Politecnico di Torino

POLITECNICO DI TORINO

Master's Degree in
INGEGNERIA INFORMATICA/
COMPUTER ENGINEERING

Master's Degree Thesis

New strategies for Deep Neural Networks explainability

Supervisors

Prof. Ernesto SANCHEZ

Dr. Annachiara RUOSPO

Candidate

Cosmin VRINCEANU

July 2022

Abstract

Convolutional Neural Networks (CNNs) are ubiquitous and seamlessly integrated in our lives. One reason for this is that increasingly powerful machines have empowered a shift towards models of increased complexity. These models enable superhuman predictive accuracy obtained through deep learning paradigms. The trade-off, however, is that the produced results are hardly explainable [1]. This may be a problem. The issue of explainability is particularly important when it is necessary to answer questions about a CNN's reliability and resilience in case of faults.

This thesis aims at shining a light on a CNN's internal behavior by building a tool that enables researchers to observe, in a 3D virtual environment, how different artificial neurons forming the network contribute to the classification of a given input. The tool accepts a description of how the network is built and takes as input a file describing the network parameters.

As a case study, LeNet was used in this thesis. LeNet is a CNN designed to recognize handwritten digits. Due to its simplicity, today it is often used as a learning tool. A 3D representation of this network is computed from an input image of size 28×28 pixels and rendered to the screen with the ability to move around, and zoom into its layers. The most expensive computational operations are executed resorting to the system GPU, improving in this way the system performance.

A CNN's architecture tries to mimic that of a biological brain. This means that, just like their biological counterpart, CNNs have several artificial neurons, which process data, and a number of artificial synapses interconnecting the neurons. The tool offers a number of functions allowing users to study specific subsets of artificial synapses, to inject different types of faults into specific artificial neurons, and to automate those operations via external scripting. The end result can be explored in real time by "flying" in the 3D virtual environment with a mouse and keyboard or by looking at a log file that gets updated on user interactions.

Lastly, the thesis takes into consideration a case study based on LeNet. The network is initially able to correctly classify an input image but carefully placed faults slowly erode the confidence level. As a result, a wrong classification is provided.

Summary

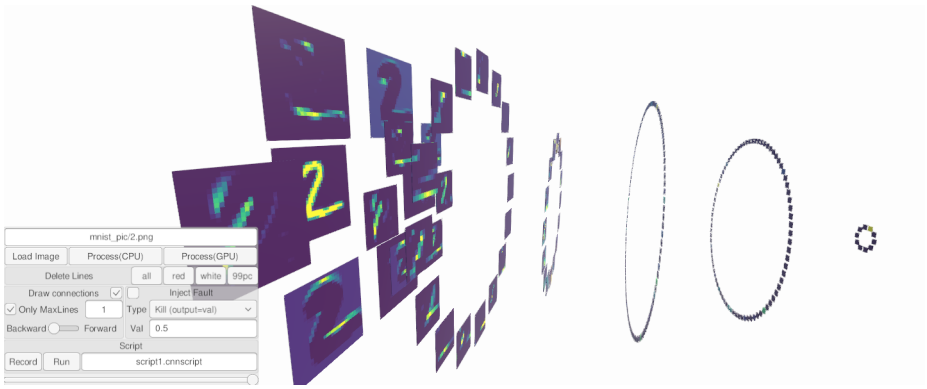


Figure 1: Screenshot of the tool created for this thesis

This thesis aims to offer a new way to analyze convolutional neural networks (CNNs). A tool was built for this scope. With minimal modifications, the tool can load a number of CNNs. It can process their inferences and keep track of all intermediate passages ranging from the initial loading of an input image up to the final classification. Many options are available to study individual neurons and synapses, to discover how they are interconnected in a single inference, and to inject faults in strategic positions. The downstream consequences of the injected faults can then be observed. The tool also allows for external scripting to automate operations over a wide range of inputs and conditions. Scripts can be either hand written or recorded from the user interface.

To prove the tool's functionality, LeNet is loaded and studied. LeNet is a very well studied CNN used to recognize handwritten digits. It consists of 7 layers, of which in the first four it alternates between convolutional and pooling layers with the last three layers being fully connected ones. The output of this network is a set of 10 probability values indicating respectively the network's confidence for the digits from 0 to 9.

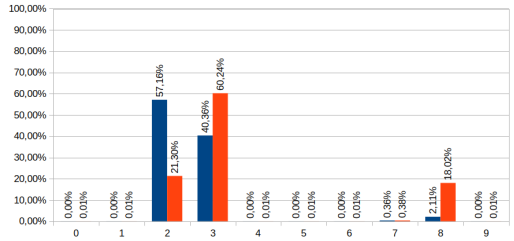


Figure 2: Experiment 1
Before vs **After** fault injection
Incorrect classification becomes correct

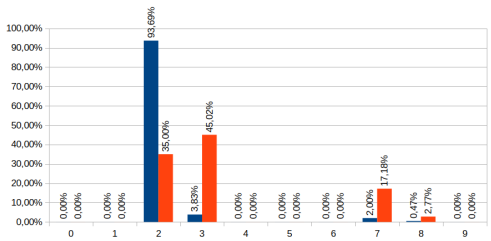


Figure 3: Experiment 2
Before vs **After** fault injection
Correct classification becomes incorrect

As experiments, two input images are loaded into the tool and processed by LeNet. In the first experiment, the network is unable to correctly classify the digit. By using the tool to study the inference process and look at each layer and neuron contained within, it is possible to deduce which neurons had the most influence in the wrong classification. Fault injection campaigns, localized in the pinpointed neurons, show that by killing the responsible neurons, the network now correctly classifies the input image. The second experiment is similar in methodology but starts from a correct classification that degrades gracefully under properly localized small scale fault injections until the classification is no longer correct.

The tool enables users to study a wide variety of CNNs and apply different techniques to try to explain the process behind inferences. This is done in the context of reliability analyses. Reliability can be defined as the probability that a hardware fault causes a failure [2]. This means that when CNNs are considered as mere software or mathematical abstractions, reliability issues are less influential, also due to their natural tendency towards neuron overprovisioning. The situation changes however if they are run on dedicated multiprocessor systems-on-a-chip (MPSoCs). A need arises to classify neurons as either critical or non-critical in this situations such that critical neurons may be uniformly distributed to the MPSoC's processing elements [3].

Table of Contents

Chapter 1

Introduction

Published in November 1998, the paper named "Gradient-Based Learning Applied to Document Recognition" by Yann LeCun, Léon Bottou, Yoshua Bengio and Patrick Haffner [4] offers an interesting analysis of how LeNet, a Convolutional Neural Network (CNN) designed to recognise handwritten characters, fares against standard handwritten digit recognition benchmarks. The paper reviews a number of strategies in building and training this CNN such that the end result outperforms all other techniques at the time, thus ensuring its recognition today as a basis in which many more advanced CNNs find their foundation.

Today, due to their performance often surpassing human-level accuracy [5], CNNs are very attractive solutions for particularly difficult tasks such as self-driving cars, where they are used for object classification and recognition, robotics, and many space related applications like earth observation [6].

Generally, artificial neural networks are considered inherently fault resistant due to two main reasons: their distributed and parallel architecture and the redundancy introduced by over-provisioning [7]. ANNs are generally supplied with an excess of neurons as to what would be minimally required for their computations. This means they can sustain a certain degree of errors before their performance slowly decreases [8]. The situation changes however when ANNs are no longer mere software or mathematical abstractions but run on dedicated multiprocessor systems-on-a-chip (MPSoCs) where some processing elements might be shared [3].

A need emerges, in these cases, to differentiate between critical or non-critical sets of neurons and explain how they contribute to the classification. With this thesis we try to answer the aforementioned need by creating a tool enabling researchers to visualise, browse and analyse a CNN. In chapter 3, we detail its features and the reasoning behind them. As a case study, we then use it to implement and examine LeNet in chapter 4. The thesis continues with chapter 5 as we look at a series of experimental results where the tool is able to explain how certain sets of neurons, when injected with faults, are able to greatly impact classification results.

Chapter 2

Background

This chapter introduces background knowledge on the topics dealt with throughout the thesis. Initially, section 2.1 offers an overview of neural networks and their main characteristics in terms of architecture, followed by an introduction to explainability and reliability in section 2.2. Then, to better follow the case study (chapter 4) presented in this thesis, section 2.3 takes a look at the types of layers most commonly found in a CNN. The chapter then concludes with a summary of LeNet's architecture and main characteristics in section 2.4.

