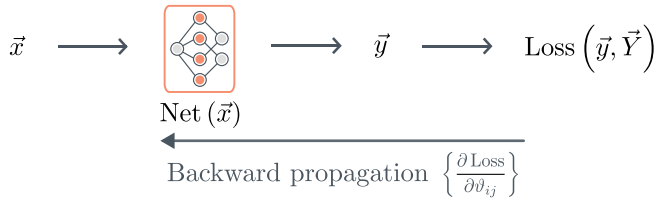
Project #2

Graduating Autograd

Deadline: 02.12.2024, 12:00h

At the heart of every deep learning framework (such as PyTorch) lies an automatic differentiation engine. Typically, an input in a training dataset is propagated through a neural network thereby producing an output and then, using a suitable loss function, compare this to the desired output. The goal is to minimize the loss function. This is done by computing the gradients of the loss function with respect to the network parameters ϑ_{ij} and adjusting them using an optimization algorithm (e.g. gradient descent). Here, the automatic differentiation engine comes into play.



In this project, you will implement your own automatic differentiation engine and learn, how deep learning frameworks function on the way.

Forward propagation vs. backward propagation

The important bit when it comes to automatic differentiation is the backward propagation. Neural networks are in essence just functions $\vec{y} = \text{Net}(\vec{x})$. While in a forward pass, the goal is to simply compute the output $\vec{y}$ of the network, backward propagation is used to find the gradient of the loss function numerically and automatically

$$\frac{\partial \,\text{Loss}(\vec{y})}{\partial \vartheta_{ij}} = \frac{\partial \,\text{Loss}(\vec{y})}{\partial \vec{y}} \frac{\partial \vec{y}}{\partial \vartheta_{ij}} = \frac{\partial \,\text{Loss}(\vec{y})}{\partial \vec{y}} \frac{\partial \,\text{Net}(\vec{x})}{\partial \vartheta_{ij}},$$

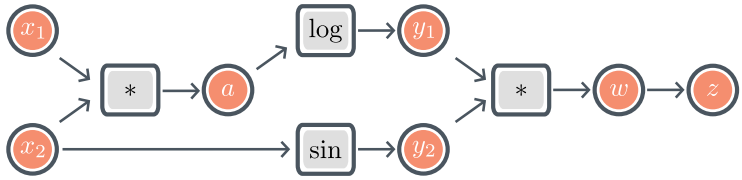
where we used the chain-rule. Things however get complex with the derivative of the network with respect to the parameters. The network contains many small operations involving the various weights in the network. All of these operations add up to the final result. For all of these atomic operations we need to differentiate and apply the chain-rule to obtain $\frac{\partial \,\text{Net}(\vec{x})}{\partial \vartheta_{ij}}$.

Building computation graphs

To be able to compute such complex derivatives, we need to somehow track every operation to the input $\vec{x}$. Later, in the backward pass, we can multiply local derivatives of all these operations, while obeying the chain-rule, to obtain the final result.
The trick here is to continuously build up a computation graph, taking note of all performed operations. Let's step back from neural networks for now – instead of considering deep neural networks, we look simple functions, changing the notation such that from now on x is the variable to which derivatives are evaluated[1]. Suppose, for example, we have a function

$$z := f(x_1, x_2) = \log(x_1 x_2) \sin(x_2).$$

The corresponding computation graph would look something like this:



Notice that two different types of nodes: operational nodes (gray) and value nodes (orange), and that after every atomic operation, an auxiliary value node is defined.

Back propagation

Now how do we obtain the derivatives $\frac{\partial z}{\partial x_i}$ in the example above?[2] Deep learning frameworks, like PyTorch, store lists of atomic derivatives (see here). So for our example, the derivatives of $*$, log and sin are simply known

$$\frac{\partial}{\partial x} a * x = a, \quad \frac{\partial}{\partial x} \log(x) = \frac{1}{x}, \quad \frac{\partial}{\partial x} \sin(x) = -\cos(x).$$

The only thing left to do now is backwards traverse the computation graph, use our atomic derivatives and apply the chain rule to obtain the desired

[1] In ML, x is often denoted the input to the network and ϑ_{ij} the parameters which are changed in the optimization.

[2] Note that we are of course only looking at the numerical derivative. So what we are really computing is e.g. $\frac{\partial z}{\partial x_i}\big|_{x_i=3}$.

derivative, e.g.

$$\begin{aligned}\frac{\partial z}{\partial x_1} &= \frac{\partial z}{\partial w}\frac{\partial w}{\partial y_1}\frac{\partial y_1}{\partial a}\frac{\partial a}{\partial x_1} \\ &= 1 \cdot y_2 \cdot \frac{1}{a} \cdot x_2 \\ &= 1 \cdot \sin(x_2) \cdot \frac{1}{x_1 x_2} \cdot x_2\end{aligned}$$

Very similarly, the other derivatives can be computed. While this algorithm does not look like a great achievement for small functions, it is a necessity for deep neural networks. Be reminded that neural networks are just a very complex function of atomic operations.

How does this look in PyTorch?

All the above happens in PyTorch under the hood. Let's look at the example function f from above and compute its derivatives using PyTorch's autograd.

```
1  import torch
2  import numpy as np
3
4  x1_val, x2_val = 1., 2.
5  x1 = torch.tensor([x1_val], requires_grad=True)
6  x2 = torch.tensor([x2_val], requires_grad=True)
7  a = x1 * x2
8  y1 = torch.log(a)
9  y2 = torch.sin(x2)
10 z = w = y1 * y2
11
12 # Initialize backpropagation
13 z.backward()
14
15 # Computation using calculus
16 dzdx1 = lambda x1, x2: np.sin(x2)/x1
17 dzdx2 = lambda x1, x2: np.cos(x2) * np.log(x1*x2) +
       np.sin(x2)/x2
18
19 print(f"PyTorch: x1.grad = {x1.grad}, x2.grad = {x2.grad}")
20 print(f"Calculus: dz/dx1 = {dzdx1(x1_val, x2_val):.4f},
       dz/dx2 = {dzdx2(x1_val, x2_val):.4f}")
```

Here, we define two PyTorch tensors `x1` and `x2` and perform the operations happening in f on them. The option `requires_grad=True` tells PyTorch to record operations on that particular tensor, i.e. built up a computation graph. In line 13, `z.backward()` starts the backpropagation, propagating

the gradient backwards through the computation graph. Play around with the PyTorch autograd yourself!

What should you do?

The goal of the project is to implement backpropagation to build your own autograd engine. Your final product should function similar to the PyTorch autograd engine, we looked at above. You don't necessarily have to implement tensors; scalar values are enough. The program should be able to compute gradients of functions similar to f. Here is a list of examples that you should try for your testing:

$$\begin{aligned} f(x_1, x_2) &= \log(x_1 x_2)\sin(x_2) \\ g(x_1, x_2) &= x_1 x_2 (x_1 + x_2) \\ h(x) &= 3x^2 + 4x + 2 \\ \text{Neuron}(\vec{x}, \vec{w}, b) &= \tanh\left(\vec{x} \cdot \vec{w} + b\right) \end{aligned}$$

Minimum requirements

Your project should fulfill some minimum requirements:

- Implement an automatic differentiation engine using the backpropagation algorithm from scratch. Do *not* use PyTorch in your implementation.
- Show, that your algorithm works on the example functions above.
- Present your work in a suitable way, explaining what you did.

Optional

- Implement neurons and layers of neurons, viz implement neural networks, taking your autograd engine as a basis.
- Using an optimization algorithm, train a simple neural network. You could e.g. train a neural network to replicate a logical AND gate or do a simple classification task.