Noise2Noise project report 2

AGNAOU Zineb, JAAKIK Marouane, SHENG Haoyu

May 27, 2022

Abstract

This paper explores the implementation of deep learning frameworks from scratch using elementary operations on Torch tensors. We then evaluate our framework against a noise2noise task that aims to learn how to restore images without Clean Data. In practice we show the different design choices that could be taken for our deep learning framework and compare its performance against state of the art frameworks in terms of latency and accuracy.

1 Introduction

A deep learning framework consist of a software package that allows data scientists to train artificial neural networks. It is designed in a way to be customized by the scientists in terms of building blocks that could be put together to create a model with given criteria for a given task. We call these building blocks network layers and we discuss the underlying algorithms for each. To that end, we use Torch.Tensor library and its elementary operations to implement our building blocks from scratch.

2 Layers

A neural network is made up of neurons that are organized in layers. There are three types of layers: an input layer, an output layer, and multiple hidden layers depending on the the network depth. The process of sending data from one layer to the next is called propagation. There are two types of propagation: forward propagation and backward propagation. In forward propagation, the data moves from input to hidden layer to output. It ends in a prediction based on the input. In backward propagation, a prediction from the output layer is back-tracked from the output to the input layer, which shows the error rate. This is then used to modify the weights and biases of each neuron, giving the neurons with a higher error rate and greater adjustment. It is important to constantly readjust the weights to minimize errors and gain higher accuracy. For more details about the the functions that are implemented in our framework, please checkout the code documentation.

2.1 Sequential

The class Sequential is a sequential container of neural network layers. It groups a linear stack of layers into one layer that would have the aggregate behaviour of all the layers it contains. This sequence of layers is effectively our model. Here the forward function aggregates the forward functions of the contained layers. The output of each forwards constitutes the input the next similarly for the backward function.

2.2 CNN

The motivation behind the implementation of a CNN lies in the nature of the task we are trying to solve. Our data is images and the most appropriate layer is The convolutional layer.

It is the core building block of a CNN, and it is where the majority of computation occurs. It requires a few components, which are input data, kernels, and a feature map. For our project we implement a 2-Dimensional Convolutional layers.

This implies that our kernels are a two-dimensional (2-D) array of weights, which represents part of the image. While they can vary in size, the kernel size is typically a 3x3 matrix; this also determines the size of the receptive field.

The filter is then applied to an area of the image, and a dot product is calculated between the input pixels and the filter. This dot product is then fed into an output array. Afterwards, the filter shifts by a stride, repeating the process until the kernel has swept across the entire image. The final output from the series of dot products is known as convolution.

Note that we use same padding as it ensures that the output layer has the same size as the input layer.

2.3 Pooling

Pooling layers, also known as downsampling, conducts dimensionality reduction in the forward pass, reducing the number of parameters in the input. Similar to the convolutional layer, the pooling operation sweeps a filter across the entire input, but the difference is that this kernel does not have any weights, here the backward also simply modifies the shape of the gradient and upsamples it to match the preceding layer weights shape. Instead, the kernel applies an aggregation function to the values within the receptive field, in our case this aggregate function is the max function.

2.4 Up sampling

Upsampling layers as their name indicates increase dimensionality in the forward pass, increasing the number of parameters in the input. Unlike the pooling layers we don't use filters and an aggregate function we simply double the size of our data along the 2 dimension. We have effectively implemented a 2-Dimensional up-sampling where the backward simply modifies the shape of the gradient and down samples it to match the preceding layer weights shape.

3 Weight Initialization

Weight initialization is an important design choice when it comes to deep learning models. A good weight initializations helps to prevent exploding and vanishing gradients as well making our objective function - criterion to be discussed below - converge faster.

3.1 normal

Sampling weights from a normal distribution.

3.2 kaiming

Similarly the kaiming samples from a normal distribution, but we leverage knowledge about the network architecture and number of parameters to derive an appropriate standard deviation for our sampling distribution. Following the paper He et al. (2015), we initialize the weights of layer l with a zero-mean Gaussian distribution with a standard deviation of $\sqrt{\frac{2}{n_l}}$ with no bias. (where n_l is the number of activations of layer l)

4 Activations

An activation function in a neural network defines how the weighted sum of the input is transformed into an output from a node or nodes in a layer of the network.

