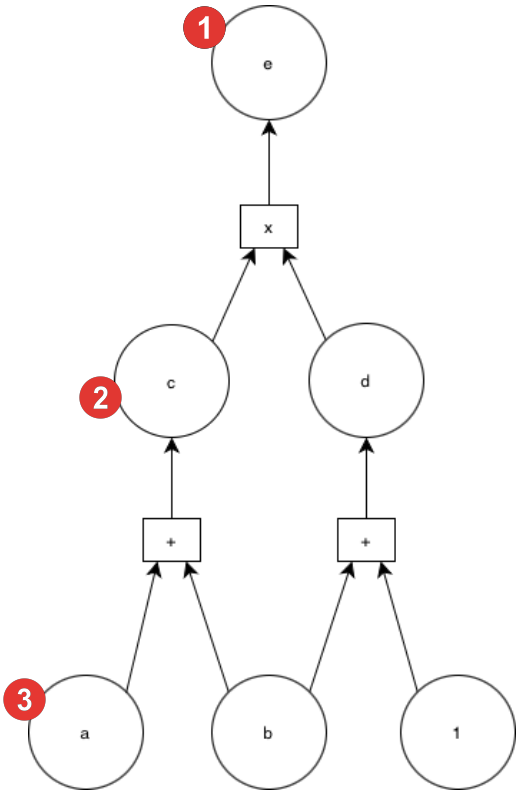
For this part, you can take a look again into the series of YouTube videos Neural networks - YouTube. In particular, the third and fourth videos, which shows how the backpropagation algorithm works in a detailed manner.

The Computation Graph: Forward Pass

To start, let's consider the following simple expression to compute: `e = (a+b)*(b+1)`, with `a` and `b` tensors. Using your library, a user should be able to write this expression as follows:

```
a = torch.tensor(2.0, requires_grad=True)
b = torch.tensor(1.0, requires_grad=True)

e = (a+b)*(b+1)
```

The snippet above should construct a computation graph that looks like the following:



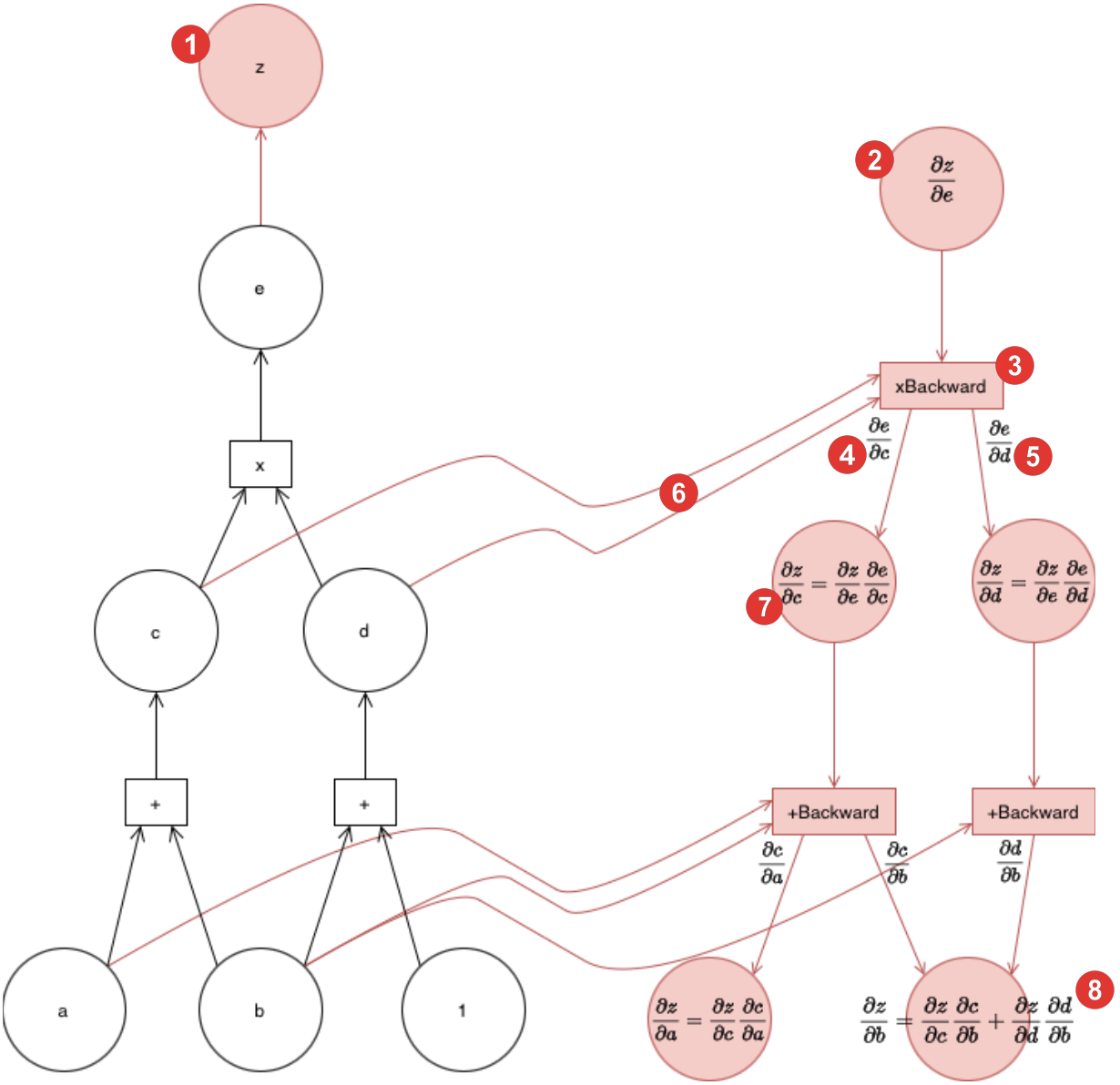
1 e is the output node that holds the end-result of the considered expression. This node is the source (or root) of the computation graph.
2 c is an intermediate node that holds the result of the addition operation between a and b.
3 a (as well as b and 1) are input nodes. This is one of the leaf nodes of the computation graph.

