Auto-Encoder Building Blocks From Scratch

Luca Zampierin (343174), Saverio Nasturzio (299307), Zhecho Mitev (323387)

[EE-559] Deep Learning, EPFL, Switzerland

I. Introduction

The current project attempts to implement the building blocks of a Convolutional Neural Network [1] from scratch, without using Pyorch specialised modules, such as *torch.nn* or *torch.autograd*. The only exception is the *torch.nn.functional.unfold* method, which supports the implementation of our core modules - **Conv2d** and **NearestUpsampling**. Regarding the architecture we choose to create an abstract class **Module**, which will be inherited by all of the implemented modules. Apart from the suggested functions *forward()*, *backward()* and *param()* we include a *zero_grad()* function for Conv2d and NearestUpsampling to be able to set the gradients of all optimized tensors to zero. Due to the fact all activation functions share the same gradient structure, we create the base **Activation** class, which is inherited by each of the implemented activation functions.

Importantly, given an arbitrary input vector, we aim to produce the same output as the PyTorch implementations of the respective modules. Even though achieving the speed of the PyTorch library is very far from easy, we attempt to make the execution as fast as possible in order have a network which is trainable in a reasonable time.

II. Models and Methods

To achieve our goal we have implemented a forward and a backward method of the following classes:

- **Conv2d** - Perform a convolution operation on an input $(N_{in}, C_{in}, H_{in}, W_{in})$ for given number of output channels C_{out}, stride s, padding p, and kernel size k [1]. The output dimensions $(N_{out}, C_{out}, H_{out}, W_{out})$ are calculated as $H_{out} = \frac{H_{in}-k+2p}{s} - 1, W_{out} = \frac{H_{in}-k+2p}{s} - 1$. Dilation is assumed to be equal to 1.
- **TransposeConv2d** Theoretically the transpose convolution operation can be seen as deconvolution, thus the forward method of Conv2d is similar to the backward method of TransposeConv2d and vice versa [2]. However, instead of implementing this module we choose to implement the nearest neighbor upsampling technique.
- **NearestUpsampling** - Applies a 2D nearest neighbor upsampling to an input. Given a positive scale factor s, it reshapes the input images into the correct dimensions. We define $H_{up} = floor(s \times H), W_{up} = floor(s \times W))$, where (H, W) is the size of a single image before upsampling, while (H_{up}, W_{up}) is the size after upsampling. Since s might be a non-integer value, we round the produced dimensions down to the closest integer. We support only 4-dimensional input for this module. To implement the backward method we apply the unfold function on the gradients with respect to input, with parameters $kernel = (s, s)$ and $stride = s$ and then sum across patches of size (s, s)
- **ReLU** - Applies the rectified linear unit function to an input x. Given by the formula: $ReLU(x) = max(0, x)$, the derivative of this function with respect to the input consists of 1s where the input is positive and 0s where the input is non-positive.
- **Sigmoid** - Applies the logistic sigmoid function σ to an input x: $\sigma(x) = \frac{1}{1+e^{-x}}$. We use it as an activation function at the end of the neural network as it produces an output between 0 and 1. The derivative is defined as follows: $\sigma(x)(1 - \sigma(x))$.
- **MSE** - Calculates the Mean Squared error of an input x compared to a target y. The forward method computes $\frac{1}{n}\sum_{i=1}^{n}(x_i - y_i)^2$, while the backward method is defined as: $\frac{-2(y-x)}{n}$
- **Sequential** - A container which can gather various modules and concatenate their layers in order to create a connected neural network. Given an input it sequentially runs the forward methods of the given modules and eventually returns the output of the last module.
- **Stochastic Gradient Descent (SGD)** - We implement as optimizer the canonical stochastic gradient descent, which updates each parameter of the network by subtracting the product of the learning rate with the gradient of the loss with respect to the module's parameter. Additionally, to improve the training convergence time and make the final network more trainable we implement an optional Momentum [3] by also subtracting a fraction of the previous iteration's decrease factor. At each step, the update works as follows: $w_t = w_t - factor_t$ with the decrease factor defined as: $factor_t = \gamma factor_{t-1} + \eta \nabla_\theta J(\theta)$

III. Implementation

A. Activations

Given the similarity in behavior of all activations, and in particular the identical formulation for the backward pass, we have decided to implement a custom class `Activation` that inherits from `Module` and returns the result of the activation

[1] Note that padding of 'same' is computed before applying any stride, and represented the padding that would keep the activation size unchanged before any stride is applied.

function in the forward pass ($\sigma(x)$), while returning $\frac{\partial L}{\partial Y} * \sigma'(x)$ in the backward pass.

B. Sequential

The Sequential module allows to construct simple networks made of sequences of custom blocks. For this purpose, for the forward pass the output of a module is piped as input to the subsequent one, while for the backward the same process is applied in reverse order since the gradient calculations are carried out from output to input. Furthermore, it exposes within the `self.params` list the parameters of the whole network to be provided to an optimizer.