2.1 Artificial Neural Networks

An Artificial Neural Network (ANN) is a collection of algorithms that work in succession on an input by emulating processes typically observed in biological brains. ANNs are usually built as a collection of individual nodes, called artificial neurons, connected by artificial synapses and organised in layers. Each connection carries a weight that simulates a synaptic connection's strength in its biological counterpart. The weights across a network can be changed to simulate feedback and correct for errors in the presence of external stimuli. The process of carefully adjusting those weights is called training.

An ANN is usually trained for a specific task. The training is done using a set of already classified data. This process is very resource intensive and is usually done in cycles, or epochs, reiterating over the training data set. At the end of each epoch, an accuracy test is conducted, and when the result is above the desired threshold, the training phase can be considered complete. For the results to be meaningful, accuracy tests are done using a dataset that is disjoint from the training set. In the case of classifier type ANNs, the accuracy is calculated as the number of correct inferences divided by the total number of inferences or, more accurately, as

$$\frac{TruePositives + TrueNegatives}{TruePositives + TrueNegatives + FalsePositives + FalseNegatives} \tag{2.1}$$

One of the many appeals of ANNs is that the execution of an inference is, in general, front loaded during the training phase, and the prediction phase is often computationally efficient. This means that the trained network can be easily employed in a range of mobile or lower power devices while preserving the ability to classify an image in real time.[9]

2.2 Explainability in the context of reliability

Because of how they are built and trained, ANNs often make it hard to understand and explain how their inferences were reached in human terms. This can be partially traced to the reason itself behind the extensive adoption of ANNs in recent years. Easier access to more powerful machines, improved learning algorithms, and the availability of vast amounts of data brought a shift towards models of increased complexity. These models enable superhuman predictive accuracy obtained through a deep learning paradigm. The trade-off is that we can no longer produce explainable and interpretable predictions [1]. This is not to say that the literature is devoid of explainable models. Many available options like linear regression or decision-tree based models produce perfectly explainable predictions. Those models, however, are still far from state-of-the-art performance.

When it comes to non-explainable models, trust can also be an important issue. In fields like healthcare or self-driving cars, where a wrong inference could lead to loss of life, choosing and training the models can elicit a moral dilemma. The situation does not improve when morality and fairness issues arise. Examples of such issues can be the trolley problem[10] for self-driving cars or skin colour discrimination in face recognition networks [11].

Reliability can be defined as the probability that a hardware fault causes a failure [2]. Analyses assessing a network's reliability are regulated by standards depending on the application domain and are usually conducted through fault injection campaigns [12]. In this context, explaining and interpreting an ANN's predictions and, therefore, understanding how a decision was reached can also guide the best approach to ensure a system is fault resistant and, therefore, more reliable. When ANNs are implemented on application specific integrated circuits (ASICs), physical hardware processing elements may be shared among different parts of the network. In these circumstances, it is very important to be able to single out critical vs non-critical portions of the network based on which artificial

neurons have the most impact on a prediction. Once identified, critical parts may also be protected with additional levels of redundancy. [3]

2.3 Convolutional Neural Networks

Convolutional Neural Networks (CNN) are a class of ANN where at least one layer uses an operation called convolution. These networks are preponderantly used to analyse images but they can also find applications with all types of temporal, spatial, and spatiotemporal data [13]. The images are treated as matrices of real numbers where the network's layers apply a series of transformations. These transformations are based on the assumption that there are strong spatial dependencies in local regions. An important property of image data is that it exhibits a certain level of translation invariance [13]: a number drawn in the center of an image is still the same number if we translate it up or down.

This section will explore three types of layers commonly encountered in a CNN. Those are convolutional layers in subsection 2.3.1, pooling layers in subsection 2.3.2 and fully connected layers in subsection 2.3.3.

2.3.1 Convolutional Layers

A CNN extracts features from an image by means of a convolution against a kernel. Convolution, in this case, is the dot product operation consisting of two matrices **A** and **B** acting respectively as the input image and the kernel. In **A** we place every pixel's color value, while in **B** we have the CNN's trained parameters. The convolution overlaps **B** over **A** as a sliding window and sums the products of items with the same coordinates. The window is then shifted by an amount we call stride, and the process is repeated to find the next values.
For clarity, an example with smaller matrices is provided bellow:

$$\mathbf{A} = \begin{bmatrix} a_1 & a_2 & a_3 \\ b_1 & b_2 & b_3 \\ c_1 & c_2 & c_3 \end{bmatrix} \tag{2.2}$$

$$\mathbf{B} = \begin{bmatrix} d_1 & d_2 \\ e_1 & e_2 \end{bmatrix} \tag{2.3}$$

$$conv(\mathbf{A},\mathbf{B}) = \begin{bmatrix} d_1a_1 + d_2a2 + e_1b_1 + e_2b_2 & d_1a_2 + d_2a3 + e_1b_2 + e_2b_3 \\ d_1b_1 + d_2b2 + e_1c_1 + e_2c_2 & d_1b_2 + d_2b3 + e_1c_2 + e_2c_3 \end{bmatrix} \tag{2.4}$$

A convolutional layer, in general, accepts an input of

$$Width_{in} \times Height_{in} \times Channels_{in} \tag{2.5}$$

and requires four configuration variables:

1. number of filters **K**,
2. their spatial extent **F**,
3. the stride **S**,
4. the amount of zero padding **ZP**.

The result of a convolution is also a matrix. Its size can be determined by the following:

$$Width_{out} = \frac{Width_{in} - F + 2ZP}{S} + 1 \tag{2.6}$$

$$Height_{out} = \frac{Height_{in} - F + 2ZP}{S} + 1 \tag{2.7}$$

$$Channels_{out} = K \tag{2.8}$$

With these variables, we can easily calculate the number of trainable parameters in a convolutional layer as:

$$N_{trainableparameters} = F^2 * Channels_{in} * K + K \tag{2.9}$$

The final term is a trainable bias parameter that is different for each kernel. This number is added to the weight of each artificial neuron in a channel at the end of the convolution.

In a convolutional layer, the end result of the convolution operation is passed through an activation function before feeding it as input to the following layer. Commonly used activation functions are the sigmoid function:

$$S(x) = \frac{1}{1 + e^{-x}} \tag{2.10}$$

and the ReLu function:

$$f(x) = max(0, x) \tag{2.11}$$

2.3.2 Pooling Layers

Pooling layers are typically positioned after a convolutional layer and are used to down-sample data. This is a useful operation when we take into account a convolutional layer's inherent sensibility to any movement in the input image. Down-sampling allows us to reach a result that is less sensitive to small changes in the position of the extracted features, thus slightly mitigating the dangers of

overfitting. However, this translation invariant effect depends on the stride and often comes as a trade-off for generalisation [14]. Another added advantage is the computational complexity reduction obtained by reducing the data size.

Typically, pooling layers can be of two types:

1. **Average Pooling** where the output value is calculated as the average of the values inside a window

2. **Max Pooling** where the output value is calculated as the maximum inside a window

A pooling layer accepts as input of

$$Width_{in} \times Height_{in} \times Channels_{in} \tag{2.12}$$

requires two configuration parameters:

1. spatial extent **F**,

2. stride **S**.

and produces an output such that:

$$Width_{out} = \frac{Width_{in} - F}{S} + 1 \tag{2.13}$$

$$Height_{out} = \frac{Height_{in} - F}{S} + 1 \tag{2.14}$$

$$Channels_{out} = Channels_{in} \tag{2.15}$$

Pooling layers have no trainable parameters since they compute a fixed function of the input.

An example with **F**=2 and **S**=1 is given for clarity using the matrix **A** defined in 2.2:

$$\textbf{AveragePooling(A)} = \frac{1}{4}\begin{bmatrix} a_1 + a_2 + b_1 + b_2 & a_2 + a_3 + b_2 + b_3 \\ b_1 + b_2 + c_1 + c_2 & b_2 + b_3 + c_2 + c_3 \end{bmatrix} \tag{2.16}$$

$$\textbf{MaxPooling(A)} = \begin{bmatrix} max(a_1, a_2, b_1, b_2) & max(a_2, a_3, b_2, b_3) \\ max(b_1, b_2, c_1, c_2) & max(b_2, b_3, c_2, c_3) \end{bmatrix} \tag{2.17}$$

2.3.3 Fully Connected Layers

In a fully connected layer, every artificial neuron in the input is connected to every artificial neuron in the output. This means that every output neuron looks at the full input volume. A fully connected layer accepts an input of

$$Width_{in} \times Height_{in} \times Channels_{in} \tag{2.18}$$

and requires as a configuration parameter the number of output artificial neurons. We can calculate the number of trainable parameters as

$$N_{trainableparameters} = N_{inputNeurons} * N_{outputNeurons} + N_{outputNeurons} \tag{2.19}$$

with the final term referring to a bias for each output neuron.

