Report of miniproject 2

François Charroin, Khalfallah Selim, Qing Jun
Deep learning Course

I. Introduction

This miniproject is linked to the other one, the goal is the same: denoising images thanks to neural networks. The particularity of this one is that all the blocks (layers, activation functions, optimizer) have to be written from scratch. Three objectives can be set in this project, the first one is gaining a great understanding of how neural networks work, the second is discovering an application of deep learning, and finally the main one is understanding how a model can be improved.

II. Overview

In AI applications, every single piece of the data observed is influenced by factors of variations. Deep learning solves this main issue by introducing representations expressed in terms of other (more simple) representations. Some examples of deep learning models are the feed forward deep network (Neural Network NN) and Multilayer percreptron (MLP). The framework we implemented builds NN using class sequential and representing MLP that combines linear and nonlinear modules. Each module has a forward and backward method to compute outputs and learn through back propagation. The learning process is done by finding the loss and its gradient and optimizing it using gradient technique such as SGD (stochastic gradient descent).

III. Modules structure

As mentioned in the introduction, several fundamental tools from deep learning, that are usually import from libraries, had to be rewritten in a Module Class which forces all the classes related to it to follow four steps (init, forward, backward, param). The modules can be separated into these main categories:

- Linear modules: Each layer in neural network has parameters to be learned. The forward method depends on these parameters that we initialize to avoid ruining the layers activation outputs. In the backward method we calculate the derivatives and store the gradients for these parameters.

- Non-linear modules (activation functions): All the activation functions don't have parameters thus param would return an empty list. Some of these functions are:
  - ReLU:

    $$f(x) = \begin{cases} x & if x > 0 \\ 0 & if x \leq 0 \end{cases} \tag{1}$$

    The gradient is given by $\frac{\partial L}{\partial x}$ where L is the loss function and x the input.

  - Sigmoid: This activation function is defined as:

    $$f(x) = \frac{1}{1+e^{-x}} \tag{2}$$

    and the gradient is given by:

    $$\frac{\partial L}{\partial x} = \frac{\partial L}{\partial y} * Sigmoid(x) * (1 - Sigmoid(x)) \tag{3}$$

  - ELU[1]: Even though it was not asked to code it, we where eager to understand how this activation function can be used, how it compares with ReLU.

    $$f(x) = \begin{cases} x & if x > 0 \\ \alpha * (e^{-x} - 1) & if x \leq 0 \end{cases} \tag{4}$$

    With alpha = 1. The gradient is multiplied by the mask of values over 0, plus alpha multiply by values under 0

  - SELU[2]: The same goes for this function, it was not asked to code it.

    $$SELU(x) = scale(max(0, x) + min(0, \alpha(exp(x)1))) \tag{5}$$

    with α = 1.6732632423543772848170429916717
    and scale = 1.0507009873554804934193349852946
    The gradient is pretty similar to ELU.

  - Sequential: This module builds all neural networks. It forms blocks with the layers and activation functions which are passed as arguments in a sequential order. It iterates over all the layers in increasing order by applying the forward function of current module on the output of the previous one. On the contrary, the backward function iterates over the layers in a decreasing order with the same principle as the forward one (output of current layer is input of the next one).

- Loss: Each module engender losses in forward (prediction) and backward (derivative propagation) blocks.

- Optimizer: The optimizer used for this miniproject is SGD which stands for stochastic gradient descent. It is used to minimise the objective function $L(\theta)$ with $\theta \in R^{dim}$ representing the model parameters vector. The parameters are updated by a step in the opposite direction of the gradient of $L(\theta)$. SGD uses (x_i, y_i) as a training input to update. SGD equation is given by: $\theta^t = \theta^{t-1} - \mu * \nabla * L(\theta, x_i, y_i)$ where μ is the learning rate.

