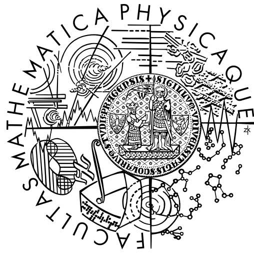

FACULTY OF MATHEMATICS AND PHYSICS
Charles University

BACHELOR THESIS

Martina Ciklamíniová

Neural Network Visualization

Department of Theoretical Computer Science and Mathematical Logic

Supervisor of the bachelor thesis: Mgr. Martin Pilát, Ph.D.
Study programme: Computer Science (B1801)
Study branch: General Computer Science

Prague 2022

I declare that I carried out this bachelor thesis independently, and only with the cited sources, literature and other professional sources. It has not been used to obtain another or the same degree.

I understand that my work relates to the rights and obligations under the Act No. 121/2000 Sb., the Copyright Act, as amended, in particular the fact that the Charles University has the right to conclude a license agreement on the use of this work as a school work pursuant to Section 60 subsection 1 of the Copyright Act.

In .... date ....

Author's signature

I would like to sincerely thank my supervisor, Mgr. Martin Pilát, Ph.D. for his patience, optimism, assistance, and guidance. Further, special thank you goes to my family and friends for their endless support and encouragement.

Title: Neural Network Visualization

Author: Martina Ciklamíniová

Department: Department of Theoretical Computer Science and Mathematical Logic

Supervisor: Mgr. Martin Pilát, Ph.D., Department of Theoretical Computer Science and Mathematical Logic

Abstract: In a number of fields, neural networks can achieve state-of-the-art performance, but understanding how and why they arrive with a solution is still unclear. Particularly for processing visual data, Convolutional Neural Networks (CNNs) have demonstrated great success. CNNs are structured to function with a two-dimensional image input, and as a result, they maintain the spatial relationships for what the model learns. The main goal of this thesis is to develop a tool for visualization of a CNN's inner activations. We apply different techniques to compare various inputs to provide an educational analysis of CNN's behavior.

Keywords: Visualization, Convolutional neural network, Artificial intelligence

Contents

Introduction **3**

1 Convolutional Neural Networks **4**
1.1 CNN structure 4
1.1.1 Convolutional layers 5
1.1.2 Pooling layers 7
1.1.3 Fully-connected layers 7
1.2 CNN Models 8
1.2.1 Simple MNIST model 9
1.2.2 VGG-16 9
1.2.3 ResNet50 10

2 Adversarial Examples **13**
2.1 Types of Adversarial Attacks 13
2.2 Adversarial Examples Generation 15
2.2.1 Fast Gradient Sign Method 15
2.2.2 Projected Gradient method 16

3 Visualization techniques **17**
3.1 Activation Based Visualizations 17
3.2 Image-Specific Class Saliency 18
3.3 Grad-CAM 19

4 Implemantation **21**
4.1 Language and Libraries 21
4.2 Project Structure and Working 21
4.2.1 Utils 21
4.2.2 Pages 25
4.2.3 Visualizer 30
4.3 Adding more visualizations 31
4.3.1 New Visualization method 31
4.3.2 New Page 31

5 User Guide **32**
5.1 Installation 32
5.2 Main Menu 32
5.3 Average Activation 33
5.4 Adversarial Example 34
5.5 Grad-CAM 36
5.6 Saliency Map 37

6 Experiments **38**
6.1 Average Activation Visualization Results 38
6.1.1 Filters Across the Network 38
6.1.2 Prediction 39
6.2 Fooling CNN 40

6.2.1 Partially Covered Input 40
6.2.2 Adversarial Example 41

Conclusion **43**

Bibliography **44**

A Attachments **47**
A.1 Contents of electronic attachment 47

Introduction

It is undoubtful that Artificial Intelligence has become a significant part of our daily lives. Many of its fields, like Machine Learning with Neural Networks, are thriving. Their popularity is at an all-time high, and with the rising power of the hardware, so is their performance.

Neural networks can achieve state-of-the-art performance in a wide range of fields, but understanding how and why they come up with a solution is still unclear. However, with application in high-risk fields like medicine or autonomous vehicles, understanding their reasoning has become very important.

In particular, Convolutional Neural Networks (CNNs) have demonstrated great success in processing visual data. CNNs are structured to function with a two-dimensional image input, and as a result, they maintain the spatial relationships for what the model learns. However, even the most successful network can be fooled with not too much effort required with so-called adversarial examples. These perturbations offer an opportunity to understand the work of CNNs better.

Considering this, the primary goal of this thesis is to create a tool that allows the visualization of a CNN's inner activations. There are many other approaches to visualizing CNNs, and we should not limit ourselves to only a single method. The aim is to implement an interface for a comprehensive educational analysis of CNN's behavior in various scenarios. In summary, these are the features that our implementation should support:

- *User-friendly environment* - the user interface must be simple enough for the tool to be straightforward to use.
- *Activation-based visualization* - the tool should provide a visualization focusing on the network's activations to examine where the CNN decides for a prediction.
- *Variety of visualizations* - the tool's purpose is to provide a broad analysis of a model. Therefore different explanations of CNN's prediction are needed.
- *Not model-specific* - various pre-trained models need to be provided. Also, the user should be able to load their model of choice.
- *Adversarial examples* - the user should be able to inspect the effects of adversarial examples.

This thesis consists of six separate chapters. Chapter 1 introduces CNNs, their structure, and the models used in the visualization tool. In chapter 2 we discuss Adversarial examples and some methods of generating them. In chapter 3 we discuss different approaches to visualize CNNs. Chapter 4 focuses on the implementation of the tool, and chapter 5 introduces the user to the use of the application. The final chapter 6 discusses our outputs of performed experiments.

1. Convolutional Neural Networks

Convolutional Neural Networks (CNNs), as a type of Artificial Neural Network (ANN), is a computing processing system inspired by how organic nervous systems, like the human brain, function. The primary building blocks are referred to as neurons, a high number of interconnected computational nodes divided into layers (input, hidden, output) that self-optimize through learning. After loading the input to the input layer, it would distribute the data to the hidden layers. The hidden layers carry out the learning process. They make decisions from the previous layer and weigh up how a stochastic change within itself detriments or improves the final output [1].

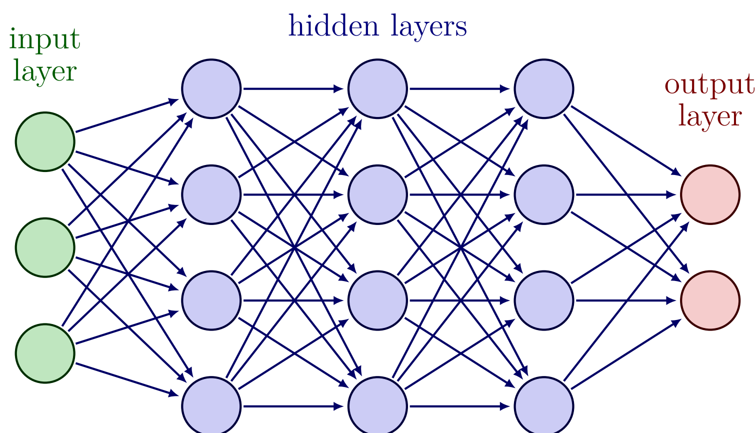

Figure 1.1: Simple ANN architecture [2].

