Deep Learning Project Part 2 — Pytorch Custom Implementation

Nicolas Wagner, Hugo Lanfranchi, Rémi Delacourt
School of Computer and Communication Sciences, EPFL, Switzerland

I. Introduction

The goal of the second part of the project is similar to the first part, but without using the *torch.nn* modules and the *autograd* property from the Pytorch library. This requires to implement the forward and backward pass manually.

The only functions from *torch.nn* allowed that we used are *torch.nn.functional.unfold* and *torch.nn.functional.fold*. Some helper functions which were allowed that we used are *torch.empty* and torch.cat, and all direct operations on Pytorch tensors were allowed as well.

We implemented useful modules (presented below) which helped us build the required network, following the way their Pytorch counterparts worked. We then compared the performance of the required network using our implementation with the performance of the network using the Pytorch library (will full access to the modules).

The training and test set have been presented in the report of part 1, and we work on the same dataset.

II. Custom implementation of the modules

A. Conv2d

This module applies a 2D convolution, just as in the Pytorch module nn.Conv2d. It is the only module that we implemented that has parameters which are the convolution kernels.

The forward pass makes use of the *torch.nn.functional.unfold* function, which, given a 4D tensor, divides it into patches of the same size as the kernel size, and each patch represents a scan of the kernel on the input. That way applying the kernel on a patch amounts to performing a dot product between the patch in 1d vector form and the kernel in 1d vector form. Thus, applying the kernel on the tensor is performing a matrix multiplication between the unfolded tensor and the (properly reshaped) kernel. This method preserves the speed guarantees of tensorized operations, which will be crucial when training a model that uses this module.

The backward pass relies on two formulas for computing the gradient of the loss $\mathcal{L}$ with respect to matrix multiplication. For $C = AB$:

$$\frac{\partial \mathcal{L}}{\partial A} = \frac{\partial \mathcal{L}}{\partial C} B^T \quad \text{and} \quad \frac{\partial \mathcal{L}}{\partial B} = A^T \frac{\partial \mathcal{L}}{\partial C}$$

Since in the forward pass we perform matrix multiplication $S = Xw$ where X is the unfolded (into patches) input tensor, w is the kernel and S is the convolved output, using the formulas above we compute $\partial \mathcal{L}/\partial X$ (which is what the backward pass is supposed to output) and $\partial \mathcal{L}/\partial w$. Now $\partial \mathcal{L}/\partial X$ is still unfolded into patches, so we make use of *torch.nn.functional.fold* (unfold's counterpart) to fold back into a 4d tensor of same size as the input size in the forward pass. Note that fold is not the inverse of unfold, but it is the inverse of unfold wrt tensor shapes. Note also that, in order to perform $\partial \mathcal{L}/\partial w$, we need X unfolded, so in the forward pass we make sure to save X unfolded for the use in the backward pass.

Finally, concerning parameter initialization, we implemented the He initialization [1]. Parameter initialization can be crucial to the performance of the training, as it can influence the time to find a satisfiable local minima when optimizing the loss, and the time to converge to a local minima. Consider an initialization where the weights of a fully connected layer have a high value. When we apply data to this layer, the output will have very high values because the layer performs scalar products. Then when applying the sigmoid activation function to the output the values will be very close to 1 where the slope of the sigmoid is very small, so gradient descent will progress slowly. This phenomenon happens also for very small values of the parameters, and we will be on the other side of the sigmoid (close to 0). He initialization provides an efficient initialization to circumvent these problems. Each parameter w_i is taken from a normal distribution with 0 mean and small variance relative to the number of parameters there are:

$$w_i \sim \mathcal{N}(0,\, \frac{2}{N})$$

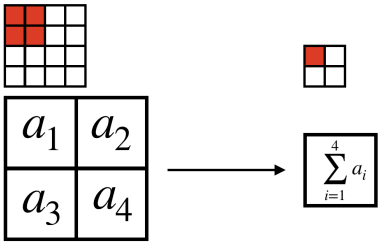
where $N = kernel\_size * out\_channels$.