C. Mean Square Error (MSE)

The implementation of the MSE follows closely the implementation of any other module. It defines as forward operation the computation of the mean of the square of differences between elements of input and target. The backward pass has been implemented by following the mathematical definition of derivative of the output with respect to the input as $\frac{-2(y-x)}{n}$

D. Upsampling

We have decided to implement as Upsampling block the concatenation of a Nearest Neighbor upsampling and a normal convolution that smooths out the replicated patches of received by the NearestNeighbor block.

1) Nearest Neighbor: In order to efficiently compute the upsampled version of the input tensor, the first approach of using nested for loops revealed itself to be unfeasible due to the lack of performance of python loops. We can instead make good use of the C primitives underneath PyTorch's tensor operations by repeat-interleaving first on the y-axis[2] and then on the x-axis. The `repeat_interleave` method efficiently applies such operation on batches of N samples over C channels without additional hurdles. The backward pass is implemented also by employing the `unfold` operation to extract patches of size `factor_size` and returning the sum. This is because the upsampled tensor as output to the forward method will impact the error equally within the same patch. We can thus add up the gradient contributions to compute the impact that one original pixel has on the loss.

E. Convolutional Layer

Our Conv2d module tries to imitate the behavior provided by PyTorch's framework by allowing the instantiation of a convolutional layer made of an arbitrary number of input channels and output channels. Furthermore, we support the application of arbitrary kernel-sizes[3], application of padding, and strides. The only missing feature with respect to the most common use-cases of PyTorch's Conv2d is the dilation factor.

All parameters are initialized following a similar approach, that is, according to a uniform distribution in the range $[-\frac{1}{\sqrt{(channels_{in} * kernel\_size^2}}, \frac{1}{\sqrt{(channels_{in} * kernel\_size^2}}]$ for the biases, and a *Kaiming initialization*[4] for the weights.

[2]with respect to the image plane

[3]only square kernels at the moment.

The core operation employed for the forward and backward passes is the cross-correlation[4] between two batch-like tensors. It has been implemented as a matrix multiplication between the second input operand (kernel) and the unfolded version of the first, followed by a reshaping to reconstruct the desired structure. As profiled in the next section, this batched matrix multiplication is the main bottleneck to the framework' efficiency due to the extensive use for each call and the heavy computations involved in performing multiplications between multi-channel, multi-sample, high-dimensional tensors.

1) Forward Pass: By definition of the convolutional layer, the forward pass consists in a cross-correlation of the input tensor with the weight tensor. For this, we employ the aforementioned cross-correlation function and make sure to process all samples and channels altogether, since the first loop-based implementation was orders of magnitude slower.

2) Backward Pass: The backward pass is slightly more complex if we need to handle non-standard convolutions with strides, padding, or dilations. In particular, without treating it as a pure linear layer, we need to derive the mathematical formulas that describe the gradient of the loss with respect to both the internal weights and the input. By following the unrolling written in [5], the mathematical structure becomes:

$$\frac{\partial L}{\partial X} = zero\_pad(zero\_interleave(\frac{\partial L}{\partial Y})) * rot\_180(W) \quad (1)$$

$$\frac{\partial L}{\partial W} = zero\_pad(X) * zero\_interweave(\frac{\partial L}{\partial Y}) \quad (2)$$

$$\frac{\partial L}{\partial B} = \frac{\partial L}{\partial Y} \quad (3)$$

The `zero_pad` function pads the input with a given number of rows and columns of zeros depending on the padding setting of the layer. The `zero_interweave` function, in our code named `dilate_efficient`, interleaves each column and row with an arbitrary number of zeros, and is similar to a dilation of the input followed by a concatenation of k columns and rows of zero at the end.

The dilate function seems irrelevant, but was also a core bottleneck at the beginning due to a poor loop-based implementation, because of its widespread use in backward passes. After rewriting the function without loops, but by applying the transformation by means of matrix multiplication after constructing special matrices that depend on the dilation factor, it now has negligible impact on computation performance.

F. Stochastic Gradient Descent with Momentum

In addition to the required standard SGD for training our network, which updates at each step all the weights by subtracting the learning rate multiplied by the gradient of the loss with respect to the parameter, we have decided to improve it by including a momentum term as suggested by [3]. In particular, the final step formulation becomes:

$$w_t = w_t - factor_t \quad (4)$$

[4]often improperly labeled convolution.

$$factor_t = \gamma factor_{t-1} + \eta \nabla_\theta J(\theta) \quad (5)$$

As suggested by [6], momentum can help SGD navigate ravines[5] that are common around local optima. Momentum helps accelerate the optimizer towards the relevant direction and dampens oscillations, all by adding a fraction of the update vector of the previous time-step. This optimization helped considerably the training of the final model with our own framework.