CNNs are different from other ANNs in that they focus on a particular type of input (a multidimensional array) rather than the entirety of the problem domain, thus making it possible to reduce the number of parameters and solve complex tasks with large models [3].

In this chapter, we take a look at CNN's key elements and how they work. We also introduce some real examples of CNN that we use in the visualization tool.

1.1 CNN structure

The architecture of a CNN can vary. However, they share a lot of fundamental elements, namely convolutional, pooling, and fully-connected layers [4]. The difference between a fully connected and convolutional layer is that neurons in a convolutional layer are connected to a local region in the layer before it. In contrast, each neuron in a fully-connected layer has connections to all activations in the previous layer.

1.1.1 Convolutional layers

As the name suggests, a convolutional layer is the main part of CNN's computations. The convolutional layer learns to represent the features of the inputs [4]. It comprises a set of filters (or convolution kernels), convolving with a given input x to generate an output feature map f:

$$f(i,j) = (x * k)(i,j) = \sum_{m} \sum_{n} x(m,n)k(i-m, j-n), \tag{1.1}$$

where k is a two-dimensional filter [5]. To convolve means multiplying the filter with a local region of neurons in the previous layer, and the sum of this product is a neuron of a feature map. The filter slides along the previous layer to obtain a full feature map as we can see in figure 1.2. We call the local region from the previous layer the neuron's receptive field.

The convolution filters do exactly what the name hints at. They filter out specific features of the image. The filters in the first convolutional layers are designed to detect low-level features such as edges and curves, while the filters in higher layers learn to encode more abstract features. The values in the filter are referred to as the weights. The weights are randomly initialized, and the network learns their values through back-propagation. Back-propagation is an algorithm for training ANNs and computes the gradient of the loss function with respect to the parameters of the network. The errors are recursively sent backward through the multi-layered network [6]..

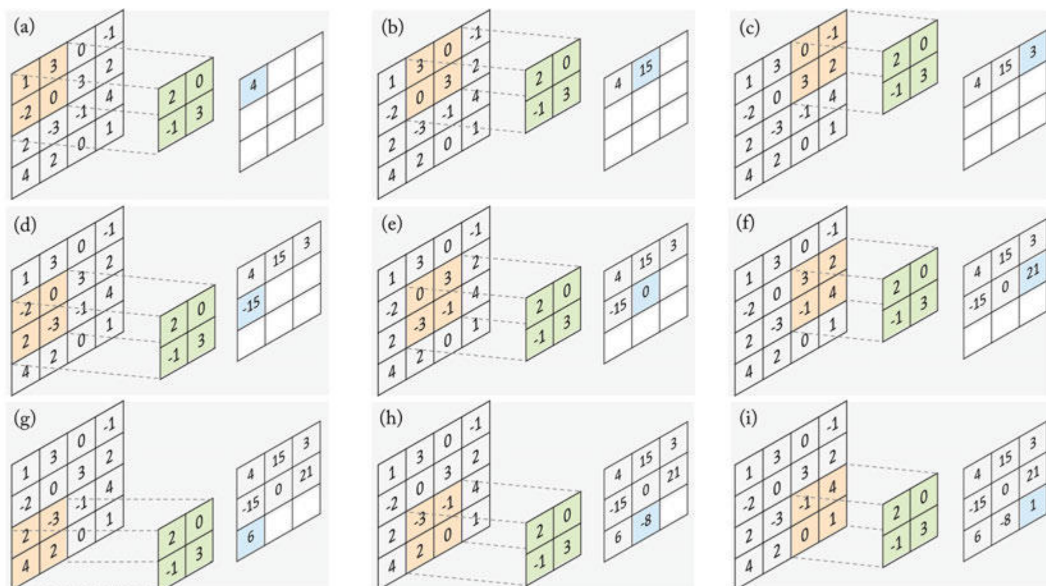

Figure 1.2: Example of a convolutional operation. The filter is sliding across the whole previous layer. The filter is highlighted in green, and orange is the receptive field of the blue destination neuron in the convolved image [6].

The convolved image is usually put through a non-linear activation function to squeeze its values into a narrower range. The non-linearity in CNN is essential to detect non-linear features. In the absence of non-linearities, the network is equivalent to a linear mapping from the input to the output domain. A non-linear function can also be perceived as a selecting mechanism, which decides whether

a neuron will fire or not given all of its input [6]. Typical activation functions are sigmoid, hyperbolic tangent function, and ReLU (the rectified linear unit).

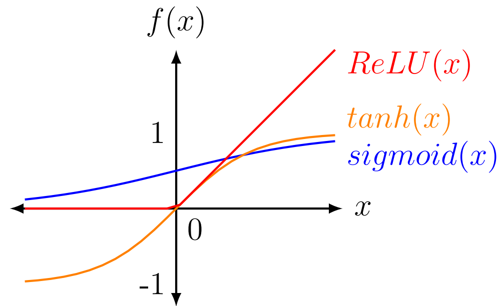

Figure 1.3: Some of the most common activation functions: Sigmoid, Tanh, ReLU.

- **Sigmoid:** as we can see in figure 1.3 the sigmoid activation function $\sigma(x)$ transforms the input in the range $[0, 1]$. It is defined as $f(x) = \dfrac{1}{1+e^{-x}}$ [7].
- **Tanh:** in contrast with the sigmoid is symmetric around 0 and its values lie in the range $[-1, 1]$. It can be defined as $f(x) = 2\sigma(2x) - 1 = \dfrac{e^x - e^{-x}}{e^x + e^{-x}}$ [7].
- **ReLU:** is defined as $f(x) = \max(0, 1)$ [7]. The ReLU function is very commonly used, and many variations exist to overcome its defects.

The non-linear activation functions are differentiable so that the weight of the network can be updated during back-propagation.

Convolutional layers can optimize their output and thus reduce the complexity of the model through hyperparameters. Based on the application, they are design choices of the network architecture set by the user. These parameters are:

- **Number of filters:** by adjusting the number of neurons in each layer concerning the same area of the input, the depth of the output volume generated by the convolutional layers may be manually set. Reducing this hyperparameter can drastically lower the total number of neurons in the network, with the expense that it can also significantly lower the model's capacity for pattern recognition [1].
- **Stride:** a step along the horizontal or vertical position the filter takes in order to calculate each value of the output feature map. The bigger the stride, the smaller the output feature map. This dimension reduction is referred to as the sub-sampling operation [6]. For example, with stride 2 in figure 1.2 the convolution operation would omit steps *(b)*, *(d-f)*, and *(h)*, and that would result in the 4×4 feature map.
- **Zero-padding:** border of zeroes added to the input to allow more control over the output's dimension. That is done by increasing the input size to achieve desired dimensions of the output feature map. The padding must involve at least one original input value in the convolution operation [6].

The spatial dimensionality of the output feature map after tweaking the hyperparameters can be calculated using the formula:

$$\frac{(V-F)+2Z}{S+1}.$$

Where V represents the input size, F represents the filter size, Z is the amount of zero padding set, and S refers to the stride [1]. The stride has to be set, so the formula gives an integer.

1.1.2 Pooling layers

Stacking more convolutional layers can increase the depth of the output. Using pooling layers reduces the complexity of a CNN. The pooling layer performs downsampling along the spatial dimensionality of the feature maps [4]. Like a convolution filter, pooling is done on a block of input values across the feature map. Therefore the stride and size of the pooled block are set. A pooling function replaces the output feature map at a certain location with a summary statistic of the nearby block outputs [5]. The Commonly used pooling functions are max pooling and average pooling. In max pooling, the pooling operator maps the block to its maximum value, while the average pooling maps the block to its average value [1].