B. Nearest Neighbor Upsampling

This module rescales the input tensor by a given integer scale factor. The rescaling rule is the nearst neighbor one, meaning that for a given pixel in the input tensor, the neighbors of the same pixel in the output tensor will be assigned the same value.

The forward pass was made easy by the tensor operation $repeat\_interleave()$ which repeats the value of the tensor along a dimension.

The backward pass of this module is more complex. We have the gradient with respect to (wrt) the output and we want to compute the gradient wrt the input, so we have to do the inverse of the upsampling (we want to downsample by the given $scale\_factor$). For a given pixel in the gradient wrt input, its value is the sum of its neighboring pixels.



So basically the backward pass is a convolution on each channel with kernel size of $scale\_factor$ and stride equal to $scale\_factor$. In order to do this, we reused *torch.nn.functional.unfold*, which divides our tensor into patches of size $scale\_factor$, and we just apply a sum on these patches to perform what we described above. Finally we resize the given output tensor to obtain our gradient wrt input.

Note that we use a combination of Nearest Neighbor Upsampling + Conv2d to implement the TransposeConv2d layer.

C. ReLU Activation

$$ReLU(x) = \begin{cases} x, & \text{if } x \geq 0 \\ 0, & \text{otherwise} \end{cases}$$

Concerning it's derivative, as it is not differentiable in 0, we use a subgradient of the function in 0, and we chose 0 so as to simplify the computations.

The forward pass is applying the function to input, and the backward pass is multiplying the gradient wrt output to derivative of forward pass.

D. Sigmoid Activation

$$\sigma(x) = \frac{1}{1+e^{-x}}$$

$$\sigma'(x) = \sigma(x)(1-\sigma(x))$$

Similarly as for ReLU, the forward pass is applying the function to input, and the backward pass is multiplying the gradient wrt output to derivative of forward pass.

E. Sequential Module

The Sequential module is a container of sequential layers of the neural network. It concatenates all the layers such that the layers are connected one to the following so as to have a proper flow of information representing our model.

For the forward pass, it propagates the forward pass of one module as input to the next module, as if the modules were a single module performing a single operation.

The backward pass is similar to the forward pass, and connects each output of a layer as input to the previous layer.

F. Mean Squared Error loss function (MSELoss)

We decided to use the Mean Square Error Loss (MSE) in our implementation. The formula associated with this loss is:

$$\frac{1}{N}\sum_{(i=1)}^{N}(x_i - y_i)^2$$

with x_i being the data and y_i being the target value.

When doing the backward pass we should return the gradient of the loss wrt the final output of the model, which is given by the following formula:

$$\frac{\partial MSE}{\partial x} = \frac{1}{N}\sum_{(i=1)}^{N} 2 * (x_i - y_i)$$

where N is the number of values in x_i.

G. SGD Optimizer

The first optimizer that we implemented is the Stochastic Gradient Descent (SGD). It is a faster alternative to the classical Gradient Descent algorithm, and updates the weights by computing the gradient over batches of samples:

$$w_t = w_{t-1} - \mu \nabla \mathcal{L}_{\mathcal{B}}(w_{t-1})$$

where μ is the step size (learning rate), w_t the weights at time t and $L_{\mathcal{B}}(w_t)$ the derivative of the (batch $\mathcal{B}$) loss with respect to w_t. SGD computes the gradient based on a batch of samples resulting in a faster computation, without compromising the true loss in expectation because it is an unbiased estimator of the true loss. Using batches of samples is a tradeoff between high variance of the loss (small batch size) and slow computation time (large batch size).

H. Adam Optimizer

The second optimizer that we decided to implement is the Adaptive Moment estimator (Adam) optimizer. In this algorithm we are running averages of both the gradients and the second moments of the gradients.This algorithm is used to accelerate the SGD algorithm by taking into consideration the exponentially weighted average of the gradients. Given weights w_t, the loss function $\mathcal{L}_{\mathcal{B}}$, hyperparameters γ (learning rate) β_1, β_2 (acceleration rates), Adam's parameter update is given by: This enables accelerating the convergence to a local minima and reduces the number of epochs needed to obtain such a minima.