The final layer in a CNN is often a fully connected one. In this case, a Softmax activation function is used to have the inference output as a probability value that adds to a value of 1 when summed over all the classes recognisable by the network. The output of each neuron $\mathbf{z_i}$ therefore becomes

$$SoftMax(z)_i = \frac{e^{z_i}}{\sum_{k=1}^{N_{outputNeurons}} e^{z_k}} \tag{2.20}$$

It can easily be derived from Equation 2.20 that

$$\sum_{i=1}^{N} SoftMax(z)_i = \sum_{i=1}^{N} \frac{e^{z_i}}{\sum_{k=1}^{N} e^{z_k}} = 1 \tag{2.21}$$

2.4 LeNet

LeNet is a CNN composed of 7 layers. Its task is to classify handwritten digits into ten categories ranging from 0 to 9. The input image is a grayscale of size 32×32 pixels with a black background and the number drawn in lighter shades. This is usually provided as a 28×28 pixels image; in this case, a zero-pad of 2 is specified in the first layer's description. Lenet's layers are:

1. **C1**: a convolutional layer (subsection 2.3.1)
2. **P1**: a pooling layer (subsection 2.3.2)
3. **C2**: a convolutional layer
4. **P2**: a pooling layer
5. **FC1**: a fully connected layer (subsection 2.3.3)

6. **FC2**: a fully connected layer

7. **FC3**: a fully connected layer

The first step in executing LeNet consists in normalizing the input image. In order to do so, the mean and standard deviation are calculated as

$$mean(image) = \frac{\sum_{i=0,j=0}^{W,H} image[i][j]}{W * H} \tag{2.22}$$

$$sd(image) = \frac{\sum_{i=0,j=0}^{W,H} (image[i][j] - mean)^2}{W * H} \tag{2.23}$$

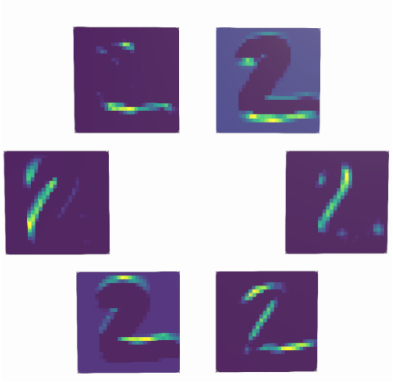
with W being the width of the image and H the height. Each pixel is then normalized such that

$$\begin{aligned} \forall \quad & 0 \leq i \leq W \quad , \quad 0 \leq j \leq H \\ & normalized\_image[i][j] = (image[i][j] - mean)/sd \end{aligned} \tag{2.24}$$

Following the inference process, we can look at LeNet layer by layer:

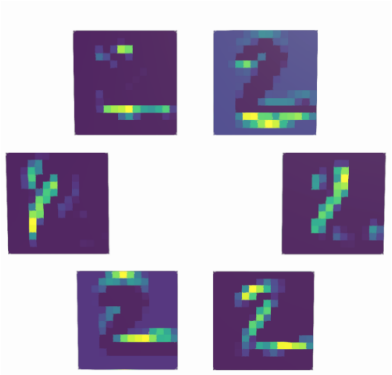
Layer C1 applies a convolution against 6 5×5 kernels with a stride of 1. This generates 6 feature maps of size 28×28 pixels each. We say that each feature map is a channel and thus the final image is

$$28 \times 28 \times 6$$



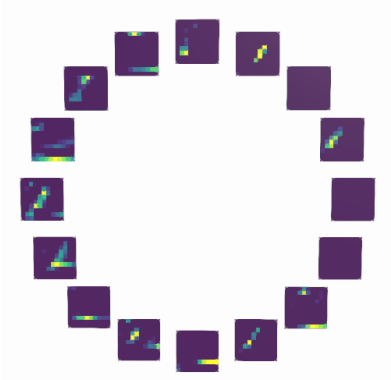
Layer P1 is a pooling layer with spatial extent $F = 2$ and stride $S = 2$. Using equations 2.13, 2.14, and 2.15 we calculate that the output size is

$$14 \times 14 \times 6$$



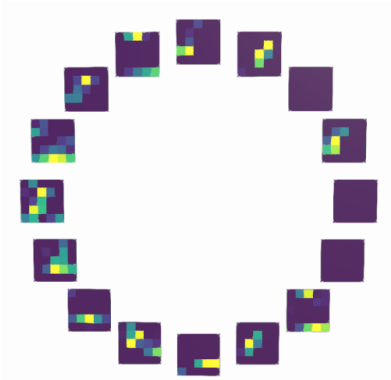
Layer C2 applies a convolution against 6×16 kernels of size 5×5 with a stride of 1 generating 16 feature maps of size 10×10 each (Equation 2.6 and Equation 2.7). This brings the output of this layer to a size of

$$10 \times 10 \times 16$$



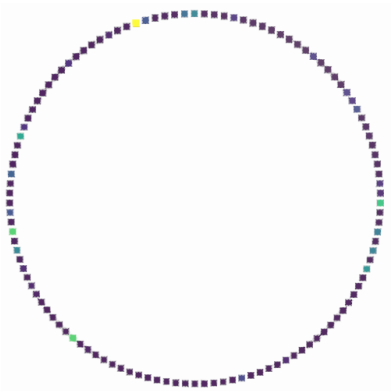
Layer P2 is a pooling layer with spatial extent $F = 2$ and stride $S = 2$. Using equations 2.13, 2.14, and 2.15 we calculate that the output is of size

$$5 \times 5 \times 16$$

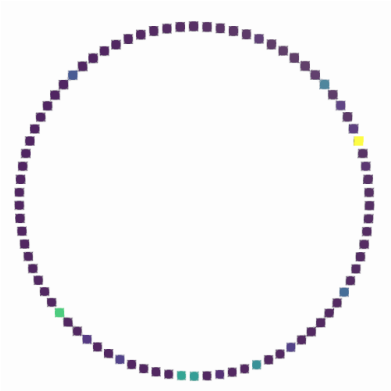


Layer FC1 is the first fully connected layer in this network but it's output is also calculated as a convolution against 16×120 kernels of size 5×5. Using Equation 2.6 and Equation 2.7 we find that the output of this layer is

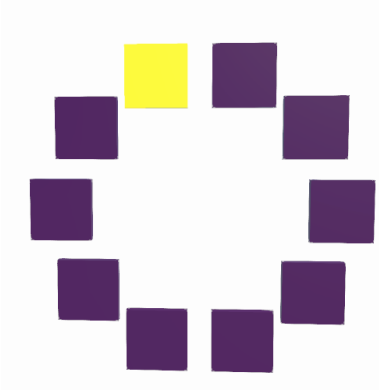
$$1 \times 1 \times 120$$



Layer FC2 is the second fully connected layer and has 84 artificial neurons.



Layer FC3 has 10 artificial neurons, representing the final classification of the network. In this layer a SoftMax activation function is used such that Equation 2.21 applies.



2.4.1 Training LeNet

LeNet is commonly trained as an exercise in learning about neural networks. The initial values of its trainable parameters are randomly assigned, and a backpropagation algorithm is used to slowly modify them until the accuracy as expressed in Equation 2.1 is above the desired threshold.

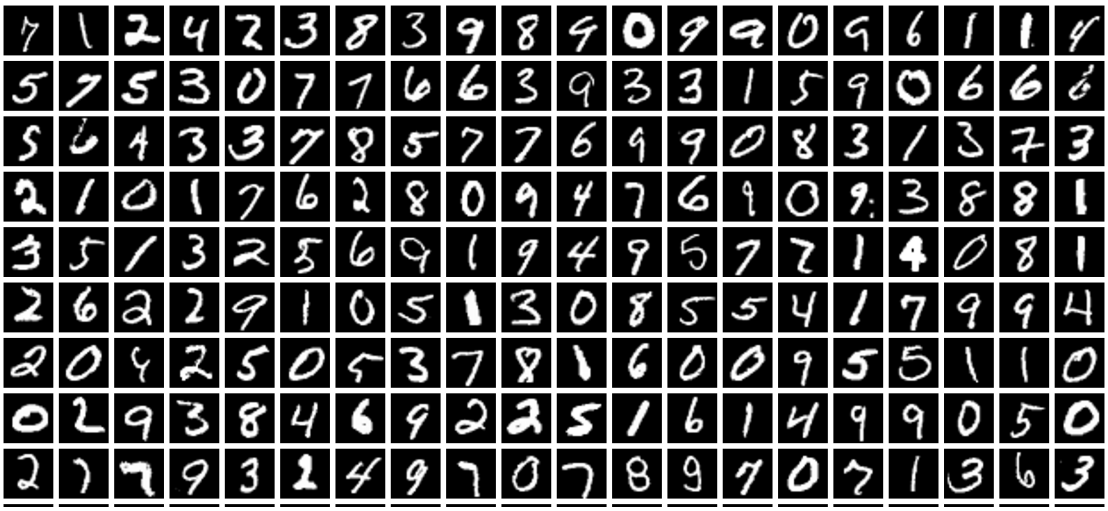


Figure 2.1: Snippet of some digits in the MNIST dataset

The MNIST dataset is a vast collection of handwritten digits which is commonly used for this purpose. The images are labeled and divided in two sets:

1. **Training Set**, composed of 60000 images
2. **Testing Set**, composed of 10000 images

Each convolutional and fully connected layer in LeNet holds a number of parameters.