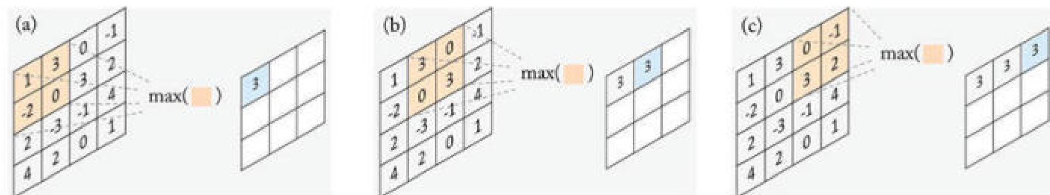

Figure 1.4: Example of max pooling with stride 1 over block of size 2×2 [6].

The goal of pooling is to transform the feature representation into a new one that preserves important information while discarding irrelevant detail [8]. Pooling layers usually lie between convolutional layers.

1.1.3 Fully-connected layers

Each neuron in a fully-connected layer is connected to all the neurons from both the previous and the next layer, similarly to traditional ANN. A fully connected layer's main disadvantage is that it has many parameters that need complicated calculations in training samples. They are usually used towards the end of the model. Its operation can be described as:

$$y = f(Wx + b)$$

where x and y are the vectors of input and output activations, respectively, W denotes the matrix containing the weights of the connections between the layer units, b represents the bias term vector, and $f(\cdot)$ is element-wise nonlinear function [6, 3].

The last fully connected layer, also called the output layer, contains class scores in the case of an image classification problem [9]. For classification tasks,

the softmax function is commonly used. It is also commonly known as a normalized exponential function because it normalizes the output of a network to a probability distribution of the output classes. The optimum parameters for a specific task can be obtained by minimizing an appropriate loss function [4].

1.2 CNN Models

Convolutional neural networks have been proven to be an effective tool for image classification [10]. In image classification, the model tries to categorize the image into one of the classes based on the main object of the image. Two of the most popular classification datasets are MNIST and ImageNet.

The MNIST (Modified National Institute of Standards and Technology) database is a collection of handwritten digits. It is a subset of a more extensive set available from NIST, hence modified NIST. There are 60,000 training and 10,000 test examples drawn from the same distribution. The original black and white (bi-level) images from NIST were size normalized, preserving their aspect ratio. As a result of the anti-aliasing technique, the resulting images contain grey levels. The images were centered in a fixed-size image where the center of gravity of the intensity lies at the center of the image with 28×28 pixels [11, 12].

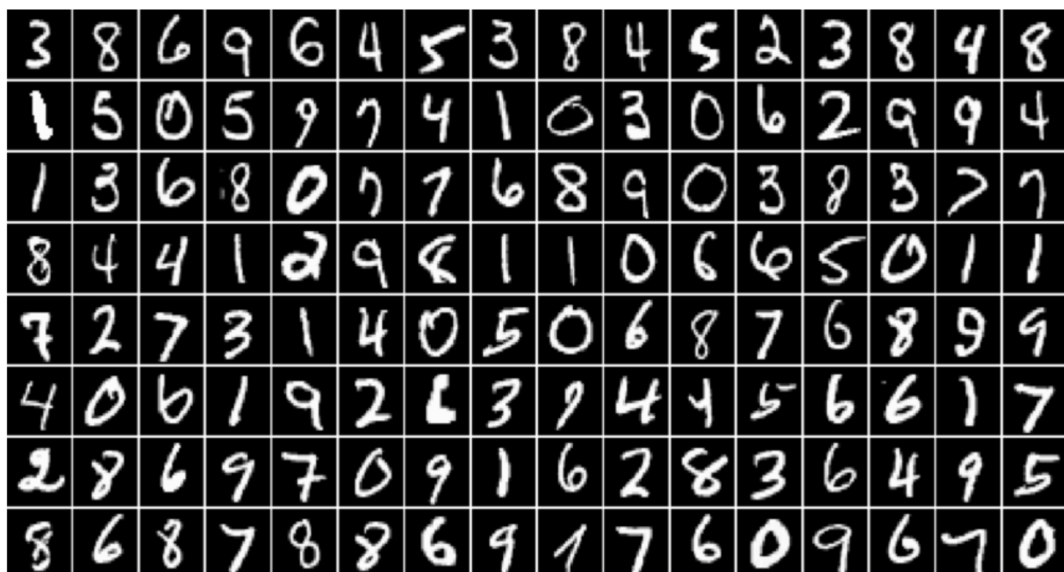

Figure 1.5: Samples from the MNIST dataset [13].

ImageNet is a large-scale ontology of images organized according to the WordNet hierarchy of nouns. Each meaningful concept in WordNet is called a ”synonym set” or ”synset.” Each synset is a node in the ImageNet hierarchy, depicted by hundreds and thousands of images. There are over 15 million hand-labeled, high-resolution images in roughly 22,000 categories. Images of each concept are quality-controlled and human-annotated. [14, 9]. The ImageNet Large Scale Visual Recognition Challenge (ILSVRC) has been running annually since 2010. It is an object recognition contest with 1000 classes. ILSVRC uses a subset of ImageNet images for training the algorithms and some of ImageNet’s image collection protocols for annotating additional images for testing the algorithms [15].

In the following sections, we look at the models used in the visualization tool.

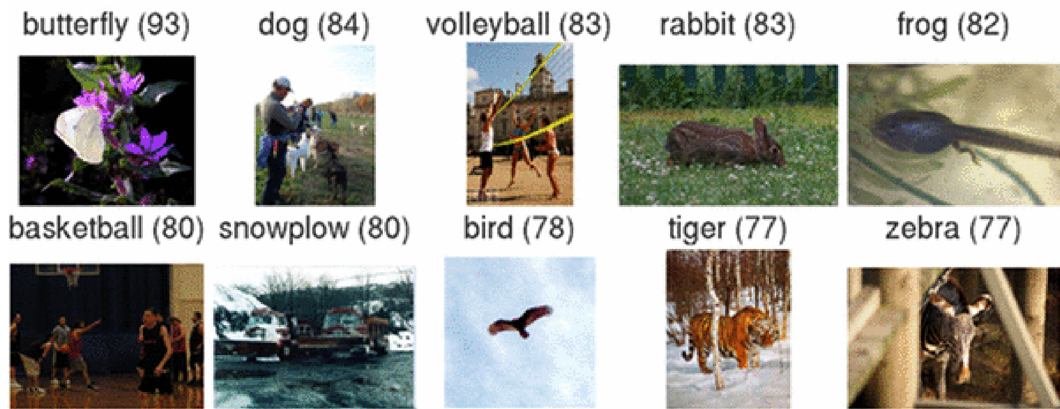

Figure 1.6: Samples from the ImageNet dataset used in ILSVRC2012-2014 [15].

1.2.1 Simple MNIST model

MNIST is a relatively simple database, so we do not need a highly complex model. In the visualization tool, we use a straightforward model with only six hidden layers, as shown in figure 1.7. Namely, it consists of a convolution layer, followed sub-sampling max-pooling layer, then two more convolution layers again followed by a sub-sampling, then a flatten layer, and ends with two fully connected (dense) layers. The flatten layer reduces the input data into a single dimension instead of 2 dimensions. The input to the model is a fixed-size 28×28 greyscale image, as is standard in the MNIST dataset. We used ReLU as the activation function in the weight layers. The output layer uses softmax as its activation. The model was trained for ten epochs on the MNIST train dataset and reached 98.95% accuracy on the test collection.