Other blocks where implemented such as:

- Conv2D:
This function applies a 2D convolution over an input signal composed of several input planes. Some of the parameters are: channels-in the number of channels in the input image, channels-out the number of channels produced by the convolution, the kernel, the stride of the convolution, and the bias. Globally the following results represent the 2d convolution:

$$out\left(N_i, C_{out_j}\right) = bias\left(C_{out_j}\right) + \sum_{k=0}^{C_{in}-1} weight\left(C_{out_j}, k\right) * input\left(N_i, k\right) \quad (6)$$

The Backward function on the other hand, the gradients by the weights and the use of the fold function. The param function return the weight and bias for the forward and backward function.

- Upsampling:
This function applies a nearest neighboor unsampling and then a convolution operator over an input image. As for Conv2D, the parameters are the channels in and out, the kernel, the stride, and finally the bias.

- TransposeConv2D The init and param functions are very similar to the ones from Conv2D. It can be seen as the gradient of Conv2D with respect to its input. The 2D transposed convolution operation is applied on the image tensors. The main feature of a transpose convolution operation is the filter or kernel size and stride.

- MSE:
This class creates a criterion that measures the mean squared (error squared L2 norm) between each element in the input and target. The forward is a mean squared error whereas the backward function contains the derivative.

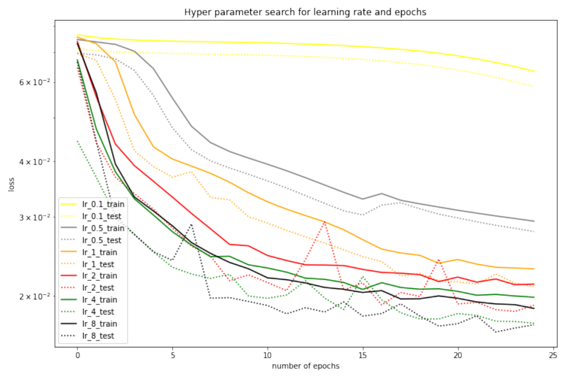
IV. Hyper parameter search

A. *Introduction*

For this section, we want to explore some parameters for this model and find which combination of parameters returns the best results. With the limitation of the training time, we only chose 5,000 images from the dataset for training, and we used a batch size of 100, which can limit the training time with 25 epochs to 5 minutes. The learning rate, training epochs, and activation will be researched in the following sections.

B. *Learning rate and epochs*

The figure 1 represents the losses for training and testing under different learning rate. With only 25 epochs, we can see there is no overfitting problem. The test loss always goes down and even smaller than the train loss with the increasing of training epochs. When the learning rate is increased, the loss became smaller. The learning rate 4 and 8 performs the best, and we decide to choose learning rate 4 since the results seem more stable.



/

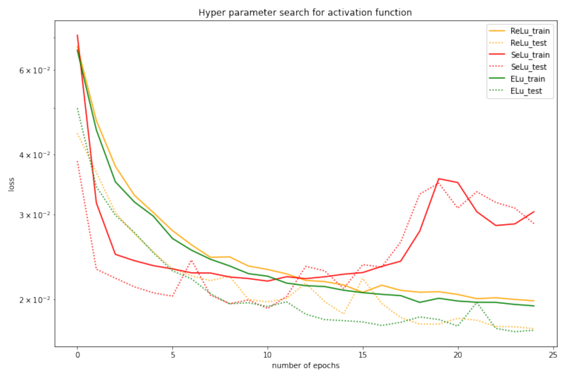
Figure 1. Learning rate search

C. *Activation function*

We also wanted to find the performance of different activation functions, the SeLu and ELu compared to ReLu.The figure 3 compares the three methods. It's obvious that the Elu gives the best results with in 25 epochs, closely followed by the ReLu. The loss of SeLu goes down fastest within 10 epochs, while the loss rise after then and it's best performance is not satisfying.To conclude, the ELu performs best.