Using equations Equation 2.9 and Equation 2.19, we can calculate their number for each layer:

1. **C1** has 1 input channel and 6 kernels of size 5×5, therefore

$$N_{C1} = 5 * 5 * 6 + 6 = 156$$

2. **P1** has no trainable parameters
3. **C2** has 6 input channels and 16 kernels of size 5×5, therefore

$$N_{C2} = 5 * 5 * 6 * 16 + 16 = 2416$$

4. **P2** has no trainable parameters
5. **FC1** has 16 input channels and 120 kernels of size 5×5, therefore

$$N_{FC1} = 5 * 5 * 16 * 120 + 120 = 48000$$

6. **FC2** has 120 input channels and 84 artificial neurons, therefore

$$N_{FC2} = 120 * 84 + 84 = 10164$$

7. **FC3** has 84 input channels and 10 artificial neurons, therefore

$$N_{FC3} = 84 * 10 + 10 = 850$$

Summing the above, the total number of trainable parameters in this network is

$$N_{LeNet} = N_{C1} + N_{C2} + N_{FC1} + N_{FC2} + N_{CF3} =$$

$$156 + 2416 + 48000 + 10164 + 850 = 61586$$

Chapter 3

Proposed approach

The proposed approach stems from the need to visualise a CNN's internal workings and analyse how artificial neurons and artificial synapses interact to produce a correct classification.
This entailed the creation of a tool able to:

1. **Have advanced graphic capabilities out of the box**
   since being able to observe, select, move and selectively enable or disable parts of the network can be a valuable ability when trying to visualise and understand it.
2. **Easily be reprogrammed to run any generic CNNs**
   provided that we have knowledge of their internal architecture and a file containing the necessary weights. Being able to quickly adapt the tool to any CNN can also allow comparative analyses where behaviours observed in small networks can be reproduced in more complex ones.
3. **Compute and track the results a CNN outputs**
   while also storing data about its individual neurons and synapses. This is to allow a degree of scrutiny into how neurons interact and aid in mapping them by criticality.
4. **Allow the user to strategically inject faults in specific neurons**,
   with multiple types of faults available and the ability to follow their influences on the network in real time.
5. **Accept external scripting and script recording**
   to easily reiterate findings and operations over a vast set of inputs. Scripts can be either recorded from within the tool or be manually written and loaded with the appropriate menu.
6. **Log interactions with the network**
   so that operations and results can be analysed at a later time.

3.1 Describing a generic CNN

The tool is developed to allow the study of any generic CNN. To ensure this, a generalised way to describe CNNs was needed and this thesis aims to provide one in terms of architecture. A CNN can be described as a list of layers, of different types, such that the first layer accepts a single image as input and the last layer offers the final classification. Each layer can be simplistically described in terms of its hyper-parameters, as described in section 2.3, and of a number of parameters obtained by training the network.

This structure is mimicked in the tool by a class, simply called ***CNN***, where the standard behavior of a convolutional neural network is modeled. ***CNN*** contains a list of ***CNN_Layer***s, an abstract class defining the common behavior every layer in the network should display.

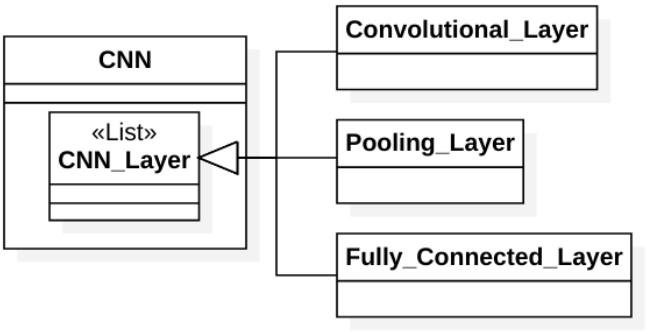


Figure 3.1: Class Diagram

The main features of a ***CNN_Layer*** are:

1. an input buffer
2. an output buffer
3. a name used to easily locate the layer
4. a ***process()*** method which inheriting classes have to implement in order to handle a layer type's specialized logic.

Currently, three classes extend ***CNN_Layer*** to implement Convolutional, Pooling and Fully Connected layers. Among these, *Convolutional_Layer* uses a *Compute Shader* to offload computations to the GPU and improve overall processing times.

The **CNN** class handles loading the first layer's input buffer. The input buffer is automatically filled with the previous layer's output buffer for each following layer.

3.2 User interface

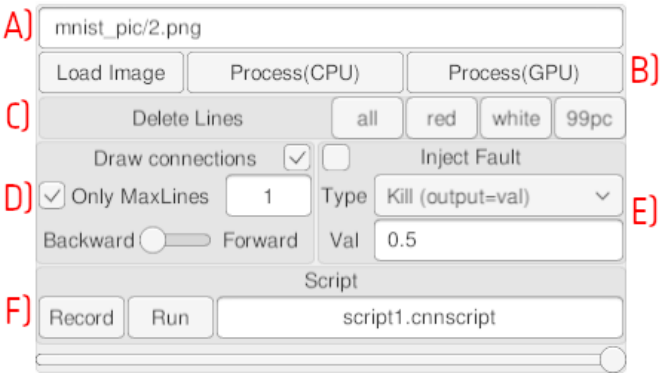


Figure 3.2: The tool's user interface

The tool offers users various ways to interact with the loaded CNN. Figure 3.2A is a text box accepting an input image's path. This path can be processed by the buttons in Figure 3.2B to load the image for preview or process the inference using either the CPU or the GPU. The two methods are internally implemented differently with the GPU method being much faster than the CPU. One noteworthy difference is in the floating point supported by the GPU method which allows only 32bit single precision floating point data. The CPU method allows full 64bit double precision.

When the *Draw connections* option is selected in Figure 3.2D, each click on a neuron will draw a series of lines on the screen, portraying the synapses connecting it either to neurons in upper or lower layers, depending on the switch in the lower side of the same section. An additional check box for *Only MaxLines* is available to limit the synapses being drawn to only those N with higher weight. The buttons in Figure 3.2C will delete, in order:

1. All lines
2. All lines with a weight above 0
3. All lines with a weight equal to 0
4. 99% of the lines, meaning that every line is deleted except for the strongest connections in the network.

Selecting *Inject Fault* in Figure 3.2E allows the user to inject a certain type of fault in any clicked neuron. The available types are:

1. Kill
   if this is selected, the killed neuron will always output the value written below at the moment of its death.

2. Stun
   if this is selected, the stunned neuron will output a different pseudo random generated value for each inference.

3. Bitflip
   if this is selected, the bitflipped neuron will output its normal value with flipped bits in the positions specified by the bitmask entered as a value.

The section in Figure 3.2F allows a user to either run an external script, by specifying its path, or record a new script. Recording mode opens a bigger text box that gets automatically populated as the user interacts with the network. All interactions are logged there in a format readily parsable by the tool, and a dedicated button allows the user to save the new script to storage. This scripting section allows the user to automate a series of actions on a network by iterating them over a broad set of input images.

3.3 Visualising the layers

As a project choice, it was decided to draw each layer as a circular collection of channels where each channel is a square map of individual neurons. This means that channels form regular polygons around an imaginary center line in convolutional and pooling layers. The polygons have as many vertices as there are channels in the layer. In fully connected layers, each individual neuron can be considered the sole neuron in a channel, and therefore, the layer's shape appears more like a circle.

