Building a Neural Network from Scratch

Arturo Cerasi (342010), Francesco Borg (343230), Tomás Feith (342553)
Deep Learning (EE-559), EPFL, Switzerland

I. Introduction

Autograd, introduced under the PyTorch framework by Paszke *et al.* (2017) [1], is the fundamental background of much of the present machine learning research. This automatic differentiation tool makes it possible to design and use networks that are a lot more complex, and do so much faster.

However, as most tools, autograd can be a crutch. Since one doesn't need to know what is happening in the network, and exactly how it works, it is entirely possible to get a network up and running, and even get reasonable results, without understanding its inner workings. But that is not enough.

So, in this project, we develop a simple model without using autograd, i.e. having to code all the methods of the model ourselves.

II. Basic Notions and Components

A. Forward and Backward Passes

Every module from a network has a forward pass and a backward pass. The forward pass is called when the input traverses the network, from the first module to the last, to get a prediction. The backward pass is called after the forward pass, and its purpose is to update the weights using the gradients from the forward pass.

So, considering a module M, with a (possibly empty) set of weights $W_1, ..., W_N$ and an input X, on the forward pass the desired output is

$$Z = M(X; W_1, ..., W_N) \tag{1}$$

The module can be linear or non-linear but, regardless, this is always the fundamental of a forward pass.

On the backward pass the gradients are travelling backwards through the network, from the output back to the input, and so the module M receives the gradient from the layer after it. There are $N+1$ quantities to compute on the backward pass of M. The quantity $\partial L/\partial X$ is the gradient of the loss with respect to the input, and this quantity will be passed on to the next module. Then there are the quantities $\partial L/\partial W_i$, $i = 1, ..., N$ which are used to update the weights.

However, none of these quantities can be computed directly, requiring the usage of the chain rule. The quantities know are $\partial L/\partial Z$, which came from the previous module, and $\partial Z/\partial X$ and $\partial Z/\partial W_i$, which can be compute knowing the characteristics of the module. So, using the chain rule one gets the following equations.

$$\frac{\partial L}{\partial X} = \frac{\partial L}{\partial Z}\frac{\partial Z}{\partial X} \tag{2}$$

$$\frac{\partial L}{\partial W_i} = \frac{\partial L}{\partial Z}\frac{\partial Z}{\partial W_i} \tag{3}$$

B. Layers and Functions Implemented

The network implement in this project will be the following.

```
Conv(stride 2),
ReLU,
Conv(stride 2),
ReLU,
Upsampling(scale factor 2),
ReLU,
Upsampling(scale factor 2),
Sigmoid
```

As such, those modules will be implemented.

C. Convolutional Layer: Extended Discussion

Arguably the most challenging component from the ones present above is the convolutional layer. So here we present a detailed discussion of its forward and backward pass.

1) Notation: Let X be the input tensor, with shape $(B, C_{in}, H_{in}, W_{in})$ and Z the output tensor, with shape $(B, C_{out}, H_{out}, W_{out})$. Furthermore we'll need to consider a kernel W, with shape $(C_{out}, C_{in}, K_0, K_1)$. The layer has the parameters padding (P_0, P_1), stride (S_0, S_1), dilation (D_0, D_1), and kernel size (K_0, K_1). The relationship between H_{out}, W_{out} and H_{in}, W_{in} is given by

$$H_{out} = \left\lfloor \frac{H_{in} + 2P_0 - D_0(K_0 - 1) - 1}{S_0} - 1 \right\rfloor \tag{4}$$

$$W_{out} = \left\lfloor \frac{W_{in} + 2P_1 - D_1(K_1 - 1) - 1}{S_1} - 1 \right\rfloor \tag{5}$$

The tensors X, Z and W are 4-dimensional tensors. So, when referring to a single entry of any such tensor A we'll use A_{ijkl}. For clarity, in PyTorch notation this would match to `A[i,j,k,l]`. Due to a shortage of letters, and to reduce confusing matters by mixing indices, we'll use both greek and latin letters, with no particular meaning to each of them.

2) Forward Pass: To compute the forward pass of the layer we need to find Z. It can be found using the equation below.

$$Z_{\alpha\beta\gamma\delta} = \sum_{i=0}^{C_{in}-1} \sum_{j=0}^{K_0-1} \sum_{l=0}^{K_1-1} W_{\beta ijm} X_{\alpha i r_0 r_1} \tag{6}$$

where r_0, r_1 are the absolute positions inside X, given by

$$r_0 = -P_0 + S_0\gamma + D_0 j \tag{7}$$

$$r_1 = -P_1 + S_1\delta + D_1 m \tag{8}$$

It should be pointed out that, using the equation above naively, it would take 7 nested loops to compute Z. That would be extremely ineffective and instead a vectorized solution should be used.

3) Backward Pass: For the backward pass, the desired quantity is $\partial L/\partial X$, where L is the loss. However, to compute this we need to take the chain of partial derivatives $\partial L/\partial X = \partial L/\partial Z \cdot \partial Z/\partial X$. In the back-propagation stage, the quantity $\partial L/\partial Z$ is given from the previous layer, so we just need to compute $\partial O/\partial X$. From equation 6, we get

$$\frac{\partial Z_{\alpha\beta\gamma\delta}}{\partial X_{ijkl}} = \delta_{i\alpha} W_{\beta jk'l'} \tag{9}$$

where $\delta_{ab} = \mathbb{1}_{\{a=b\}}$ is the Kronecker delta, and $k' = (k + P_0 - S_0\gamma)/D_0$ and $l' = (l + P_1 - S_1\delta)/D_1$.