1.2.2 VGG-16

In the 2014 ILSVRC, the VGG architecture was introduced to the world. Despite being the challenge's runner-up, the VGG became one of the most popular CNN models. Its popularity is due to its simple and elegant architecture with only 16–19 (depending on the configuration) weight layers. The VGGnet architecture strictly uses 3×3 convolution filters. Stacking two 3×3 convolution layers is equivalent to one 5×5 convolution layer, and stacking three 3×3 convolution filters replaces a 7×7 convolution layer. Using smaller filters leads to a relatively reduced number of parameters and efficient training and testing because it computes faster than a large convolution filter. Most importantly, with smaller filters, one can stack more layers resulting in deeper networks, which is the central idea of this architecture [16, 6, 9].

The configuration D commonly referred to as VGG-16, is one the most successful and is the one we use in the visualization tool. It contains 16 weight layers. The default input size of an image for this model is $224 \times 224 \times 3$. The image is passed through stacks of convolution layers with a stride 1 and padding 1, followed by a max pooling layer with 2×2 window with a stride of 2. There are two stacks of two and three stacks of three convolutional layers. Towards the end, there are three fully-connected layers. All hidden layers use the ReLU activation function, while the final layer is a softmax classification layer [6, 9].

```
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 conv2d (Conv2D)             (None, 26, 26, 32)        320

 max_pooling2d (MaxPooling2D)  (None, 13, 13, 32)      0

 conv2d_1 (Conv2D)           (None, 11, 11, 64)        18496

 conv2d_2 (Conv2D)           (None, 9, 9, 64)          36928

 max_pooling2d_1 (MaxPooling2D)  (None, 4, 4, 64)      0

 flatten (Flatten)           (None, 1024)              0

 dense (Dense)               (None, 100)               102500

 dense_1 (Dense)             (None, 10)                1010

=================================================================
Total params: 159,254
Trainable params: 159,254
Non-trainable params: 0
_________________________________________________________________
```

Figure 1.7: Model summary of the simple MNIST model used in the tool.

1.2.3 ResNet50

The deep CNNs had been proven to be successful, but adding layers increases performance only up to a certain depth, then it rapidly decreases. Unexpectedly, such degradation is not caused by overfitting but rather a high training error. The Residual Network (ResNet) battles this issue with the identity skip connections in the residual blocks. The ResNet architecture won the ILSVRC 2015 challenge. The ResNet architecture is a stack of residual blocks with different depth variations [18].

The residual block can be formaly denoted as:

$$y = \mathcal{F}(x, \{W_i\}) + x, \tag{1.2}$$

where x and y are the input and output vectors of the layers considered and W_i their weights. The residual block splits the transformation function into an identity and a residual mapping. The function $\mathcal{F}(x, \{W_i\})$ represents the residual mapping and can contain multiple convolutional layers. In figure 1.9, it contains two layers, so $\mathcal{F} = W_2\sigma(W_1x)$ where σ denotes ReLU operation. The original input x is added to the transformation using the "skip identity connection", a direct connection bypassing the transformation layers from the input, performing the identy mapping. The second ReLU nonlinearity is performed after the addition [18, 6].

ResNet has different depth varieties, such as 34, 50, 101, or 152 layers for the ImageNet dataset. For a deeper network (50 and more layers), it uses the bottleneck design to improve efficiency. Each residual function $\mathcal{F}$ uses a stack of three layers with 1, 3, and 1 convolution filters, where the 1 layers reduce and

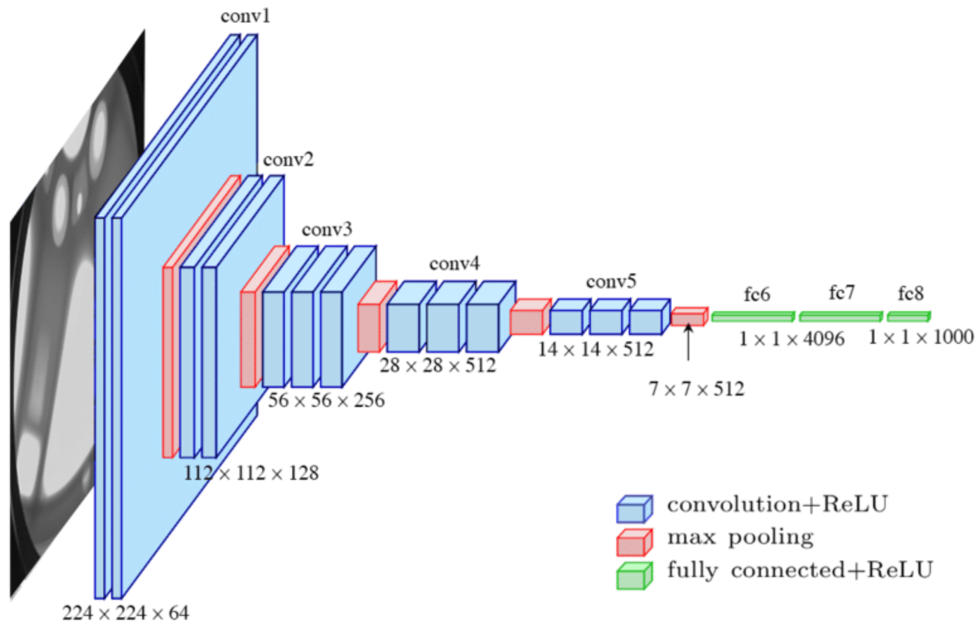

Figure 1.8: The VGG-16 architecture [17].

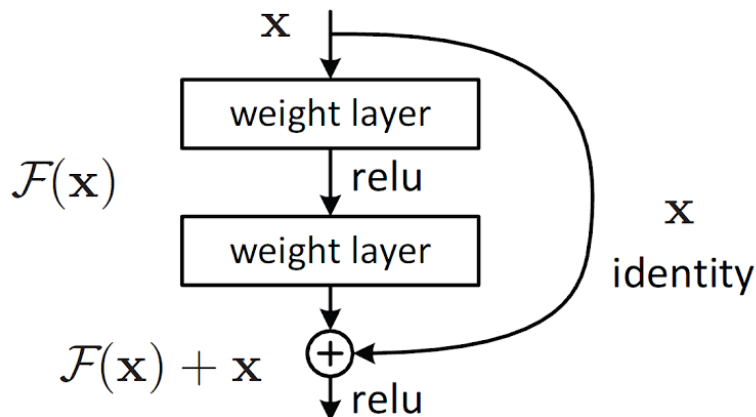

Figure 1.9: Residual block [18].

then increase dimensions, leaving the 3 layer a bottleneck with smaller input and output dimensions [18, 9].

The unreferenced mapping learned in traditional architectures is sometimes far more complex than the residual feature mapping. Therefore in practice, such an architecture achieves stable learning of very deep models.

In the visualization tool, we use the configuration of ResNet with 50 layers (Resnet50). This configuration consists of 16 residual bottleneck blocks stacked on top of each other. After the last block, a global average pooling layer is added. There is only one fully connected layer to classify 1,000 classes with softmax (see figure1.10) [18].