In other words, all operations that are executed have to be traced in a computation graph.

What do we want? Modify the behavior of the primitive operations, like addition, multiplication, division, etc., in a way that one can trace the computation of the forward pass in the form of a computation graph, similar to the one obtained above. You are free on how to construct the computation graph, i.e., what to store and in which form you store it. Keep in mind that it is an OOP project.

GradientEngine: Backward Pass

In addition to tracing the forward pass operations, the computation graph keeps track of the gradient functions corresponding to these operations. These additional nodes will allow performing the backward pass to populate the nodes of the graph that require a gradient. The computation graph above will be extended in following manner:



A dummy node `z` (1) that is equal to `e` (z=Identity(e)) is added to the graph simply to be able to compute the gradient for the node e. This allows the backward process (the chain rule) to start with a value.

(2) is the derivative of the dummy node z with respect to e, noted $\frac{\partial z}{\partial e}$. It is equal to 1. This derivative is the starting point for the backward pass.

(3) is the derivative of the multiplication operation between c and d. Recall that the derivative of the multiplication `e=c*d` is:

- with respect to c : $\frac{\partial e}{\partial c} = d$ (4)
- with respect to d : $\frac{\partial e}{\partial d} = c$ (5)

(6) to compute the actual value of the derivative of the multiplication operation (which are equal to c and d, respectively), we need the actual values of the variables c and d. This is why the nodes corresponding to these variables are linked to the derivative of the multiplication (3)

(7) is the derivative of the output (z) w.r.t. c, i.e., $\frac{\partial z}{\partial c}$. It is computed using the chain rule by multiplying (2) and (4).

(8) is the derivative of the output (z) w.r.t. the input b, i.e., $\frac{\partial z}{\partial c}$. Note that here, node b receives two derivatives ($\frac{\partial c}{\partial b}$ and $\frac{\partial d}{\partial b}$). In this situation, the incoming gradients are added-up together.