4.1 ReLU

The rectified linear activation function or ReLU for short is a piecewise linear function that will output the input directly if it is positive, otherwise, it will output zero. Its simple form provides an gradient that is fast to compute, it also reduces the likelihood of the gradient to vanish. The computations of ReLU are shown in the Table 1.

4.2 Leaky ReLU

For the RELU, when the value is negative, no learning happens as the new weight remains equal to the old weight since the value of the derivative is 0. This is called the "Dead ReLU" issue. We prevent that using the leaky ReLU. The computations of leaky ReLU are shown in the Table 1.

Table 1: Activation functions and their derivatives

Activation Function	Function	Derivative
ReLU	$f(x) = \begin{cases} 0 & \text{for } x < 0 \\ x & \text{for } x \geq 0 \end{cases}$	$f'(x) = \begin{cases} 0 & \text{for } x < 0 \\ 1 & \text{for } x \geq 0 \end{cases}$
Leaky ReLU	$f(x) = \begin{cases} 0.01x & \text{for } x < 0 \\ x & \text{for } x \geq 0 \end{cases}$	$f'(x) = \begin{cases} 0.01 & \text{for } x < 0 \\ 1 & \text{for } x \geq 0 \end{cases}$

5 Criterions

In calculating the error of the model during the optimization process, a loss function must be chosen. The loss informs us about how well the model's current weights perform for the given task and also gives an insight on how to update our the weights for a lower loss after back-propagating it through the network. The choice of cost function is tightly coupled with the choice of output unit. Most of the time, we simply use the cross-entropy between the data distribution and the model distribution or simply a mean squared error for a regression task.

5.1 Mean squared error

Mean squared error (MSE) measures the amount of error in our model. It computes the average squared difference between the input and predicted values. The formula of MSE is: $\sum_{i=1}^{D}(x_i - y_i)^2$ where x_i is input and y_i is output.

6 Optimizers

Once we have chosen our objective function and computed the output loss, we need an algorithm to update our weights effectively given this loss. An optimizer does exactly that. The learning rate controls how aggressively do we perform such an update. Thus, it helps in reducing the overall loss and improve the accuracy.

6.1 Stochastic gradient descent

Gradient descent is an iterative algorithm such that at each step it moves down the slope of our objective function to find its minimum according to the computed gradient. It proves very computationally costly to compute such a gradient when averaging over all the data points hence the stochastic in SGD. We randomly compute the gradient over a mini-batch of data instead of the whole dataset. It converges faster than the classical GD. The updates of parameter θ could be written as: $\theta = \theta - \eta \cdot \nabla_\theta J(\theta; x^{(i:i+n)}; y^{(i:i+n)})$ where n is the batch size.

7 Results

We compare the performance of our DL framework to pytorch in terms of latency performance and accuracy for the given task.

7.1 Latency Performance

We compare the latency for the forward passes of our framework layers vs their counterpart in pytorch

Table 2: Latency Comparisons in seconds of CNN layer

layer	**forward**	**backward**
Ours	3.01 ± .01	2.7 ± .015
Pytorch	1.95 ± .005	1.65 ± .008

7.2 PSNR scores

We compare the PSNR scores between our framework model and its counterpart in pytorch

Table 3: PSNR scores comparisons

Model	**PSNR score**
Ours	Nan
pytorch	25.48
Top-1 entry	36

8 Conclusion

Implementing a deep learning framework from scratch gave us a good insight about the inner workings of classical deep learning architectures and algorithms. It remains very challenging from a software engineering perspective to build a framework comparable to pytorch or tensorflow in terms of speed, accuracy and maniability.

Bibliography

K. He, X. Zhang, S. Ren, and J. Sun. Delving deep into rectifiers: Surpassing human-level performance on imagenet classification. In *Proceedings of the IEEE international conference on computer vision*, pages 1026–1034, 2015.