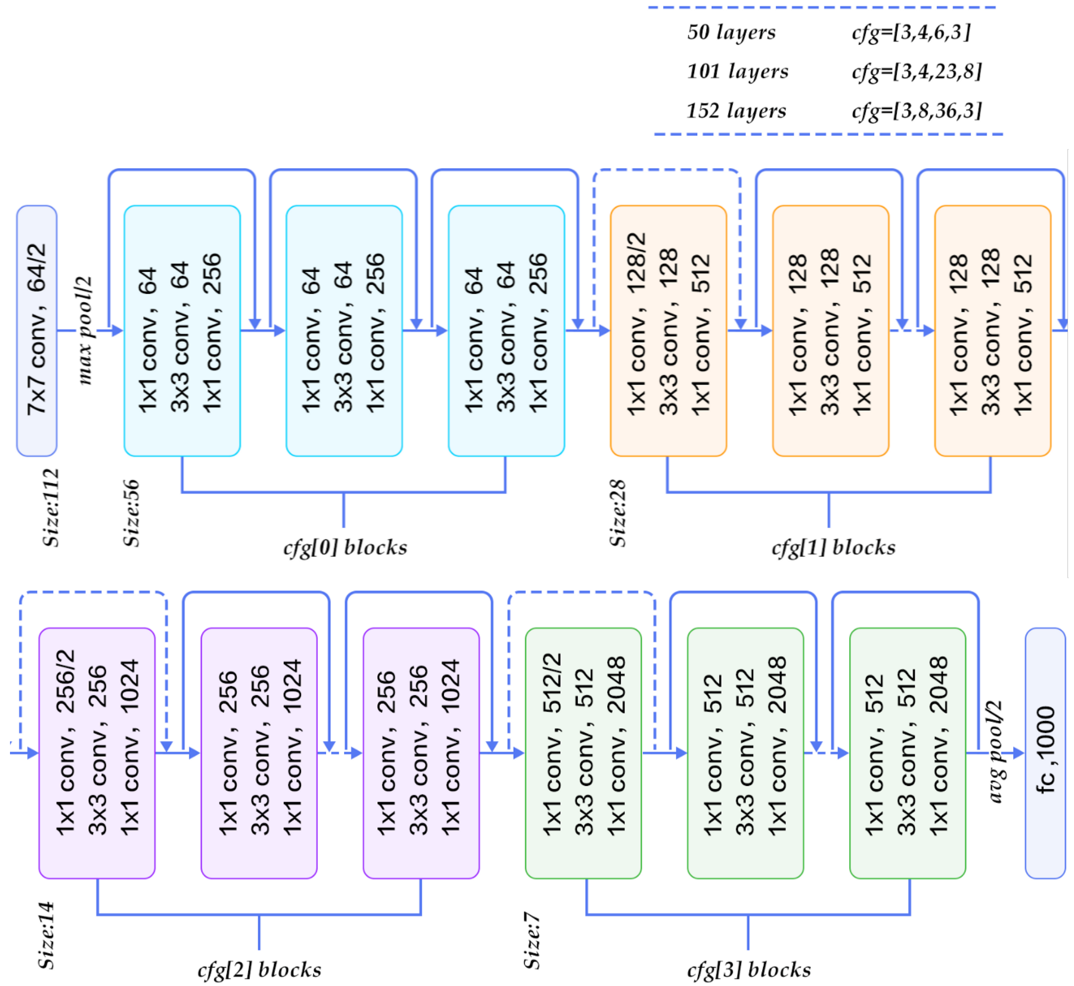

Figure 1.10: ResNet architecture [19]. ResNet50 corresponds to a varient with 50 layers i.e. it contains 3 cfg[0] blocks, 4 cfg[1] blocks, 6 cfg[2] blocks, and 3 cfg[3] blocks. The input comes in the shape $(224 \times 224 \times 4)$.

2. Adversarial Examples

The CNNs are reaching state-of-the-art performance, nearing human abilities in image classification. However, by applying a certain, to human eye hardly noticeable, perturbation we can cause the network to misclassify an image. This property was first explored in the 2014 paper [20], where the perturbed images were labeled ”adversarial examples.” The adversarial example is found by maximizing the network's prediction error. The paper even states that the specific nature of the perturbations is not a random artifact of learning but rather non-intuitive characteristics and intrinsic blind spots of the deep neural networks.

Szegedy et al. [20] pointed out an interesting property of adversarial examples, their generalization of deceptive images across different CNNs. The same noise can fool various models. The 2014 paper [21] argued that linear behavior in high-dimensional spaces is sufficient to cause adversarial examples rather than excessive nonlinearity of the network as was thought previously. CNNs as complex Deep Learning models are not linear mapping of the input to class scores. The convolution of a filter with its input and matrix multiplications, the components that comprise CNN, are linear functions. CNNs are also often intentionally designed to behave in very linear ways, making them easier to optimize.

These results are neither CNN-, image-specific, nor a ”flaw” in Deep Learning. However, the fundamental issue affects a variety of other fields (such as voice recognition systems) and, more significantly, old-fashioned linear classifiers (such as the Softmax classifier).

2.1 Types of Adversarial Attacks

We can categorize the adversarial attacks and their strength considering the attack surface and adversarial goals and capabilities. In this we taxonomize the threat models as seen in [22, 23]

An adversary may try to tamper with the model, the data gathering and processing, or the model's outputs. We identify the threat surface to anticipate where and how an adversary would try to sabotage the system that is under attack.

- **Evasion Attack:** tries to deceive the system by adjusting malicious samples during the testing phase.
- **Poisoning Attack:** tries to sabotage the learning process by introducing thoroughly designed samples into the training data.
- **Exploratory Attack:** tries to learn as much as possible about the underlying system's learning algorithm and training data pattern with black-box access to the model. It does not influence the training dataset.

An attacker's knowledge about the system is referred to as adversarial capabilities. If two adversaries are working on the same attack surface and one is an internal adversary, has access to the model architecture, and the other has access only to the set of images fed to the model during testing time, the former adversary is thought to have far more knowledge and is thus strictly ”stronger.”

- There are three general attack strategies for altering the model during the training phase to influence or corrupt the model directly by altering the training dataset:
  - **Data Injection:** There is no access to the training data or the learning algorithm but can augment new data to the training set (inserting adversarial samples into the training dataset).
  - **Data Modification:** There is no access to the learning algorithm but has full access to the training data (modifying the data before training).
  - **Logic Corruption:** The adversary can tamper with the learning algorithm (controlling the model by altering the learning logic).
- Testing phase attacks can be generally classified into either White-Box or Black-Box attacks:
  - **White-Box Attacks:** A white-box attack involves an adversary with full knowledge of the classification model. The attacker has access to the training data distribution and algorithm employed during training (such as gradient-descent optimization). Additionally, the fully trained model architecture's parameters are known. All available information is utilized to find the vulnerability of the feature space, and by altering an input, the model is exploited. The attack strategy is based on a greedy local search that uses changes in input to approximate the gradient for the current output.
  - **Black-Box Attacks** A black-box attack makes no assumptions regarding the model and analyzes the vulnerability of the model using knowledge of the settings or previous inputs. The adversary does not attempt to discover the parameters of the target model. A black-box adversary's main goal is to use the data distribution to train a local model. The attack strategy is based on the network loss function's gradient for the input.