Calling the `backward()` method of a given node (or tensor) of the graph should trigger the backpropagation of the gradient starting from the given node back to the leaf nodes of the computation graph. Here is the complete code from the user's perspective, including the forward and backward passes:

```
a = torch.tensor(2.0, requires_grad=True)
b = torch.tensor(1.0, requires_grad=True)

e = (a+b)*(b+1)

e.backward()

print(a)  # tensor(2., requires_grad=True)
print(b)  # tensor(1., requires_grad=True)

print(e)  # tensor(6., grad_fn=<MulBackward0>)

print(a.grad)  # tensor(2.)
print(b.grad)  # tensor(5.)
print(e.grad)  # tensor(1.)
```

The backward pass will be triggered by calling the method `.backward()` on the tensor corresponding to the source node `e of the graph`.
The gradient will be computed from each of the gradient functions traced by the computation graph.
This process accumulates the gradients in the respective node's .grad attribute.
Ultimately, this will result in propagating the computed gradient, using the chain rule, starting from the source node `e` (the output) throughout the graph until the leaf nodes `a` and `b`.

Here is the pseudo-algorithm of the joint forward and backward process in terms of the computational graph.

Let $v_1,\ldots,v_N$:= topological_ordering(graph)

Let v_N be the source (or output) node

for i = 1 to N

compute v_i as a function of $Parents(v_i)$

$$\frac{\partial z}{\partial v_N} = 1$$

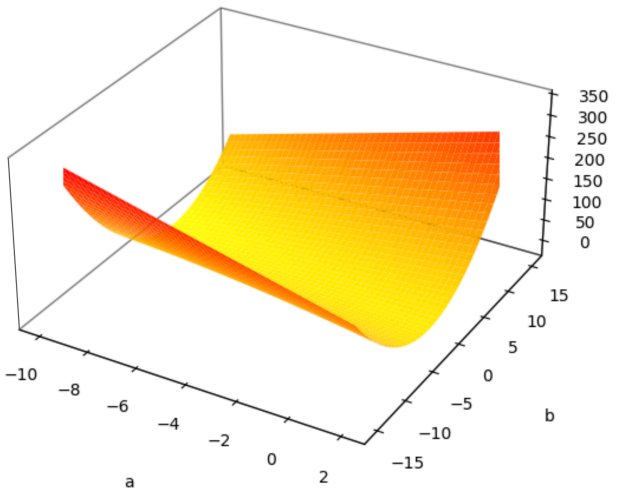
for i = N-1 to 1

$$\frac{\partial z}{\partial v_i} = \sum_{j \in Children(v_i)} \frac{\partial z}{\partial v_j} \frac{\partial v_j}{\partial v_i}$$

Optimization: the Gradient Descent Algorithm

Training a neural network model is an optimization problem where the objective is to find a configuration of the model's parameters (weights and biases) that minimizes the cost function.

