Tensor-valued autograd engine

Maksim Sapronov

ČVUT–FIT

sapromak@fit.cvut.cz

January 3, 2024

1 Introduction

One of the most common methods for training neural networks is gradient descent. However, its implementation requires the ability to compute gradients of the model parameters with respect to the loss function. This project is a part of a deep learning library (**kittygrad**) that enables users to address this issue on the fly using dynamic computational graphs and the chain rule.

2 Methods

2.1 General approach

Unlike the classical definition of gradient in vector calculus [5], in the field of deep learning, the term **gradient** is more expansive and extends to the tensor case, not just the vector. However, this does not hinder its calculation using the **chain rule** [4], which in scalar form takes the following expression:

$$\frac{\mathrm{d}f}{\mathrm{d}x} = \frac{\mathrm{d}f}{\mathrm{d}g}\frac{\mathrm{d}g}{\mathrm{d}x}$$

When constructing the decomposition of a function as a graph through individual operators, we can utilize this rule during the backward pass to compute gradients of individual intermediate tensors. These gradients, in turn, iteratively allow us to find the gradient with respect to the arguments or parameters of our function.

2.2 Details

2.2.1 Forward pass

Let's assume that we want to perform an operation on a tensor involving another tensor, a NumPy array, or a scalar. First, we call the frontend function (for example, this could be the dunder method `__mul__`). Next, if necessary, the two operands are cast into tensor form using the `autocast` decorator. If the operation is element-wise, broadcasting is also applied to align shapes. Then, within the frontend function, additional conditions are checked to determine the validity of the operation. If everything is OK, the backend function is called, passing an additional context variable to store intermediate values that may be needed in the backward pass. Subsequently, the necessary computation takes place, and depending on the `requires_grad` attribute of the resulting tensor, a backward node is created. The resulting tensor is returned with a reference to it.

The `autocast` decorator uses the `type` method and the `broadcast_tensors` function, both of which are also differentiable. Therefore, a single operation may generate multiple backward nodes simultaneously. More generally, this doesn't mean that complex operations are implemented using a set of simpler ones. For example, computing the variance does not create a **MeanBackward** node, even though the mean value is computed.

2.2.2 Backward pass

The computation of gradients begins with calling the `backward` method on the final tensor. Instead of using topological sorting and BFS, kittygrad traverses the backward graph using DFS and the concept of locks. The backward node of each computed tensor in the forward pass has a `_lock` attribute, indicating the number of operations in which this tensor participated. Thus, when an update to the gradient arrives at the node of this tensor, it accumulates until the lock value drops to zero. This allows simulating topological sorting without an additional graph traversal and, most importantly, enables using lock values to verify the correctness of the constructed computational graph and the `backward` method call itself. For example, kittygrad will promptly inform you if you initiate the process from the middle of the computational graph or if there are multiple final tensors awaiting the `backward` method call.

Backpropagation ends in the **AccumulateGrad** nodes, which transfer the accumulated gradients to the `grad` attribute of the tensor itself. If the tensor was not initialized by the user but was created during the forward pass, it won't receive its gradient. This can be bypassed by calling `retain_grad` method on it, which will force its backward node to

also perform the **AccumulateGrad** functions.

2.2.3 Tensor version control

Another important aspect worth mentioning is the version control of tensors. In order for inplace operations on tensors to work correctly and without creating unnecessary copies of them, kittygrad assigns a zero version to each tensor, which increments if it is modified in-place (see `inplace` decorator). If such a tensor is retained in the context of a backward node and is needed for gradient computation, its version is compared to the one it had at the time of creating the backward node, and an exception is raised in case of an error.

Some operations do not create full-fledged tensors in computer memory but rather use another **view** on an existing data **storage**. In such cases, the resulting tensor shares its version with its predecessor. This is achieved by using the `share` decorator.

3 Results

The implementation successfully handles all provided tests and edge cases. An example of usage can be found in the corresponding folder or readme file, where a visualization of the computational graph is also provided.

4 Conclusion

In conclusion, it's worth noting that while all the set goals of this project have been achieved, it does not mark the end of the development of the kittygrad library itself. In the future, it will be equipped with everything necessary for simple and elegant neural network training:

- `nn` module,
- Gradient descent optimizers (SGD, Adam, ...),
- Convolution, attention, transformers.

References

[1] Alexey Andreyevich Radul Jeffrey Mark Siskind Atilim Gunes Baydin, Barak A. Pearlmutter. Automatic differentiation in machine learning: a survey. online, 2018. [cit. 2024–01–03] `https://arxiv.org/abs/1502.05767`.

[2] PyTorch Contributors. Pytorch autograd mechanics. online, 2023. [cit. 2024–01–03] `https://pytorch.org/docs/stable/notes/autograd.html`.

[3] PyTorch Contributors. Pytorch documentation. online, 2023. [cit. 2024–01–03] `https://pytorch.org/docs/stable/index.html`.

[4] Wikipedia contributors. Chain rule. online, 2024. [cit. 2024–01–03] `https://en.wikipedia.org/wiki/Chain_rule`.

[5] Wikipedia contributors. Gradient. online, 2024. [cit. 2024–01–03] `https://en.wikipedia.org/wiki/Gradient`.

[6] NumPy Developers. Numpy documentation. online, 2023. [cit. 2024–01–03] `https://numpy.org/doc/stable/`.

[7] Andrej Karpathy. micrograd. online, 2020. [cit. 2024–01–03] `https://github.com/karpathy/micrograd/tree/master`.

[8] Andrej Karpathy. The spelled-out intro to neural networks and backpropagation: building micrograd. online, 2022. [cit. 2024–01–03] `https://www.youtube.com/watch?v=VMj-3S1tku0`.

[9] Elliot Waite. Pytorch autograd explained - in-depth tutorial. online, 2018. [cit. 2024–01–03] `https://www.youtube.com/watch?v=MswxJw-8PvE`.