The following broad categories can be used to classify adversarial goals according to their influence on the integrity of the classifier output:

- **Confidence Reduction:** The adversary seeks to lower the target model's prediction confidence.
- **Misclassification:** The adversary tries to change an input's classification to a different class than it originally was.
- **Targeted Misclassification:** The adversary tries to produce inputs that force the output of the classification model to be a specific target class.
- **Source/Target Misclassification:** The adversary attempts to create inputs that cause the classification to be a particular target class.

2.2 Adversarial Examples Generation

Our visualization tool works with already trained models, so we only focus on evasion attacks during the testing phase. Moreover, since we already have access to the models, we can focus on white-box attacks. In this section, we describe the generation methods used in the tool. Both of the methods described are gradient-based, also called first-order methods.

2.2.1 Fast Gradient Sign Method

The fast gradient method (FGSM), first introduced in [21], was yielded by the view that the linear behavior is behind the existence of adversarial examples. In model training, the goal is to minimize the loss, and the gradient descent technique uses the gradient to update the weights. The FGSM generates the adversarial pattern that maximizes the loss for an input image. Therefore we can take the gradients with respect to the input. The adversarial perturbation η for the input x is described as:

$$\eta = \epsilon \, \text{sign}(\nabla_x J(\theta, x, y)), \tag{2.1}$$

where θ denotes the parameters of a attacked model, y ground-truth label corresponding to x, $J(\theta, x, y)$ the loss function and ϵ is a constant scaler. The idea is to find the direction each pixel in the input contributes to the loss and add perturbation accordingly, large enough for misclassification but small enough, so it stays undetected. Since the model is no longer being trained and the parameters remain constant, computing the gradient is very efficient. The only focus is to deceive a trained model.

The pertubation described in 2.1 is not targeted. If we wanted to target class $\tilde{y}$ we would use $\eta = -\epsilon \, \text{sign}(\nabla_x J(\theta, x, \tilde{y}))$.

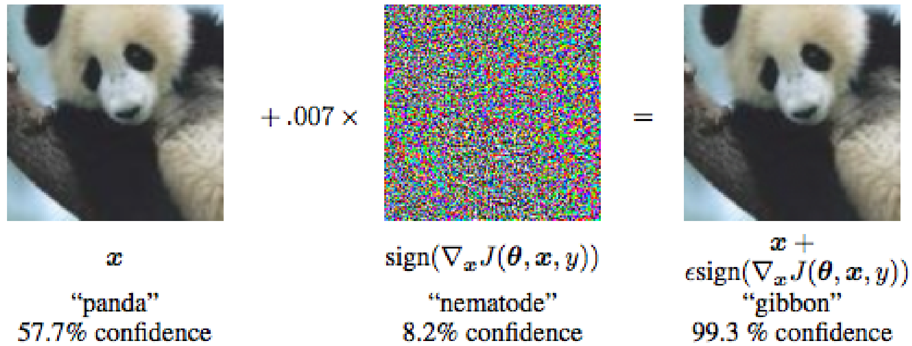

Figure 2.1: Demonstration of the FGSM. Original image x (on the left), labeled as "panda" with 57.7% confidence is transformed into adversarial example (on the right) labeled as "gibbon" with 99.3% confidence. The adversarial pattern (in the middle) was generated using ϵ of .007 [21].

The adversarial example $\tilde{x}$ is computed as the sum of the original image x and adversarial pattern η, formally:

$$\tilde{x} = x + \epsilon \, \text{sign}(\nabla_x J(\theta, x, y)).$$

In the figure 2.1 we can see FGSM generated adversarial example for $\epsilon = 0.007$. Even though FGSM is one of the very first methods for generation of adversarial examples, it is very fast and reliable approach.

2.2.2 Projected Gradient method

We can look at the FGSM attack as a one-step strategy for maximizing the loss. A simple iterative variation of FGSM was introduced in [24] called Basic Iterative Method (BIM). BIM repeatedly runs FGSM while the perturbation is bounded to an ϵ-neighborhood:

$$\tilde{x}_0 = x, \qquad \tilde{x}_{n+1} = \mathrm{Clip}_{x,\epsilon}\{\tilde{x}_n + \alpha\,\mathrm{sign}(\nabla_x J(\tilde{x}_N, y))\}, \tag{2.2}$$

where $\mathrm{Clip}_{x,\epsilon}\{x'\}$ is a function performing per-pixel clipping of x', keeping it in L_∞ ϵ-neighborhood of the original x.

The multi-step alternative is a more powerful attack, and the perturbations are less detectable than the plain one-step variant (see figure 2.2).

Figure 2.2: Comparison of FGSM and basic iterative variation with $\epsilon = 32$ [24].

BIM is equivalent to the projected gradient descent (PGD) on the negative loss function [25]:

$$x_{n+1} = \Pi_{x+\mathcal{S}}(x_n + \alpha\,\mathrm{sign}(\nabla_x J(\theta, x, y))), \tag{2.3}$$

where $\Pi_{x+\mathcal{S}}$ is a projection onto the set of allowed perturbations $\mathcal{S}$ for each data point x (i.e. $\Pi_{x+\mathcal{S}}(z) = \mathrm{Clip}_{x,\epsilon}\{y\}$), and α is the step size. The difference is that PGD initializes to a random perturbation (decided by the L_∞ norm from the $\mathcal{S}$) and does random restarts, while BIM initializes to the original x. Then a series of gradient descent steps decreases the probability of the ground-truth label. During a targeted attack, the gradient descent increases the probability of a target label, and 2.2 is modified as the targeted version of 2.1.

3. Visualization techniques

This chapter looks at different techniques used to visualize CNN models. When working with a CNN model, there are various approaches we can take to understand its working better. The straightforward method is to portray the model's architecture. If we look inside the model, another simple approach is visualizing learned convolution filters to see what properties our model is learning. Other methods work with the model's parameters like its activations and gradient to see what parts of the input are essential for the output. We describe some of the latter approaches in the upcoming sections.

3.1 Activation Based Visualizations

Activation-based techniques interpret the activations of individual neurons or groups of neurons to gain an understanding of what they are learning. When we visualize activations of CNN, we can be input-specific or try to find broad features learned.

Feature visualization is a method for making the learned characteristics explicit. Feature visualization finds the input that maximizes the activation of the unit of a CNN (individual neurons, feature maps, entire layers, or the final class probability), similarly to adversarial examples. Feature visualizations offer a distinct perspective on how CNNs behave and show that CNNs initially learn basic edge and texture detectors before moving up to more complex part and object detectors (see figure 3.1). Many feature visualization images are incomprehensible. They include certain abstract aspects for which we lack words or mental concepts [26].

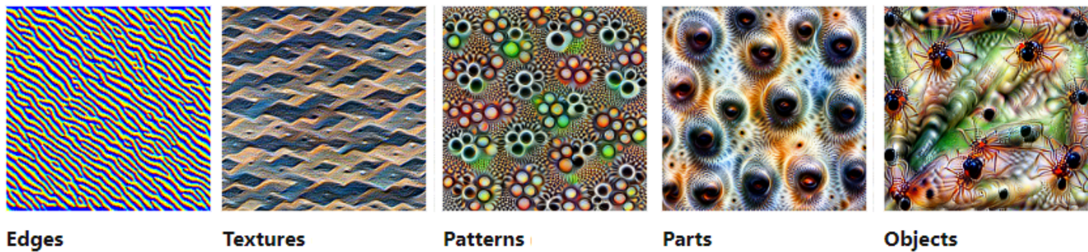

Figure 3.1: Examples of feature visualization. Feature depicted from left: edges, textures, patterns, parts, and objects [27].