So, the full expression for $\partial L/\partial X_{ijkl}$ becomes

$$\frac{\partial L}{\partial X_{ijkl}} = \sum_{\beta=0}^{C_{out}-1} \sum_{\gamma=0}^{H_{out}-1} \sum_{\delta=0}^{W_{out}-1} \frac{\partial L}{\partial S_{i\beta\gamma\delta}} W_{\beta jk'l'} \mathbb{1}_{\{k'\in A, l'\in B\}} \tag{10}$$

with $A = \{0, ..., K_0 - 1\}$ and $B = \{0, ..., K_1 - 1\}$.

III. Results

A. Benchmarking

1) Forward Pass Comparisons: To compare our implementations we started by testing their forward passes, and comparing their results with PyTorch's implementations. These were computed for all of the modules implemented, and the results are in table I. The symbol Δ was used to represent the difference between the measurements in PyTorch and ours, i.e. $\Delta Y = Y_{ours} - Y_{Pytorch}$. The tests were performed using one single pass, with a batched, multi-channel tensor. Concretely, the tensor had dimensions `(100, 3, 10, 10)`.

Method	$\overline{\lvert\Delta Z\rvert} \cdot 10^7$
`ReLU`	0.00±0.00
`Sigmoid`	0.00±0.00
`MSE`	0.00±0.00
`Conv2d`	3.77±1.08
`Upsampling`	0.00±0.00

TABLE I: Difference between our implementations and PyTorch, for the forward passes. Values computed as the mean of 10^4 runs, each with a tensor of batch 100. Uncertainty is the standard deviation of the runs.

2) Backward Pass Comparisons: Next we wanted to benchmark our backward passes, comparing our results with their PyTorch equivalent. We considered three quantities $\partial L/\partial X$, $\partial L/\partial W$ and $\partial L/\partial b$. It should be pointed out that the last two are only applicable for the modules `Conv2d` and `Upsampling`, as they are the only ones with learnable parameters, and that none can be computed for the ReLU, as it is non-differentiable in X and has no parameters. The results are in table II.

Like in the previous section, the tests were performed using one single full pass, i.e. forward+backward, of a batched, multi-channel tensor, with dimensions `(100, 100,3, 10, 10)`.

3) Time-Efficiency Comparisons: The computation speed is a very important factor for Deep Learning. As such, the final benchmark performed was for the time per forward pass, and how it compared with PyTorch's version. The results obtained are in table III. These results were obtained using a CPU, so they are not indicative of execution speed in real-world context, but it at least gives a term of comparison between our versions and PyTorch's.

Method	$\overline{\lvert\Delta \frac{\partial L}{\partial X}\rvert}$	$\overline{\lvert\Delta \frac{\partial L}{\partial W}\rvert} \cdot 10^8$	$\overline{\lvert\Delta \frac{\partial L}{\partial b}\rvert} \cdot 10^8$
`Sigmoid`	$(2.49991\pm0.00001)\cdot10^{-1}$	–	–
`MSE`	0.00±0.00	–	–
`Conv2d`	$(7.29\pm2.17)\cdot10^{-10}$	8.62±1.34	1.25±1.11
`Upsampling`	$(6.15\pm1.73)\cdot10^{-12}$	18.95±8.44	0.00±0.00

TABLE II: Difference between our implementations and PyTorch, for the backward passes. Values computed as the mean of 10^4 runs, each with a tensor of batch 100. Uncertainty is the standard deviation of the runs. W and b, when applicable, stand for the module's weight and bias, respectively.

Method	t_{us} (μs)	t_{PT} (μs)
`ReLU`	17.8±3.8	4.5±0.7
`Sigmoid`	35.8±4.8	3.2±0.2
`MSE`	25.8±2.24	44.2±4.6
`Conv2d`	121.8±13.9	48.4±5.2
`Upsampling`	353.9±49.0	73.3±5.3

TABLE III: Execution times for our modules, t_{us}, and PyTorch's, t_{PT}. Values computed on CPU, as the mean of 1000 runs, each with 1000 calls. Uncertainty is the standard deviation of the runs.

B. Simplified Network: Results

Using the network described in section II-B, it was trained with the provided training data, and its performance was measured. However, we weren't able to make the network converge adequately. Several parameters of kernel size, padding, in channels and out channels were tested, but for all the trails performed the network would always start to output a zero matrix after some iterations.

Given the results in the previous section we are confident that each individual implementation is working adequately. So either we weren't able to select appropriate parameters for the network, or the modules aren't working correctly together. Further research is needed to pinpoint exactly what isn't working.

IV. Discussion

As can be seen in section III-A, the individual results for each of the modules implemented are satisfactory.

For the forward pass the differences between ours and PyTorch's version are all zero, except for the `Conv2d` layer, which shows a difference of $(3.77\pm1.08)\times10^{-7}$, and we believe this difference can be attributed to round-up errors.

For the backward pass we observe extremely low values error in all the metrics considered, except for $\partial L/\partial X$ for the `Sigmoid` layer, which shows a difference of $(2.49991\pm0.00001)\times10^{-1}$. This difference might be due to a difference in implementation. The sigmoid function is given by $\sigma(x) = 1/(1+e^{-x}) = e^x/(1+e^x)$. These two equivalent formulations can be used, and it seems PyTorch uses the former for positive values of x, and the latter for negative. So this may be the cause for the differences observed in the backward pass. Further research into this is necessary.

Finally, for the execution time we see that our implementations are, overall, slower than PyTorch's. But this is not unexpected, and the results achieved seem acceptable.

References

[1] A. Paszke, S. Gross, S. Chintala, G. Chanan, E. Yang, Z. DeVito, Z. Lin, A. Desmaison, L. Antiga, and A. Lerer, "Automatic differentiation in pytorch," 2017.