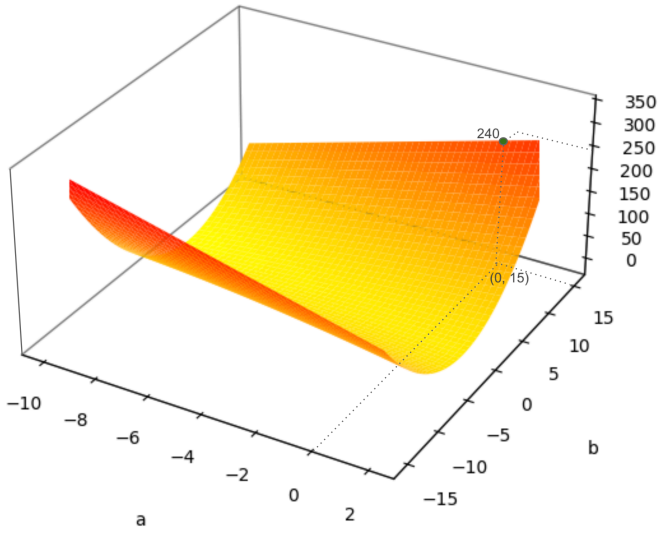
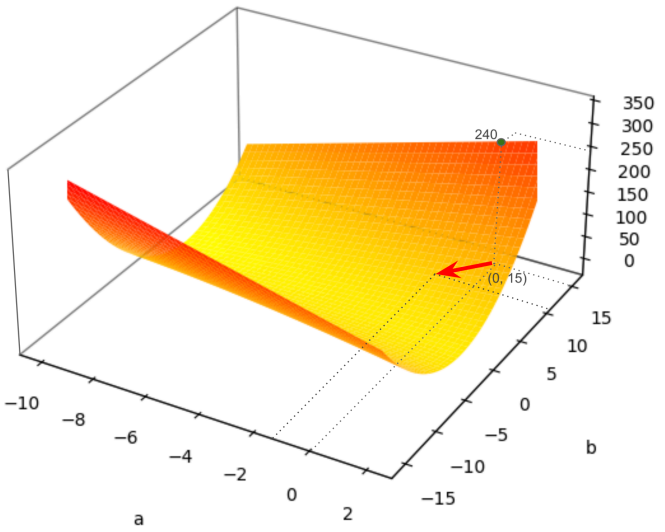
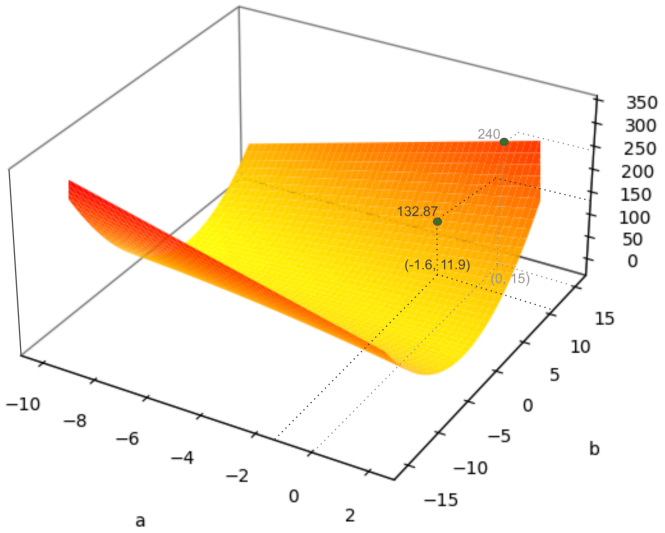
If we consider the previous expression (i.e., `e = (a+b)*(b+1)`) as a cost function to minimize, its corresponding loss (or optimization) landscape will look like the following:



When training a neural network, we usually start from a random initial configuration of the neural network's parameters. This initial configuration corresponds to a point in the above optimization landscape.

Using the gradients computed by the Automatic Differentiation Engine (see dedicated Section), now the goal is to actually move from the (random) initial configuration of the model's parameters towards a configuration that minimizes the cost function. For this, we will use the gradient descent algorithm.

Let's detail further what is happening here:

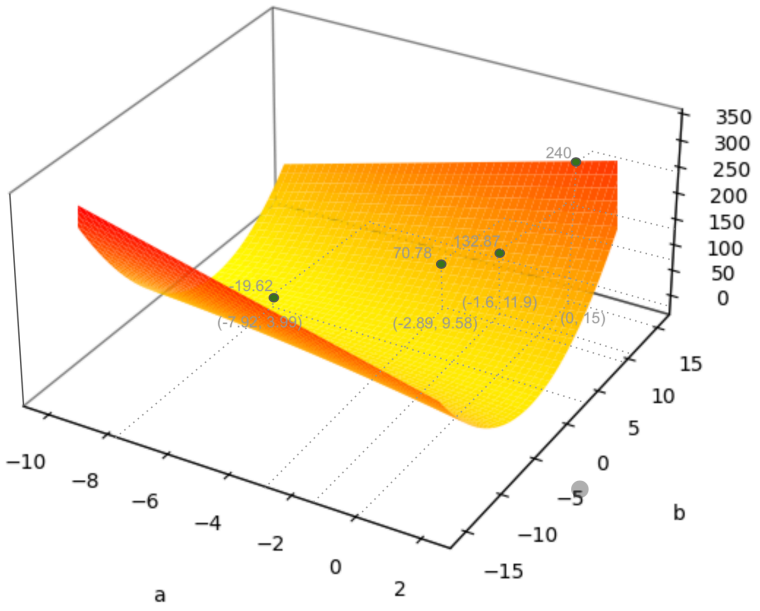
Let's say that the initial configuration of the parameters `a` and `b` were randomly chosen to be `0.` and `15.`, respectively. The value of the expression `e` at this initial configuration is equal to `240.`	
If we compute the gradient at this point (`0.`, `15.`), we will obtain (`16.`, `31.`). The negative of the gradient corresponds to the direction of steepest descent in the loss landscape at the point (`0.`, `15.`). In other words, in order to minimize the expression (i.e., `e = (a+b)*(b+1)`), we have to move the configuration of the parameters (`a` and `b`) following the direction provided by the gradient. The amount of move can be chosen accordingly, but the most important thing is the direction where to move.	
Effectively moving the initial configuration of the parameters (`0.`, `15.`) following the direction provided by the negative gradient will lead us to this new configuration (`-1.6`, `11.9`). This new configuration has a value (`132.87`) lower than that of the initial configuration of the parameters (`240`). We proceed like this until we find a configuration of the parameters that cannot be improved anymore, meaning that we reached a configuration of the parameters that has a gradient equal to zero. Note that, here, we used a learning rate of `0.1`.	

