Mini Project 2

Giovanni La Cagnina
École Polytechnique
Fédérale de Lausanne
Lausanne, Switzerland
Email: giovanni.lacagnina@epfl.ch
SCIPER: 342352

Pau Autrand Caballero
École Polytechnique
Fédérale de Lausanne
Lausanne, Switzerland
Email: pau.autrandicaballero@epfl.ch
SCIPER: 343812

Joshua Voelkel
École Polytechnique
Fédérale de Lausanne
Lausanne, Switzerland
Email: joshua.voelkel@epfl.ch
SCIPER: 343212

I. Introduction

In the upcoming subsections, we will describe both our quantitative and qualitative approach to build each of the given blocks. For the following notation, we denote by x the input tensor received from the previous layer, and by y the output of the given layer.

II. Convolution Layer

A. Forward pass

As proposed in the exercise sheet, we implemented the forward pass of the convolution using `unfold` and `fold`. The former extracts the patches, which arise by sliding the kernel, κ, over the matrix, into columns. Then, we used `reshape` to bring the κ into a row vector. This allows us to perform a convolution using linear matrix multiplication. After multiplying the reshaped κ with the unfolded input matrix, we used the command `fold` to obtain the final desired shape of the output matrix. In this last step, we had to determine the desired output height and width, which is a result of input height(h)/width(w), stride(s), padding(p), dilation(d) and κ size (k). To be more precise the output height and width are given by

$$h_{out} = *\frac{h_{in} + 2p[0] - d[0](k[0]-1) - 1}{s[0]} + 1,$$
$$w_{out} = *\frac{w_{in} + 2p[1] - d[1](k[1]-1) - 1}{s[1]} + 1,$$

where the indexing $[0]$ and $[1]$ denote the first and second parameters respectively [3]. The weights and the bias tensors of the convolution layer are initialized randomly using a Uniform distribution $\mathcal{U}(-\sqrt{q}, \sqrt{q})$, where $q = 1/(c_{in}k[0]k[1])$.

B. Backward pass

The backward pass of the convolution layer requires a bit more complex mathematical derivation.

Gradient w.r.t kernel: We can firstly note that during the forward pass κ has been multiplied with the unfolded input tensor, therefore the gradient w.r.t. κ is simply the gradient w.r.t.the output multiplied by the unfolded transpose input tensor. This obtained tensor then is reshaped to match the shape of κ. This can be seen as a convolution forward pass.

Gradient w.r.t bias: The gradient with respect with the bias is simply the gradient w.r.t. the output summed over the batch, height and width dimensions.

Gradient w.r.t. input: As we have seen in class (lecture 7.1), the gradient w.r.t. the input can be seen as a transposed convolution between the derivative w.r.t. the output and κ. As it is shown in [2], a transposed convolution is the same as a standard convolution between the derivative w.r.t. output and the flipped κ. Mathematically, this can be summarized as

$$\begin{aligned}\frac{\partial \ell}{\partial x} &= \frac{\partial \ell}{\partial y} * \kappa \\ &= \frac{\partial \ell}{\partial x} \circledast \tilde{\kappa},\end{aligned}$$

where $*$ denotes a transposed convolution, $\circledast$ denotes a standard convolution and $\tilde{\kappa}$ represents the flipped kernel κ. We implemented this by flipping the κ and conducting a standard convolution forward pass as described before.

III. Upsampling Layer

A. Forward pass

For the upsampling layer we decided to implement a combination of Nearest Neighbor Upsampling (NNU) + Convolution. The reason for this was ,on one hand, we already implemented the convolution and thus only needed to implement the NNU. On the other hand, NNU + Convolution can be favored in comparison to Transopsed Convolution as the latter can cause checkerboard artifacts due to uneven overlaps [1]. For the NNU, we wrote a function which takes a tensor as input and scales both the height and width according to a predefined scale factor. The output matrix then contains at each position the figure which is the closest to out of the input matrix. Then, we apply a standard convolution forward pass to this output matrix using the identical steps as described in the previous subsection.

B. Backward pass

In the forward pass, we applied the NNU and the convolution sequentially. In the backward pass, we therefore have to first call the backward pass of the convolution (described in the previous subsection), where the gradients w.r.t. weights and bias are calculated. For the gradient w.r.t. the input, we receive the gradient from the backward pass of the convolution layer.

This clearly has the wrong dimension compared to the input of the NNU, as the latter increased the size of the input in the forward pass. It can be shown that in the backward pass of the NNU, it is necessary to sum over the gradients of the duplicated elements. Like this, we also get the right input dimension, as by summing over the gradients, we do some kind of downsampling, shrinking the tensor to get the desired shape.