In our visualization tool, we implement an input-specific visualization. Our approach focuses on the output feature maps of the convolution and fully connected layers. We omit the outputs of other layers (e.g. pooling layers) because they do not bring any new information to the visualization

The main idea is to compare the feature maps of filters on each layer to see where the different classifications start to differ. In other words, we are trying to find a point in the network where the model decides on a class. There are two modes of this visualization, depending on what we are comparing:

- **Images** - we compare the activations of two input images. This can be

useful in comparing adversarial examples with the original images to see where the network was confused.

- **Classes** - we compare the pixel-wise average activation of two classes (the two top predictions of a set of input images). It is interesting to see how the activations blend for inputs of the same class and how it differs from another.

3.2 Image-Specific Class Saliency

Simonyan et al. introduced in [28] a visualization technique called "Image-Specific Class Saliency." This technique computes a class saliency map specific to a given image based on computing the gradient of the output with respect to the input image. The resulting image highlights the most important regions of the input that contribute the most to the class score, making it a class-specific visualization.

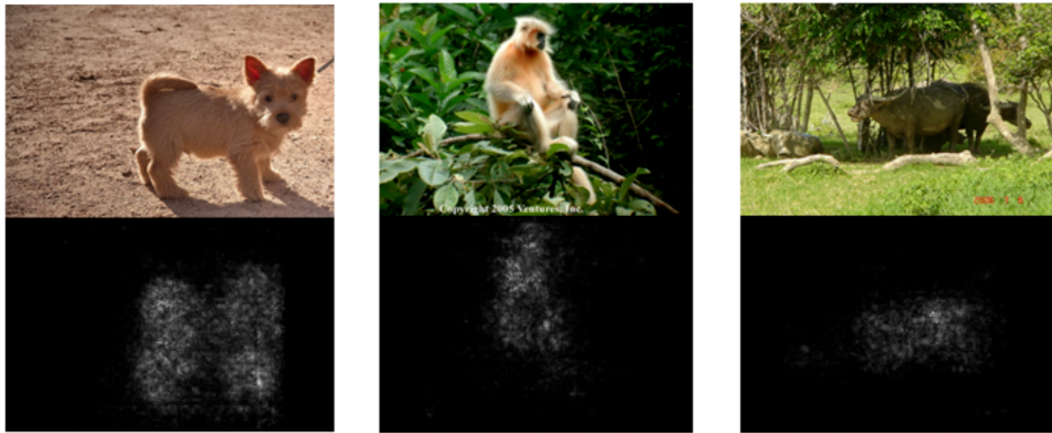

Figure 3.2: Example of the saliency maps on images from the ILSVRC-2013 test set for the highest scoring class. [28].

The method to generate class saliency map $M_c \in \mathbb{R}^{u \times v}$ is described in [28] as follows. For a given input $x \in \mathbb{R}^{u \times v}$, class C, and score function $S(x)$, we want to rank the pixels of x based on the influence on score $S(x)$ to obtain a class saliency map. We compute the gradient of the loss function with respect to the input x for the class C, resulting in a map of negative and positive values with the size of the input features. The score function $S(x)$ is a highly non-linear function of the input. The concept behind employing the gradient is to approximate the score using a first-order Taylor expansion:

$$S(x) \approx w^T x + b, \tag{3.1}$$

where b is the model's bias, and w is the derivative of the score with respect to x:

$$w = \frac{\partial S}{\partial x}. \tag{3.2}$$

We find the derivative w by back-propagation. Let us denote the index $h(i, j, c)$ of the element in w where c is the color channel of the pixel (i, j) in x. We take

the maximum magnitude of w across all color channels to compute the values of the class saliency map M_c for each index (i, j):

$$M_{ij} = \max_{c} \left| w_{h(i,j,c)} \right| . \tag{3.3}$$

The magnitude of the derivative reveals the pixels with the most impact on the class score. Since only one back-propagation step is needed, computing the image-specific saliency map for a single class is very quick.

3.3 Grad-CAM

Gradient-weighted Class Activation Mapping (Grad-CAM) is a technique for producing visual explanations for the decisions of CNNs and was proposed by Selvaraju et al. in [29]. Grad-CAM uses the gradients flowing into the final convolutional layer to produce a coarse localization map highlighting the regions of important neurons in the image for a prediction. Grad-CAM is a class-discriminative visualization. It is a generalization of a technique called Class Activation Mapping (CAM). CAM interprets CNNs without any fully-connected layers. Grad-CAM applies to a broader range of CNN models.

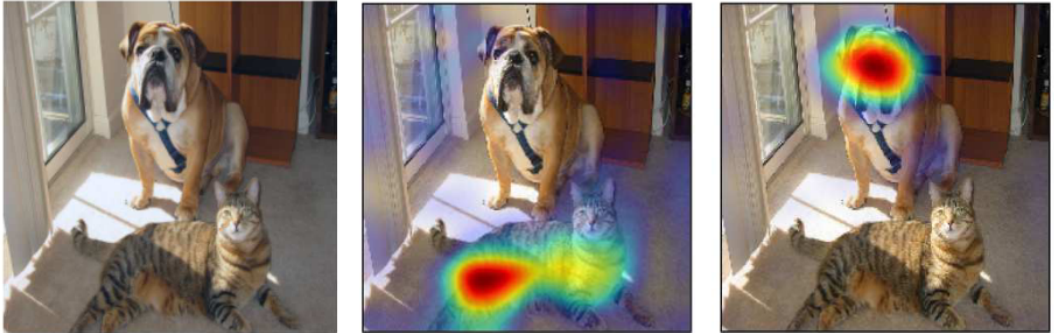

Figure 3.3: Example of the Grad-CAM for the 'tiger cat' class and 'boxer' class. Left: Original image with a cat and a dog. Middle: Grad-CAM 'tiger cat'. Right: Grad-CAM 'boxer' [29].

As is described in [29], to obtain the class-discriminative localization map Grad-CAM $L_c \in \mathbb{R}^{u \times v}$ for an input $x \in \mathbb{R}^{u \times v}$ and a class c, we first compute the gradient $\frac{\partial y_c}{\partial A^k}$ of the score y_c for class c (the activation of the neurons before the softmax layer) with respect to feature maps A^k of a convolutional layer. To obtain the importance of the neuron α_c^k, we apply global-average-pooling to the gradients to weight each feature map neuron by the gradient:

$$\alpha_{c,k} = \overbrace{\frac{1}{Z} \sum_{i} \sum_{j}}^{\text{global average pooling}} \underbrace{\frac{\partial y_c}{\partial A_{ij}^k}}_{gradients} . \tag{3.4}$$

The weight α_c^k reflects the significance of feature map k for a target class c and represents a partial linearization of the CNN downstream from A.

Lastly to obtain L_c, we apply ReLU after a weighted combination of forward activation maps:

$$L_c = \text{ReLU} \underbrace{\left(\sum_k \alpha_c^k A^k \right)}_{\text{linear combination}} . \tag{3.5}$$

We are only interested in features that positively impact the class of interest, so we apply ReLU to the linear combination of maps. Without using ReLU here, localization maps occasionally emphasize classes other than the one required because ReLu omits the negative pixels, which are probably part of different picture types.