Algorithm 1 Adam optimizer

Input: $\gamma, \beta_1, \beta_2, w_0, \mathcal{L}_{\mathcal{B}}, m_0 = 0, v_0 = 0$

for t=1... **do**

$m_t \leftarrow \beta_1 m_{t-1} + (1-\beta_1)\nabla \mathcal{L}_{\mathcal{B}}(w_{t-1})$

$v_t \leftarrow \beta_2 v_{t-1} + (1-\beta_2)\nabla \mathcal{L}_{\mathcal{B}}(w_{t-1})^2$

$\hat{m}_t \leftarrow m_t/(1-\beta_1^t)$

$\hat{v}_t \leftarrow v_t/(1-\beta_2^t)$

$w_t \leftarrow w_{t-1} - \gamma \hat{m}_t/(\sqrt{\hat{v}_t} + \epsilon)$

III. Training & Results

A. Training

The framework of the training follows the one of the Pytorch library. To predict, the $forward()$ method of the model is called on the input, and the predictions are then compared with the groundtruths via the loss. The backward is similar, but we give the derivative of the loss as input to the $backward()$ of the model, which is slightly different from the Pytorch library. This is because $autograd$ is set to false so the tensors do not carry their gradients with them.

The $optimizer.step()$ updates the parameters appropriately in-place, which were references to the optimizer at its creation.

The overall training framework is kept identical with the Pytorch library framework.

B. quantitative results

We present here some quantitative results we have obtained during our implementation to search for the best parameters for our model.

Let's compare the effects of different weights initialization, this was a paramount step in our implementation as when we started initializing the weights with a proper method and value the obtained PSNR (dB) started to be significantly good. Here the best initialization method we used is the He initialization described in the Conv2d section.

Init method	Batch size	LR	PSNR
Filled with 1.0	10	1 * 10^-3	4.52
Filled with $1 * 10^-3$	10	1 * 10^-3	23.24
HE initialization	10	$1 * 10^-3$	24.40

Table 3 : PSNR score on validation dataset with different initialization methods

This validates the theory explained in the Conv2d part about weight initialization, where weights too big or too small imply not a really good training.

Let's compare the effects of our different optimizers. We observe bellow the different results with different batch sizes and learning rates, with a total number of epochs of 5.

Batch size	LR	SGD (dB)	Adam (dB)
10	1 * 10^-2	21.35	22.78
10	1 * 10^-3	19.34	24.40
10	1 * 10^-4	16.94	24.39
100	1 * 10^-2	19.31	23.52
100	1 * 10^-3	16.15	23.85
100	1 * 10^-4	12.58	23.17

Table 3 : PSNR score on the validation dataset using either SGD or Adam

We observe a significant increase in the PSNR score when using the Adam optimizer instead of the SGD optimizer no matter what parameters we give the model. This is because Adam takes advantage of the accelerated gradient decent.

C. validity of our implementation

To be able to compare our implementation with the Pytorch equivalent, we implemented the provided architecture with the $torch.nn$ modules and compared the performance with our hand-crafted modules. We compared the different performances on the PSNR score on the validation dataset with different batch sizes and learning rates, with a total number of epochs of 5 and the Adam optimizer.

Batch size	LR	Pytorch	In-House
10	1 * 10^-2	22.85	22.78
10	1 * 10^-3	24.41	24.40
10	1 * 10^-4	23.51	24.39
100	1 * 10^-2	23.69	23.52
100	1 * 10^-3	23.53	23.85
100	1 * 10^-4	22.82	23.17

Table 3 : PSNR score on validation dataset using either the pytorch modules or our implementation

We observe that we have similar results as the output as the modules implemented with pytorch, meaning that our implementation performs and works as one would expect using the regular pytorch package.

References

[1] S. K. Kumar, "On weight initialization in deep neural networks," 2017. [Online]. Available: https://arxiv.org/abs/1704.08863