D. *Search for parameter number on layers*

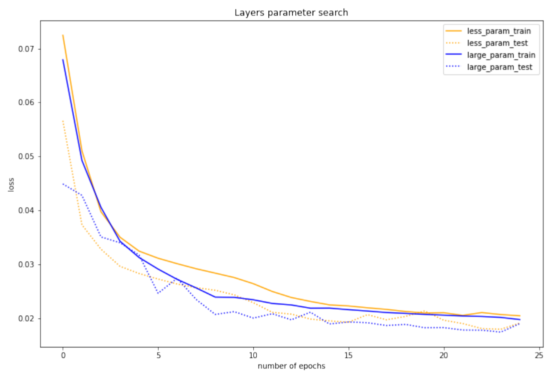
Besides the learning rate and activation function, we also want to see how parameter number in convolution layer would affect the results. We compared the structure with channels from 3 to 256 to another from 3 to 96 in convolution layer. The results can be seen in the figure



/

Figure 2. Activation function search

3. From the figure3, it's easy to find the one with more parameters get better results which is quite intuitive.



/

Figure 3. Layer parameter search

V. Challenges

The second part of this project had many challenges. One of which was about understanding the general structure of the different blocks, and understanding how they interacts with each others. The second great challenge was about understanding how to transpose and reshape the different tensors. A lot of testing on jupyter notebook had to be done. In complement to the TransposeConv2D function we used, we tried to implement Upsampling with nearest neighbors and then convolution but had poor results by using it. The color accuracy seems good with it but the definition is lowered. The upsampling function has been added to the final code despite the poor results because developing it and testing was an interesting process. To help us understand a bit more how the functions should behaves, teaching assistant were a big help, also some online ressources have been used[3][4][5] [6][7].

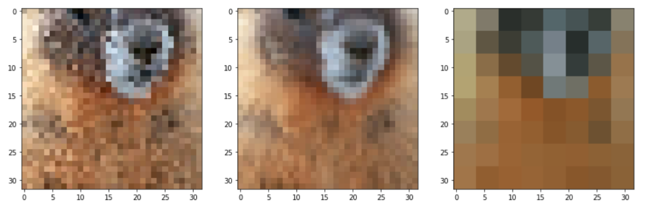


Figure 4. The first image is the input, the second the target, the third the result by using upsampling

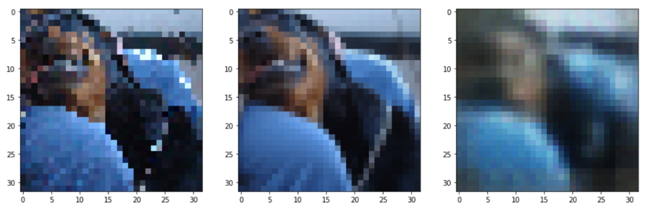


Figure 5. The first image is the input, the second the target, the third the result by using transposeConv2D

VI. Conclusion

In a nutshell, this miniproject allowed us to discover the principles of modules and models in deep learning and helped us enjoy to put it into practice to solve some modern ML problems, it also helps us to work with deep learning libraries and designing our layers, which is useful as some module like transformers and custom optimizers are frequently used in this field.

References

[1] ELU, pytorch documentation: https://pytorch.org/docs/stable/generated/torch.nn.ELU.html

[2] SELU, pytorch documentation: https://pytorch.org/docs/stable/generated/torch.nn.SELU.html

[3] Conv2D, pytorch documentation: https://pytorch.org/docs/stable/generated/torch.nn.Conv2d.html

[4] ConvTranspose2D, pytorch documentation: https://pytorch.org/docs/stable/generated/torch.nn.ConvTranspose2d.html

[5] repeat interleave, pytorch documentation: https://pytorch.org/docs/stable/generated/torch.repeat_interleave.html

[6] eriklindernoren (2019), github: https://github.com/eriklindernoren/ML-From-Scratch

[7] ddbourgin (2020), github: https://github.com/ddbourgin/numpy-ml