Teaching a Convolutional Neural Network to perform transformations

Mihai Bojescu
Master in Artificial Intelligence and Optimisation
Faculty of Computer Science of University "Alexandru Ioan Cuza" of Iași
Iași, Romania
bojescu.mihai@gmail.com

***Abstract*—This document contains details on how to make a Convolutional Neural Network learn to perform certain transformations, its architecture, the loss function used when training, the results of the model and comparisons between the results of the model and the actual transformations.**

***Index Terms*—neural network, convolution, convolutional neural network, nn, cnn, transformation, images, loss function**

I. Introduction

Convolutional neural networks can be trained to perform a lot of image-related tasks, most of them falling into two categories: regression and classification. This document details how a Convolutional Neural Network was "taught" how to perform the following transformations:

1) resize the image from 32×32 pixels to 28×28 pixels
2) grayscale the image
3) horizontal flip the image
4) vertical flip the image

The following chapter will detail the architecture of the network and its parameters.

II. Architecture

The model in this project is composed from the following layers:

1) 2 Convolutional layers
2) 1 Max pooling 2D layer
3) 2 Fully-connected layers

Each convolutional layer goes through a MaxPooling2D pooling function. The result of these layers go through the fully-connected layers. The model was constructed to be relatively lightweight such that it can perform the "taught" transformations with haste, and relatively accurate such that its outputs can be more beliveable.

III. Layer parameters

The aforementioned layers used in the model described in this document use the following parameters:

1) 1^{st} Convolutional layer:
   - Input channels: 3
   - Output channels: 32
   - Kernel size: 3
2) 2^{nd} Convolutional layer:
   - Input channels: 32
   - Output channels: 64
   - Kernel size: 3
3) The Max pooling 2D layer:
   - Kernel size: 2
   - Stride: 2
   - Padding: 0
4) 1^{st} Fully-connected layer:
   - Size of $(64 * 36) \times 512$
5) 2^{nd} Fully-connected layer:
   - Size of $512 \times (28 * 28)$

IV. The loss function

For the training of the model, the "MSE" loss was picked for the following reasons:

1) It penalizes larger error more heavily than smaller ones due to the squaring operation. As this model performs image regression, large errors are more easily seen, thus they should be minimised the most.
2) It provides easy-to-understand results. The calculations can be performed on paper in an easier way compared to a sigmoid function.
3) It is a commonly used loss function for various regression tasks, including image regression.

The chosen loss function has a few downsides though:

1) It is sensitive to outliers (example: intensity 0 pixels, intensity 255 pixels), which will disproportionately infuence the loss.
2) It does not produce accurate results pixel-wise, thus the images will look blurry.

V. Early stopping

In this project, a new concept was introduced to the model trainer, which is "early stopping". Early stopping is a way to decrease the chance of model overfitting by stopping the training process once the model did not minimise the loss for a certain amount of epochs, named "delay" or "patience".

For the current iteration of the model trainer, a "patience" quota of 5 epochs was chosen. This hyperparameter was chosen from some observations of the model's behaviour, which:

- stopped *too early* when given $patience < 5$
- overfitted *too much* when given $patience > 5$, as it trained for all the epochs

VI. Training

When training, the losses trended downards, bottoming at 50 epochs:

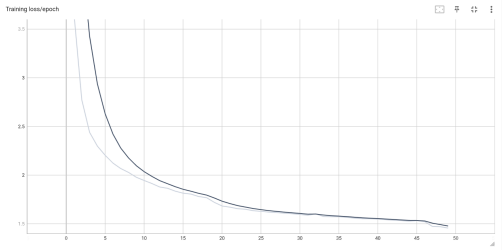

Fig. 1: Training loss graph

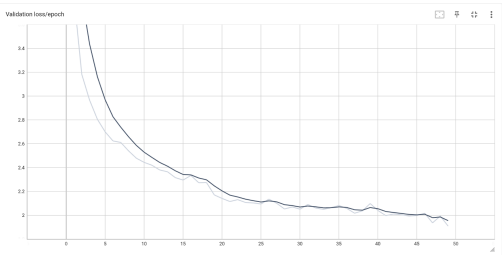

Fig. 2: Validation loss graph

While the accuracy trended upwards in a constant manner, plateaing at about 50 epochs:

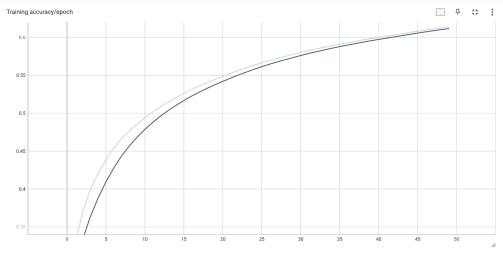

Fig. 3: Training accuracy graph

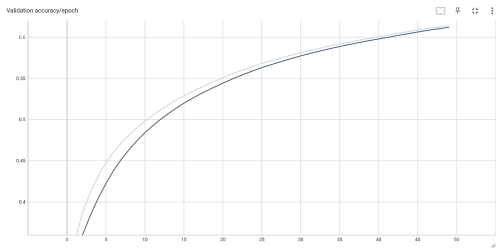

Fig. 4: Validation accuracy graph

VII. Results

A. Outputs and comparison with the ground truths

The model can produce fairly convincing pictures, especially when the inputs present the following traits:

1) A consistent background
2) A high contrast
3) Many straight edges

The results are not perfect though, as they are really blurry and present some distinct artifacts. The model faces problems especially with:

1) Pets against backgrounds
2) Low contrast pictures

Some results, compared to the ground truths for the model would be:

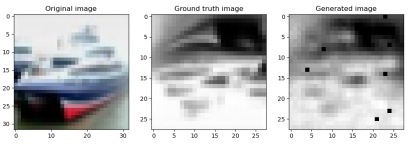

Fig. 5: A docked boat

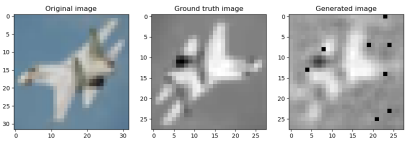

Fig. 6: A plane in the sky

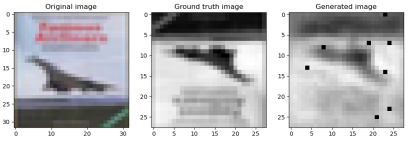

Fig. 7: A bird in the sky

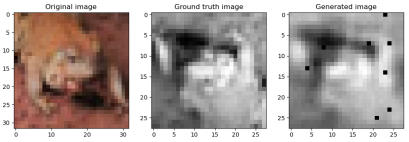

Fig. 8: A frog on rocks

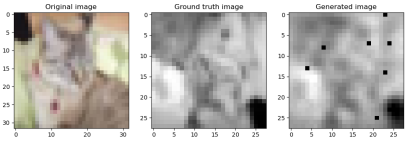

Fig. 9: A cat against a background

B. Timings

During the experimentation phase, the following trends could be seen for the previously outlined model:

1) The model performed the actions slower than the transformations on the CPU
2) The model performed the actions faster than the transformations on the GPU

The observations were performed when running the model on the following configurations:

1) CPU runs: 1 x Intel Core i7 1260p
2) GPU runs: 1 x Nvidia RTX A5000

NOTE: The CPU and GPU runs used different CPUs, as the GPU was rented from a cloud provider. This is the reason for the rather big differences in the timings for the transformations.

An abstract of the results would be the following:

Run	Timing for transformations	Timing for model
1	1.1458s	1.2898s
2	0.9764s	1.2944s
3	0.9786s	1.1745s
4	0.9837s	1.2515s
5	0.9729s	1.3458s

Fig. 10: Timing results for running the trained model on the CPU

Run	Timing for transformations	Timing for model
1	1.5567s	0.4800s
2	1.5094s	0.4755s
3	1.5700s	0.4562s
4	1.5671s	0.5263s
5	1.5739s	0.4927s

Fig. 11: Timing results for running the trained model on the GPU

The experiments above back up the trends observed for the model.

VIII. Conclusions

While a simple fully-connected neural network stuggles with beliveable results when performing regression tasks (such as Lab04), a convolutional neural network can perform these actions better, abeit with a higher computational cost.

While being stochastic in nature, Neural networks can also be "taught" to perform actions that deterministic algorithms do with great amounts of accuracy, with *good enough* performance on a CPU and better performance on an accelerator such as a GPU.