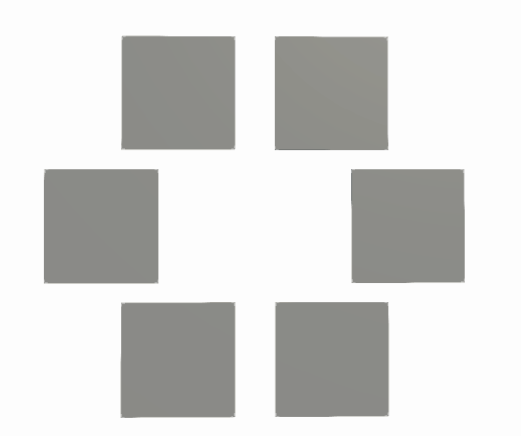


Figure 3.3: representation of convolutional or pooling layers

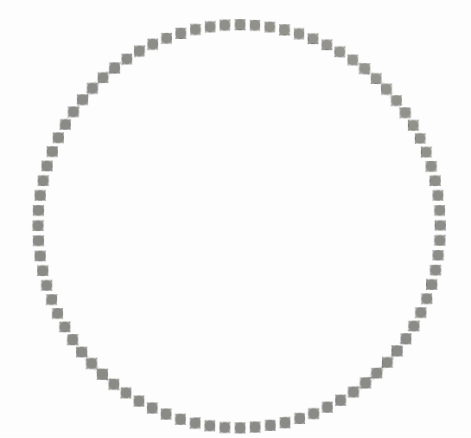


Figure 3.4: representation of fully connected layers

When drawn, an artificial neuron's color depends on its value. A simple color choice would be just to map the value to a 0-255 grayscale. This, however, tends to create some visualisation issues that can end up hiding features in a channel.

Figure 3.5: Visualization of the Viridis color map

Comparative analyses of different color map types show that **Viridis** is one of the most accurate color maps currently used to visualise data [15] so it is used instead. Viridis maps values inside a normalised range to colors extending from a deep purple to a bright yellow. An interesting property of this color map is its perceptual uniformity, meaning that closer values will map to closer looking colors while maintaining a good separation between values that are far apart. Given how colors are distributed in the range, it is also very robust to colorblindness [16].

Chapter 4

Case study

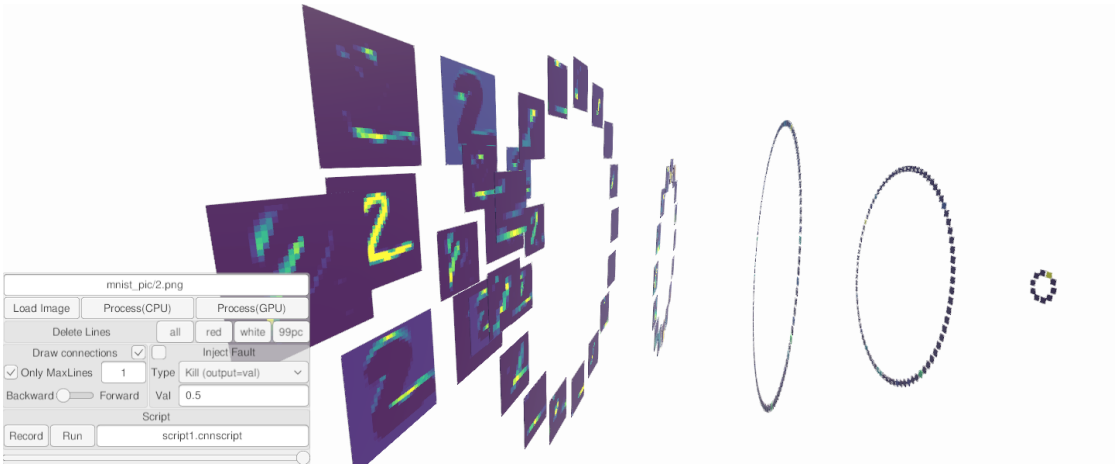


Figure 4.1: a screenshot of the tool loaded with LeNet

A great part of this thesis consisted in coding and testing the tool. In order to do so, access to high level 3D graphics primitives was needed. There are many available solutions that provide this but very few allow for an equally direct access to the GPU. Unity was thus chosen as the engine supporting this tool.

The language chosen to develop this tool is C#. This is due to Unity directly allowing C# scripting to better customise the behavior and interactions between 3D objects. C# is a general purpose object oriented language with strong static typing. It is often used in applications dealing with the .Net framework. In the tool, C# is used both to develop the logic behind generic CNNs, and to connect interface inputs such as button presses, checkbox state changes, sliders and neuron clicks to actions enabling the user to interact with the network.

4.1 Loading a generic CNN

In order to load a generic CNN in the tool, the first steps are to understand its architecture in terms of configuration variables (hyper-parameters) characterising each layer (as expressed in section 2.3) and build a file containing the network's trained parameters. The file is not necessarily bound to a specific format or syntax, as it needs to be parsed manually in the tool's code. However, the way kernels are flattened in convolutional layers needs to follow a specific order (or be parsed in a way that they eventually do) in order to maintain a single parameter order convention throughout the project. A flattening example is provided in pseudo-code for a convolutional layer's parameters:

```
parameters[output_channels][input_channels][height][width];
n = 0;
for ( 0 <= i < output_channels )
    for ( 0 <= j < input_channels )
        for ( 0 <= k < height )
            for ( 0 <= l < width )
                flat_parameters[n++] = parameters[i][j][k][l]
```

Figure 4.2: Convolutional Layer flattening pseudocode

A CNN object is created and, after each layer is initialised in its constructor with a name, a set of weights and a set of biases, we can assign it to the CNN's layers list in the appropriate position.

```
c1 = new Convolutional_Layer(name="c1_name", weights=c1_filters, bias=c1_bias);
p1 = new Pooling_Layer("p1_name", size=2, stride=2);
fc1 = new Fully_Connected_Layer(name="fc1_name", weights=FC1_weights, bias=fc1_bias);

CNN MyCNN = new CNN("my_cnn_name");
MyCNN.layers[0] = c1;
MyCNN.layers[1] = p1;
MyCNN.layers[2] = fc1;

MyCNN.process(LoadImage("path"));
MyCNN.draw();
```

Figure 4.3: Simplified pseudocode to load a generic CNN

To evaluate an inference it is sufficient to call the *.process()* method on the CNN object and pass in the input image as a parameter. This can be followed by *.draw()* if we want to also draw the network on screen.

4.2 Loading LeNet

As proof of the tool's capabilities, we load LeNet. In order to do so we refer to section 2.4 for each layer's hyper-parameters and we train the network from scratch using [17] and the MNIST dataset as explained in subsection 2.4.1. Once the accuracy is above 90%, the matrices containing the trained parameters can be flattened and serialized to a file. This file is then parsed in a function in the tool's code as in Figure 4.4,

```
var json = File.ReadLines("Assets/data_flat.json")
                .SkipWhile(line => !line.Contains("C1_weight"))
                .Skip(1)
                .Take(1).ToList<String>();
float[] C1_weights = JsonConvert.DeserializeObject<float[]>(json[0]);
```

Figure 4.4: Parsing LeNet's trained parameters from file

and used to create the layer objects as in Figure 4.5.

```
Convolutional_Layer conv1 = new Convolutional_Layer("conv1",c1_filters);
Pooling_Layer pool1 = new Pooling_Layer("pool1",2,2);
```

Figure 4.5: Creating layer objects

Lastly, the CNN object can be created and populated with LeNet's layers as in Figure 4.6.

```
CNN  Lenet5 = new CNN ("Lenet5-prova3");
Lenet5.layers[0] = conv1;
Lenet5.layers[1] = pool1;
Lenet5.layers[2] = conv2;
Lenet5.layers[3] = pool2;
Lenet5.layers[4] = fc1;
Lenet5.layers[5] = fc2;
Lenet5.layers[6] = fc3;
```

Figure 4.6: Creating layer objects

Chapter 5

Experimental results

In this section we introduce three experiments collected through the study of LeNet with the tool developed for this thesis. The experimental results were obtained using both images from the MNIST training dataset and derived ones obtained by editing sources from that same dataset to hide parts of images that were being recognised as features. In section 5.1, we present an in depth walk-through study of an inference. With section 5.2, the objective of the experiment is to test that the tool allows us to understand which part of the network is crucial in an incorrect inference. By finding the right artificial neurons to inject with faults so that their output is suppressed, we notice that the inference is now correct. With section 5.3 we approach a similar situation but from a different direction. The network is initially able to correctly classify the input image but, by studying some of its inactive parts, we find that if a fault were to be injected in some of them, the network's performance would gracefully degrade until the inference is no longer correct.