IV. Performance Optimizations

The main module and the most complicated of all is the Conv2d module. Therefore, we have spent most of the time correcting and optimizing its implementation. Our first solution used nested for loops to iterate over all input and output channels for both forward and backward implantation. Even though such solution is very intuitive, it turns out that it does not scale well with the number of channels (input and output) and samples that normal deep network require. Because of this, the program was extremely slow when the the neural network size was increased. Furthermore, since the framework is implemented in pure Python, notorious for its inefficient *for loops*, we should make good use of the available Tensor computations that PyTorch provides, given that they rely underneath on a C-based implementation. Therefore, in order to improve our implementation, we can replace the loops with multi-dimensional batch matrix multiplication that Tensors support. We call this implementation 'Improved matrix multiplication', since it is solely based on matrix multiplication and delivers faster execution time compared to our 'Baseline' for loop implementation. We compare the runtime of our implementations of **Conv2d** and **NearestUpsampling** in Tables I - IV, in order to show the effect of increasing the number of samples and channels. All experiments have been run 20 times on a very small 2GB Intel GPU and the average time (ms) has been recorded.

Nearest neighbour upsampling forward	Time (ms)
Improved matrix multiplication (10 samples, C=8)	5.72
Improved matrix multiplication (10 samples, C=16)	22.85
Improved matrix multiplication (100 samples, C=16)	238.99
Baseline (10 samples, C=8)	45.94
Baseline (10 samples, C=16)	213.29
Baseline (100 samples, C=16)	4956.49

TABLE I: Runtime of the forward method of the Upsampling Layer

Table I illustrates that the forward implementation of 'Improved matrix multiplication' scales linearly with the increase of samples, while its scaling with respect to the number of channels in not linear. Still, the speedup compared to the Baseline model is significant. We also note that the Baseline runtime scale is not linear with samples. The same holds for input channels. In Table II, we show the time for running a forward and a backward pass as well as computing a loss, which is equivalent of running one epoch. We observe that the baseline algorithm performs significantly worse due to its inability to scale with increasing input size.

Nearest neighbour upsampling forward and backward	Time (ms)
Improved matrix multiplication (10 samples, C=8)	10.45
Improved matrix multiplication (10 samples, C=16)	42.99
Improved matrix multiplication (100 samples, C=16)	427.89
Baseline (10 samples, C=8)	287.19
Baseline (10 samples, C=16)	1143.90
Baseline (100 samples, C=16)	19 028.43

TABLE II: Joint runtime of the forward and backward methods of the Upsampling Layer

Similar tests are performed on the Conv2d layer in order to compare the two implemented methods. We observe a quadratic complexity in terms of channels and a linear one in terms of samples. The complexity of the baseline method appears to be worse than quadratic.

2D Convolution forward	Time (ms)
Improved matrix multiplication (10 samples, C=8)	15.72
Improved matrix multiplication (10 samples, C=16)	60.06
Improved matrix multiplication (100 samples, C=16)	613.49
Baseline (10 samples, C=8)	50.11
Baseline (10 samples, C=16)	280.2
Baseline (100 samples, C=16)	14 478.34

TABLE III: Runtime of the forward method of Conv2d module

2D Convolution forward and backward	Time (ms)
Improved matrix multiplication (10 samples, C=8)	22.53
Improved matrix multiplication (10 samples, C=16)	93.12
Improved matrix multiplication (100 samples, C=16)	1148.99
Baseline (10 samples, C=8)	332.87
Baseline (10 samples, C=16)	2252.92
Baseline (100 samples, C=16)	39 813.14

TABLE IV: Joint runtime of the forward and backward methods of the 2D Convolutional Layer

V. Noise2Noise architecture

In order to test our implementation of the framework, we have constructed a shallow net for Noise2Noise applications, with the goal of estimating the statistics of the noise and be able to improve the SNR[6] of 32x32 images. The model follows the suggested structure from the assignment as with strided convolutions and upsampling layers, but reduces the number to only 1 upsampling layer. The first convolutional layer has 3 input channels and 16 output channels, while the second convolution has 16 and 32, respectively. Then the upsampling layers scale back the final output to 3 channels. By tuning the learning rate ($\eta = 10^{-6}$) and momentum ($\gamma = 0.9$) of our optimizer we can reach a reasonable 23db PSNR. Despite the performance optimizations, due to the less efficient method of computing gradients compared to *autograd*, we observe 10.7 times performance slowdown with respect to PyTorch.

[5] areas where surface is not uniformly steeped in all dimensions.

[6] Signal to Noise Ratio

REFERENCES

[1] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," *Advances in neural information processing systems*, vol. 25, 2012.

[2] V. Dumoulin and F. Visin, "A guide to convolution arithmetic for deep learning," 2016. [Online]. Available: https://arxiv.org/abs/1603.07285

[3] N. Qian, "On the momentum term in gradient descent learning algorithms," *Neural networks : the official journal of the International Neural Network Society*, vol. 12 1, pp. 145–151, 1999.

[4] K. He, X. Zhang, S. Ren, and J. Sun, "Delving deep into rectifiers: Surpassing human-level performance on imagenet classification," in *2015 IEEE International Conference on Computer Vision (ICCV)*, 2015, pp. 1026–1034.

[5] H. Inada, "Calculate cnn backprop with padding and stride set to 2." [Online]. Available: https://hideyukiinada.github.io/cnn_backprop_strides2.html

[6] S. Ruder, "An overview of gradient descent optimization algorithms," 2016. [Online]. Available: https://arxiv.org/abs/1609.04747