IV. ReLU

The implementation of the ReLU activation function was relatively straightforward. The forward pass of the ReLU is simply the definition and is given by

$$\text{ReLU}(x) = \max(0, x).$$

The backward pass of the ReLU outputs the derivative w.r.t. the ouput of the layer multiplied by the first derivative of the ReLU w.r.t. the input function, which is given by

$$\frac{\partial \text{ReLU}}{\partial x}(x) = \begin{cases} 0 & x < 0 \\ 1 & x > 0 \end{cases}$$

and is technically undefined for $x = 0$. However, we defined it to be zero for simplifying reasons.

V. Sigmoid

Also the Sigmoid activation function was relatively straightforward to implement. Analogue to the ReLU activation function, the foward pass of Sigmoid is given by its defintion and reads

$$\text{Sigmoid}(x) = \frac{1}{1 + e^{-x}}.$$

Also analogue to the ReLU activation function, we implemented the backward pass by multiplying the derivative w.r.t. the output of the layer with the derivative of the sigmoid function w.r.t. the input tensor x. This derivative is given by

$$\frac{\partial \text{Sigmoid}}{\partial x}(x) = \frac{e^{-x}}{(e^{-x} + 1)^2}.$$

VI. Sequential

This class provides a sequential container. In other words, the value of a Sequential provides manually calling a sequence of modules that allows treating the whole container as a single module.

The forward pass for the sequential simply iterates over all the modules that are to the class. The forward takes one tensor as input, which is fed to the first of the models. Then, the output is fed into the next model and this process is repeated until the last module. Finally the forward returns the output of the last model.

The background proceeds in the same way but with the gradients and in reversed order. Additionally, the sequential class saves in the same list all the parameters of all the other modules, which can be retrieved with the *param* function.

VII. Mean Squared Error

The mean squared error of the input tensor x given the target tensor t is defined as

$$\text{MSE}(x, t) = \frac{1}{n} \sum_{i=1}^{n} (x_i - t_i)^2,$$

where we take the mean of the squared deviations between the output tensor and the target tensor. We implemented this in the forward pass. The backward pass outputs simply the derivative w.r.t. to x and is given by

$$\frac{\partial \text{MSE}}{\partial x}(x) = \frac{2}{n}(x - t).$$

VIII. Stochastic Gradient Descent

After each batch is processed, i.e. performed a forward and backward pass, the Stochastic Gradient Descent (SGD) module is called to update the weights using the previous weights, the accumulated (over the batch) gradient of the loss function and the learning rate η. To be more precise, this update step reads as follows

$$w_{t+1} = w_t - \eta \sum_{b=1}^{B} \nabla \ell_{n(t,b)}(w_t),$$

where $n(t, b)$ denotes the order of how the samples are visited. In our case, we chose $n(t, b)$ to be sequential as we do not have any specific order in our data set. However, we shuffle our train data sets at each epoch.

IX. Training

The training in the second Miniproject is considerably different than the one in the first Miniproject. Our convolution model achieves very good performance, and also the same results as the `nn.Conv2d` forward pass from the Pytorch framework. However, the Upsampling layer doesn't seem to get as good results as we expected. Furthermore, the MSE, SGD, RELU and Sigmoid functions work fine and smoothly.

We have chosen a kernel size of (2,2), and we go from 3 channels to 128 in the first convolution, and from 128 to 256 in the second one. Moreover, in this one we have substantially increased the learning rate to 0.5, since it has proven to be much more effective and consistent. It is important to notice that the training speed is very fast compared to the models used in the Miniproject 1, which was expected since the model we are using is much more simple. All the functions were adapted so that the code could run on GPU. With that, one epoch on the whole train dataset is only about 20 seconds.

One difference with the Pytorch implementation is that we need to call the backward of both the MSE and the Sequential. That is because the backward pass is not applied directly on the tensors but on the different classes. Also the loading of the model had to be done layer by layer, as opposed to the pytorch implementation which allows it to be saves as a whole dictionary.

REFERENCES

[1] Chris Olag Augustus Odena, Vincent Duomlin. Deconvolution and checkerboard artifacts. https://distill.pub/2016/deconv-checkerboard/, 2016. Accessed: 2022-05-25.

[2] Saniya Maheshwari. Backpropagation in convolutional networks. 2021.

[3] PyTorch. Conv2d. https://pytorch.org/docs/stable/generated/torch.nn.Conv2d.html, 2022. Accessed: 2022-05-25.