5.1 Experiment 1: A walk-through study

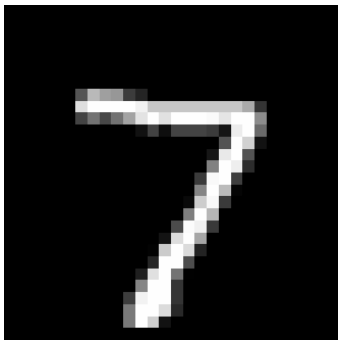


Figure 5.1: The first image in the MNIST training dataset.

In this experiment we are going to study the inference calculated from Figure 5.1. The input image is loaded in the tool where it gets zero-padded, normalised and displayed using the veridis color map (Figure 5.2). The tool is configured to processes the image layer by layer and display it in a 3D environment where the user can rotate and move around it. Our first observation is that LeNet correctly classifies this input image as a 7. This information can be visually extracted by looking at the last layer in the network, FC3 (Figure 5.3), where the most probable classification appears yellow. The layer is to be read starting from the left and going clockwise. In this case, the classification carries a confidence factor of **0.999999999084410%** in double precision. By clicking the seventh neuron in FC3, we can explore how it is connected to neurons upstream by drawing on screen its incoming synapses. Randomly doing so, however, would end up producing a number of drawn synapses so high that it would be impossible to extract any information from it. It is thus important in this case to limit the number of synapses on screen to those we find most significant. To do so, we ensure that **Draw Connections** as in Figure 3.2D is selected with **OnlyMaxLines** checked and the direction set to backwards. The value next to **OnlyMaxLines** needs to be chosen carefully as limiting the number of synapses to a value that is too low could potentially hide useful data, while setting a value that is too high might hinder our ability to properly study the network by drowning out important information. At any time, by using the buttons in Figure 3.2B, we can decide to delete all or a subset of the drawn synapses.

Figure 5.2: Loaded Figure 5.1

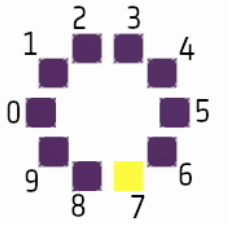


Figure 5.3: Layer FC3 obtained from Figure 5.1

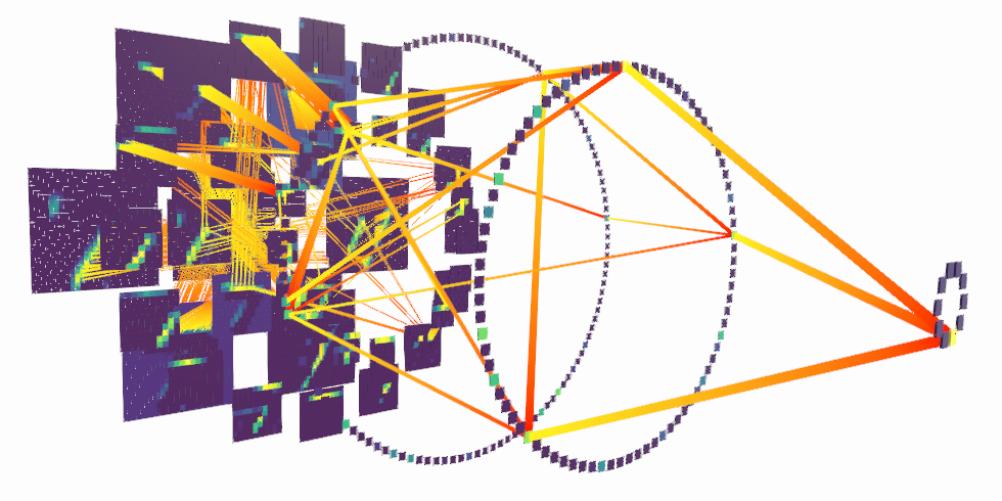


Figure 5.4: Network view after click on FC3[7] with $OnlyMaxLines = 3$

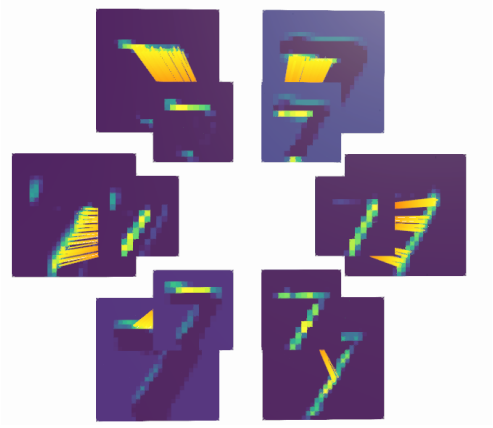


Figure 5.5: Zoomed in network with only C1 and P1 visible

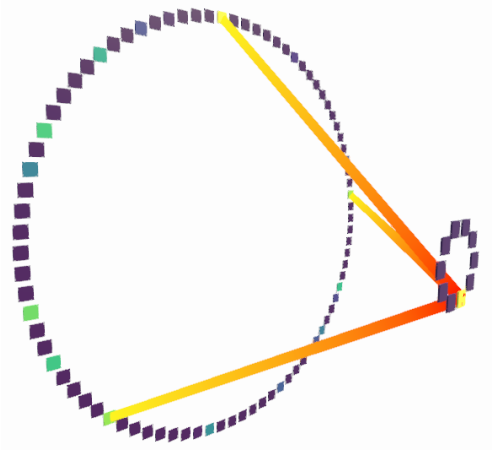


Figure 5.6: Zoomed in network with only FC2 and FC3 visible

Synapses are drawn wider or narrower based on the weight of the associated connection. When the weight is 0, synapses are drawn in white. By studying how they connect different parts of the network, we can extrapolate which features are being extracted and considered in a layer and which are inconsequential according to the training experienced by our network.

Using the commands in Figure 3.2E, we can test the assumptions made in the previous paragraphs by injecting faults in strategic positions. Fault injected neurons are visually depicted in red. After any number of faults have been

injected, the network needs to be computed again in order to see their effects. As a test, we can inject kill faults ($output = 0$) in the three most significant neurons in Figure 5.6. The result is that the network's inference confidence decreases:

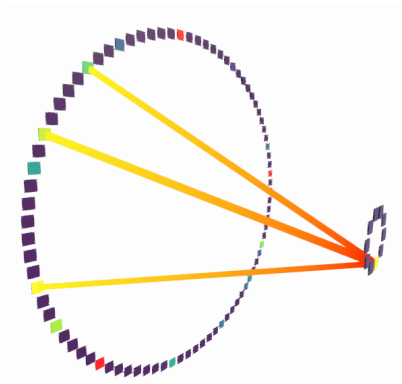


Figure 5.7: Figure 5.6 after the most significant neurons were fault injected

$$before : 0.999999999084410\%$$

$$after : 0.999987126472159\%$$

but the classification stays correct. This is because the neurons we have killed were not critical to this inference.

We can script this operation to be repeated with different input images. To do so, we click the Record button in Figure 3.2F, change the picture path in Figure 3.2A and click Process. We can then see that the panel gets automatically filled with commands as in Figure 5.8. These commands can be hand edited in the tool or saved to a file and opened with any preferred editor. In order to repeat our study on multiple images, we can append an arbitrary number of

*load pic /some_path/mnist_pic/**N**.png*

to the file, write its path in the tool and run it.

Figure 5.8: Scripting window filling with commands

The output produced by running this script can be seen in real time in the tool or studied later by analysing the logs.

5.2 Experiment 2

Figure 5.9: Image 983 from the MNIST training dataset.

One of the images that better showcases the tool's capabilities is image 983. In the MNIST dataset, this image is labeled as a **3**. However, the trained LeNet used for this thesis incorrectly classifies 983 as a **2**. Loading the image with the tool, we can look at the network and examine the process that resulted in this classification.