The following code corresponds to applying the gradient descent algorithm to minimize the expression `e`.
We start, as in the description above, with initial values for `a` and `b` equal to `0.` and `15.` respectively. (1)
The gradients of `a` and `b` with respect to the expression `e` are computed by calling `.backward()`. (2)
The values of `a` and `b` are then updated by subtracting their respective gradients (`grad_a` and `grad_b`) weighted-down by the learning rate (3). This process is repeated, in this case, 10 times.

```
val_a, val_b = 0., 15.  (1)
learning_rate = 0.1
#learning_rate = 1  # with lr=1, we end-up diverging

for i in range(10):
    print('################## iteration {} ##################'.format(i))
    print('a ', val_a, ' b ', val_b)
    print('e = ', (val_a+val_b)*(val_b+1))

    a = torch.tensor(val_a, requires_grad=True)
    b = torch.tensor(val_b, requires_grad=True)

    e = (a+b)*(b+1)
    e.backward()  (2)

    grad_a = a.grad
    grad_b = b.grad
    print('grad_a ', grad_a.detach(), ' grad_b ', grad_b.detach())

    # update parameters
    val_a = val_a-learning_rate*grad_a
    val_b = val_b-learning_rate*grad_b  (3)
```

In the following, the details of computation are provided. At each iteration, we print the current values of a and b, the values of their respective gradients, and the value of the expression obtained with the current values of a and b. We can see that the successive values of the expression e are decreasing, meaning that following the negative direction provided by the computed gradients is effectively minimizing the expression e.

```
################## iteration 0 ##################
a  0.0  b  15.0
e =  240.0
grad_a  tensor(16.)  grad_b  tensor(31.)
################## iteration 1 ##################
a  tensor(-1.6000)  b  tensor(11.9000)
e =  tensor(132.8700)
grad_a  tensor(12.9000)  grad_b  tensor(23.2000)
################## iteration 2 ##################
a  tensor(-2.8900)  b  tensor(9.5800)
e =  tensor(70.7802)
grad_a  tensor(10.5800)  grad_b  tensor(17.2700)
################## iteration 3 ##################
a  tensor(-3.9480)  b  tensor(7.8530)
e =  tensor(34.5710)
grad_a  tensor(8.8530)  grad_b  tensor(12.7580)
...
################## iteration 9 ##################
a  tensor(-7.9226)  b  tensor(3.9922)
e =  tensor(-19.6213)
grad_a  tensor(4.9922)  grad_b  tensor(1.0619)
```



Note that if you use a learning rate of 1, the gradient descent algorithm process diverges. In other words, instead of obtaining decreasing values for the expression e, we get increasing values.

Testing on a real-world dataset

Your library should be tested both on simple expressions, like the one considered above, and relatively complex neural network models, with a succession of layers, nonlinear activation functions, and output layers (e.g., LogSoftmax), like the neural network defined in Section NN library from the user's perspective.

Your library should handle model definition, data loading, and training via gradient descent optimization.

The user interface (or API) should be unified and look like Pytorch's interface.

You have to provide various test files in order to test your implementation and showcase what works during the project’s defense.