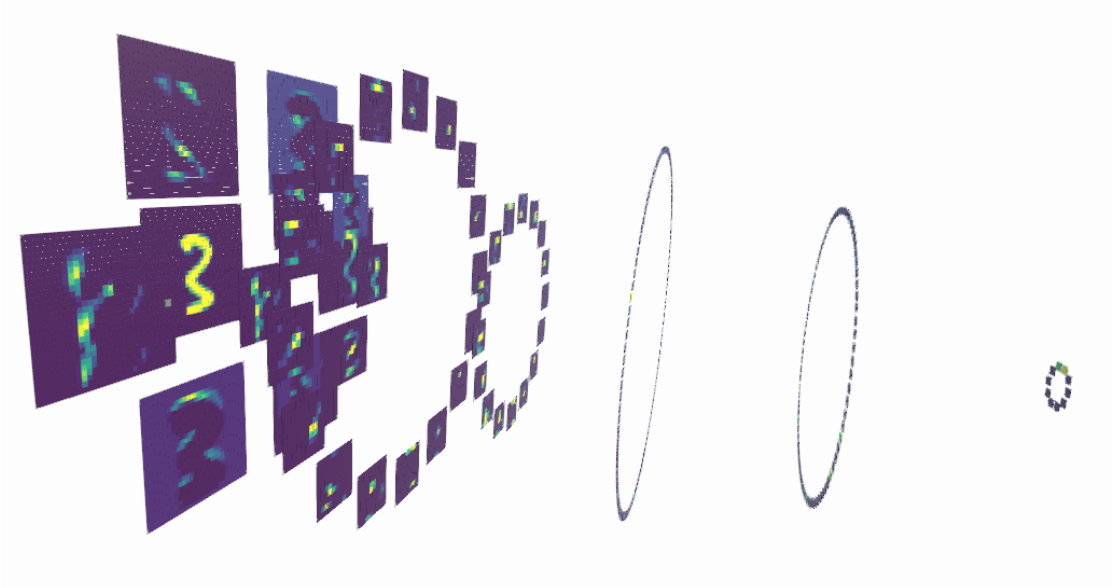


Figure 5.10: Complete network with image 983 loaded

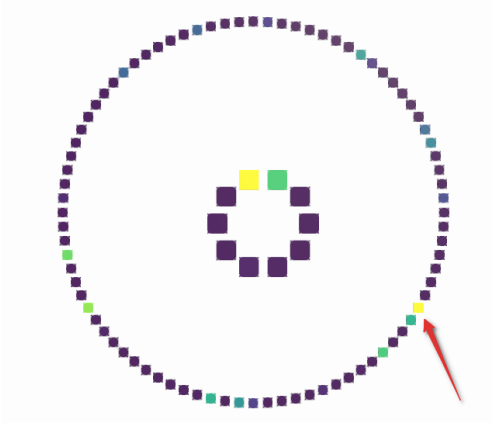


Figure 5.11: Zoomed in network with only FC2 and FC3 visible, channel 49 marked with a red arrow

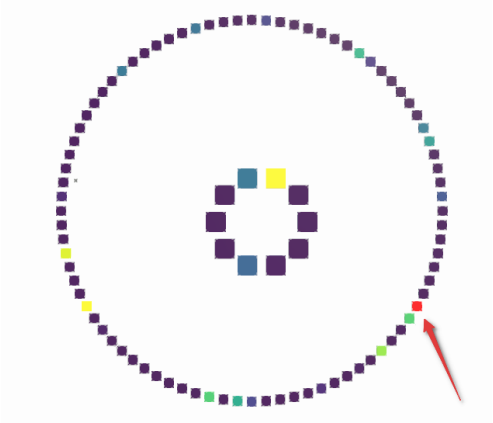


Figure 5.12: Same as Figure 5.16 but with a kill fault injected in channel 49

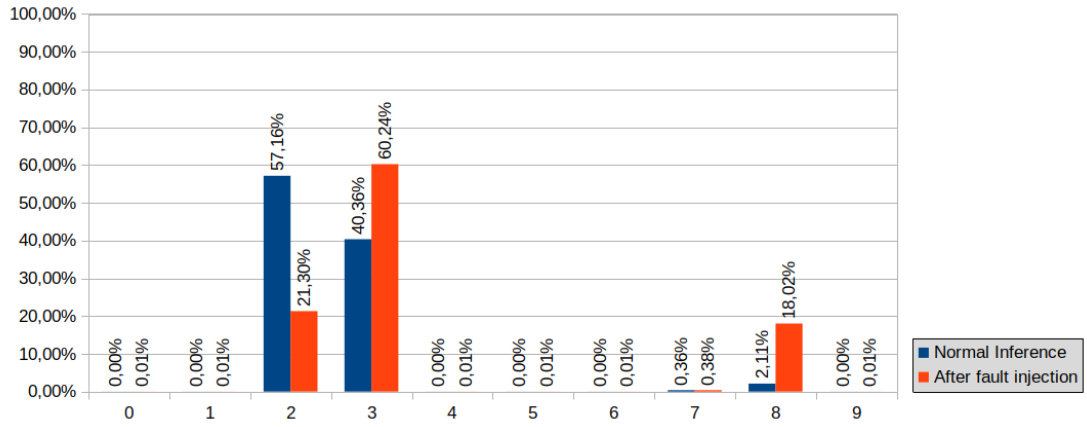


Figure 5.13: Chart comparing confidence percentages before and after fault injection

We observe that channel 49 in the second fully connected layer has an activated artificial neuron with a very high value (indicated by its color as explained in [chapter talking about veridis, to be linked]). By injecting a fault of type *kill* with a value of 0, we can observe in Figure 5.17 and Figure 5.13 that the inference results have changed a lot. The image is now correctly recognized as the digit **3**.

5.3 Experiment 3

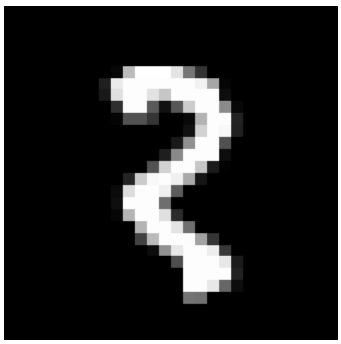


Figure 5.14: 983A, modified version of image 983 from the MNIST training dataset.

For this experiment we load a modified version of 983 which we call 983A. In this modified version, some pixels in the lower left corner have been deleted with an external photo editing software. From a human point of view, this makes the image look closer to a **2** than a **3**. LeNet's inference also correctly classifies this as the digit **2** with a very high confidence percentage. Looking at the network, however,

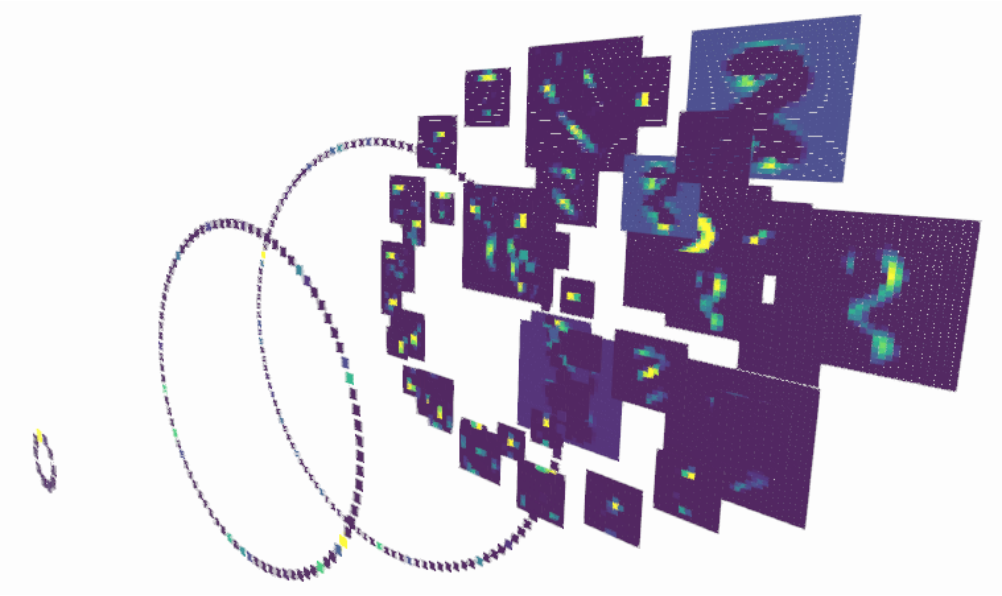


Figure 5.15: Complete network with image 983A loaded

we observe that some feature maps in P2 are completely empty. If we browse the features nearby, the maximum discernible raw activation value is around 10.

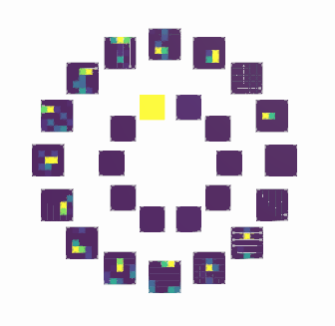


Figure 5.16: Zoomed in network with only P2 and FC3 visible

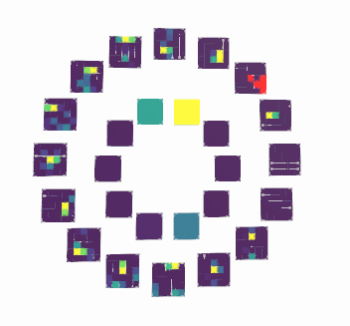


Figure 5.17: Same as Figure 5.16 but with kill faults injected in channel 6

By injecting a number of *kill* faults with a value of 10 in channel 6 of layer P2, we can observe that the confidence value slowly decreases until a wrong inference is provided.

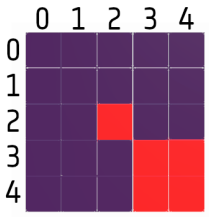


Figure 5.18: Layer P2, channel 6

Fault injected at	Classified digit	Confidence
-	2	93,69%
(2,2)	2	82,63%
(2,2),(3,3)	2	72,51%
(2,2),(3,3),(4,4)	2	67,21%
(2,2),(3,3),(4,4),(4,3)	2	47,42%
(2,2),(3,3),(4,4),(4,3),(3,4)	3	45,02%

Table 5.1: Confidence level decreasing as more faults are injected

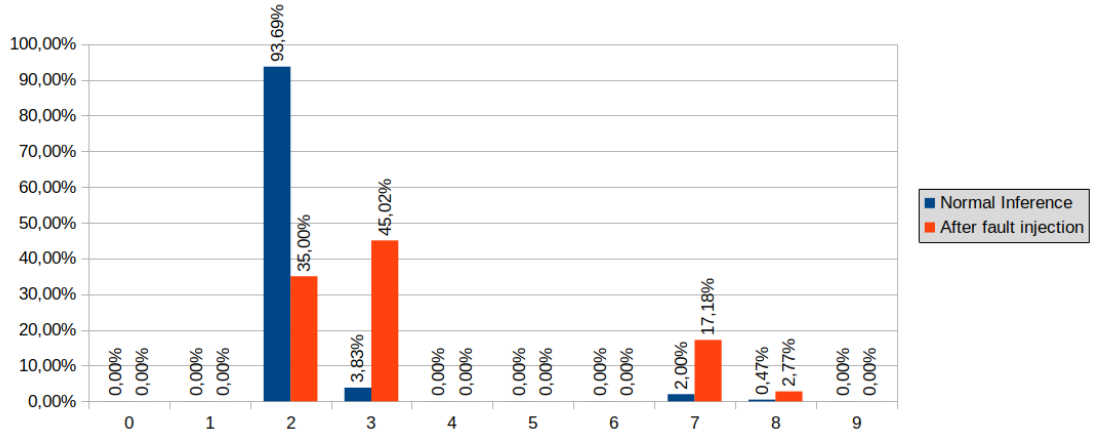


Figure 5.19: Chart comparing confidence percentages before and after fault injection

Chapter 6

Conclusions and future work

This thesis provides a tool to better visualise and study Convolutional Neural Networks. The experimental results show that, by making use of the functions provided, it is possible to reach a better understanding of the internal workings of a CNN. By carefully studying a neuron's output over a wide enough number of iterations, it is possible to test its criticality in a controlled environment with direct access to every part of the network.

Future work may extend the tool's functionality by implementing a way to keep track of overall sessions containing multiple inferences. This would also be useful when working on multiple networks in parallel, trying to extrapolate behaviours in a simpler network and then test for them in a more complex one. The fault injection mode could also be upgraded to include additional types of faults and the possibility of mass fault injection following a number of user defined patterns.

Acronyms

AI
: artificial intelligence

ANN
: artificial neural network

CNN
: convolutional neural network

DNN
: deep neural network

ASIC
: application specific integrated circuit

MPSoC
: multi processor system on chip

Bibliography

[1] Pantelis Linardatos, Vasilis Papastefanopoulos, and Sotiris Kotsiantis. «Explainable AI: A Review of Machine Learning Interpretability Methods». In: *Entropy* 23.1 (2021). ISSN: 1099-4300. DOI: 10.3390/e23010018. URL: https://www.mdpi.com/1099-4300/23/1/18 (cit. on pp. i, 3).

[2] Giorgio Di Natale, Dimitris Gizopoulos, Stefano Di Carlo, Alberto Bosio, and Ramon Canal. *Cross-Layer Reliability of Computing Systems*. iet - the institution of engineering and technology, Jan. 2020, pp. 1–328. URL: https://hal.archives-ouvertes.fr/hal-02986877 (cit. on pp. ii, 3).

[3] Annachiara Ruospo and Ernesto Sanchez. «On the Reliability Assessment of Artificial Neural Networks Running on AI-Oriented MPSoCs». In: *Applied Sciences* 11.14 (2021). ISSN: 2076-3417. DOI: 10.3390/app11146455. URL: https://www.mdpi.com/2076-3417/11/14/6455 (cit. on pp. ii, 1, 4).

[4] Y. Lecun, L. Bottou, Y. Bengio, and P. Haffner. «Gradient-based learning applied to document recognition». In: *Proceedings of the IEEE* 86.11 (1998), pp. 2278–2324. DOI: 10.1109/5.726791 (cit. on p. 1).

[5] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. «Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification». In: *IEEE International Conference on Computer Vision (ICCV 2015)* 1502 (Feb. 2015). DOI: 10.1109/ICCV.2015.123 (cit. on p. 1).

[6] Vasileios Syrris and Sveinung Loekken. «Editorial of Special Issue "Machine and Deep Learning for Earth Observation Data Analysis"». In: *Remote Sensing* 13.14 (2021). ISSN: 2072-4292. DOI: 10.3390/rs13142758. URL: https://www.mdpi.com/2072-4292/13/14/2758 (cit. on p. 1).

[7] Steve Lawrence, C. Giles, and Ah Tsoi. «What Size Neural Network Gives Optimal Generalization? Convergence Properties of Backpropagation». In: (Mar. 2001) (cit. on p. 1).

[8] El Mahdi El Mhamdi and Rachid Guerraoui. «When Neurons Fail». In: (Jan. 2016). DOI: 10.1109/IPDPS.2017.66 (cit. on p. 1).

[9] Charu C. Aggarwal. *Neural Networks and Deep Learning. A Textbook*. Cham: Springer, 2018, p. 30. ISBN: 978-3-319-94462-3. DOI: 10.1007/978-3-319-94463-0 (cit. on p. 3).

[10] Sven Nyholm and Jilles Smids. «The ethics of accident-algorithms for self-driving cars: An applied trolley problem?» In: *Ethical theory and moral practice* 19.5 (2016), pp. 1275–1289 (cit. on p. 3).

[11] Sidney Perkowitz. «The Bias in the Machine: Facial Recognition Technology and Racial Disparities». In: *MIT Case Studies in Social and Ethical Responsibilities of Computing* Winter 2021 (Feb. 2021). https://mit-serc.pubpub.org/pub/bias-in-machine (cit. on p. 3).

[12] Annachiara Ruospo, Ernesto Sanchez, Marcello Traiola, Ian O'Connor, and Alberto Bosio. «Investigating data representation for efficient and reliable Convolutional Neural Networks». In: *Microprocessors and Microsystems* 86 (2021), p. 104318. ISSN: 0141-9331. DOI: https://doi.org/10.1016/j.micpro.2021.104318. URL: https://www.sciencedirect.com/science/article/pii/S0141933121004786 (cit. on p. 3).

[13] Charu C. Aggarwal. *Neural Networks and Deep Learning. A Textbook*. Cham: Springer, 2018, p. 315. ISBN: 978-3-319-94462-3. DOI: 10.1007/978-3-319-94463-0 (cit. on p. 4).

[14] Coenraad Mouton, Johannes C. Myburgh, and Marelie H. Davel. «Stride and Translation Invariance in CNNs». In: *Artificial Intelligence Research*. Ed. by Aurona Gerber. Cham: Springer International Publishing, 2020, pp. 267–281. ISBN: 978-3-030-66151-9 (cit. on p. 6).

[15] Yang Liu and Jeffrey Heer. «Somewhere Over the Rainbow: An Empirical Assessment of Quantitative Colormaps». In: *Proceedings of the 2018 CHI Conference on Human Factors in Computing Systems*. CHI '18. Montreal QC, Canada: Association for Computing Machinery, 2018, pp. 1–12. ISBN: 9781450356206. DOI: 10.1145/3173574.3174172. URL: https://doi.org/10.1145/3173574.3174172 (cit. on p. 16).

[16] *Introduction to the viridis color maps*. URL: https://cran.r-project.org/web/packages/viridis/vignettes/intro-to-viridis.html#introduction (visited on 06/02/2022) (cit. on p. 16).

[17] *LeNet-5 from Scratch | harrypnh*. URL: https://github.com/harrypnh/lenet5-from-scratch (cit. on p. 19).