4. Implemantation

The main goal of this thesis was to implement a visualization tool for better understanding CNNs. The resulting application is called *CNN Visualizer* and incorporates the visualization techniques described in chapter 3.

In this chapter, we describe the implementation of the application: *CNN Visualizer*. In attachment A.1, a directory tree of all the mentioned files is provided.

4.1 Language and Libraries

Before we get to the project details, we need to state the basics. The program is implemented in the programming language Python 3.8. It is a widely used language, especially in machine learning. Therefore there are a lot of useful libraries. In the visualization app, besides the standard libraries, we also use TensorFlow, Keras, NumPy, Opencv, Pillow, Matplotlib, and Cleverhans.

We use TensorFlow and Keras for working with models. TensorFlow [30] is an open-source platform for machine learning, and Keras [31] is a Python deep learning API that runs on top of TensorFlow. They contain a wide range of accessible utilities, and both belong among the most popular software used in machine learning.

For data processing, we utilize NumPy and also TensorFlow. NumPy [32] is open-source software that adds support for multi-dimensional arrays and matrices and a vast number of high-level mathematical functions operating on these arrays.

We do not have high requirements for the graphical interface. Therefore we use the Tkinter, the standard Python GUI toolkit. Pillow [33], a Python Imaging Library (PIL), is used for manipulating and displaying images. Some visualizations use OpenCV [34] for image processing, and Matplotlib [35] for plotting graphs.

We also use Cleverhans [36], which provides a standardized reference implementation of adversarial attacks.

The specific versions of all the packages can be found in the *requirements.txt* in the electronic attachment to this thesis.

4.2 Project Structure and Working

In this section, we describe the overall structure and implementation of the *CNN Visualizer*. The project can be divided into a computational and a visual part implemented in the python files `utils` and `pages` respectively. Then the python file `Visualizer` ties the application together.

4.2.1 Utils

The `utils` python file contains a `Tools` class. As we mentioned, it serves as the computing engine. Therefore the `Tools` class contains the function generating the visualizations. You can find the overview of the class methods in the figure 4.1.

utils.Tools

__init__(self)	find_class(self, preds, wanted, n=1)
set_data_folder(self, path)	find_frequent_classes(self, classes, n=2)
mnist_dataset(self, n=1)	new_shape(self, n)
load_image(self, path)	scale(self, data)
default_dataset(self, n)	average_activation_difference(self, data, classes=True, ord=2)
choose_dataset(self, n)	vis_aad(self, data=None, n=20, classes=True, ord=2, load=None)
custom_load(self, paths)	adversarial_example(self, image, method, target, eps, norm_ord)
set_dataset(self, x)	vis_ae(self, load=None, method="FGSM", epsilon=0.1, target='', ord=np.inf)
load_model(self, path)	find_top_n_pred(self, prediction, n=3)
mnist_model(self)	find_last_conv(self)
resnet_model(self)	compute_heatmap(self, img_array, pred_indces=None)
create_model(self)	vis_gc(self, data=None, load=None, class_indices=[], n=5)
check_model(self)	vis_sm(self, data=None, load=None)

Figure 4.1: The `Tools` class and its methods.

Attributes and Methods

These are the attributes of the `Tools` class:

- **model** - all computations share this pre-trained model
- **input_shape** - specification of the input shape to be compatible with the model
- **dataset** - what kind of data are we dealing with
- **data_folder** - a path to a directory where to find the data while loading defaultly

They are initialized when the class is constructed, but the user can update them.

The class `Tools` contains all logic behind the *CNN Visualizer* app. Therefore it consists of a lot of functions. The following methods are used for acquiring data:

- **set_data_folder(path)** - sets the *data_folder* parameter to *pathit*
- **set_dataset(x)** - sets the *dataset* parameter to x
- **mnist_dataset(n)** - loads and returns n random images from the MNIST dataset
- **load_image(path)** - loads, preprocesses and returns image from *path*
- **default_dataset(n)** - loads and returns dataset of n images from *the data_folder* directory
- **choose_dataset(n)** - calls either *default_dataset(n)* or *mnist_dataset(n)*, according to the current *model*, and returns the output

- **custom_load(paths)** - returns a dataset of loaded images from *paths*

These methods are implemented for data handling:

- **find_class(preds, wanted, n=1)** - for models VGG16 and ResNet50 returns the names and confidence of top n predicted classes, otherwise the class number and its confidence

- **find_frequent_classes(classes, n=2)** - returns n most frequent classes from the list *classes*

- **new_shape(n)** - returns (x, y) for n, such that x is almost square root of n, used to reshape the one-dimensional output of a fully-connected layer

- **scale(data)** - returns scaled *data* to the values 0-255

- **find_top_n_preds(prediction, n=3)** - returns indices of the top n predicted classes

- **find_last_conv()** - returns the last convolutional layer of the model

And for acquiring the model, we use these methods:

- **load_model(path)** - loads model from *path* and sets *model* paramer

- **mnist_model()** - sets *model* parameter to an instance of the ResNet50 model

- **resnet_model()** - sets *model* parameter to an instance of the MNIST model

- **create_model()** - sets *model* parameter to a default model, the VGG16

- **check_model()** - checks if the *model* is set, if not sets it by call suitable method

Since the methods generating visualizations are the most important and complicated, we describe them individually in the upcoming sections.

Generating Average Activation

As stated in section 3.1, this approach focuses on the output feature maps of the convolution and fully connected layers. To obtain average activation for each filter on each layer we call function **vis_aa(data, n, classes, ord, load)**, with the parameters and their default values:

- **data -** images as arrays used in the computation. If None, they need to be loaded. Default: None.

- **n -** defines how many images are to be loaded. Default: 20.

- **classes -** determines the visualization mode, whether we are comparing classes or images. Default: True.

- **ord -** the order of the norm used in the computation. Default: 2.

- **load -** list of paths to the images we want to load and use in visualization. Default: None.

This method aims to ensure the model is set and the desired data is obtained and pre-processed. Then it calls **average_activation_difference(data, classes, ord)** method to perform the computation and at last return its output.

The **average_activation_difference(data, classes, ord)** function parses through every output feature map of each filter in all convolutional and fully-connected layers. The data are divided into two groups according to the visualization mode. Then the averages are computed respectively, and so is their difference. We calculate the norm of each average difference activation for ordering the filters on each layer. Then if the current feature map is the output of a fully-connected layer, we reshape the one-dimensional activations as close to a square as possible for comprehensive visualization. Finally, we uniformly scale the activations and match them with appropriate descriptions. Then the average activations for each filter and layer are returned.

Generating Grad-CAM

To compute gradient-weighted class activation map we call the **vis_gc(data, load, class_indices, n)** function, with the parameters and their default values:

- **data -** images as arrays used in the computation. If None, they need to be loaded. Default: None.
- **load -** list of paths to the images we want to load and use in visualization. Default: None.
- **class_indices -** list of classes we want to see their heatmap. Default: [].
- **n -** number of top predicted classes for the heatmap. Default: 5.

This method ensures the model is set, and the desired image is obtained and pre-processed. It also assigns relevant descriptions to the predicted classes and the original image. Then to perform the computation we call the function **compute_heatmap(img_array, pred_indces)**, and at last return its output.

The **compute_heatmap(img_array, pred_indces)** method, with respect to the activations of the final convolutional layer, calculates the gradient of the desired predicted class for our input image. Then takes the mean intensity of the gradient of each desired output neuron over a specific feature map channel. To generate the heatmap class activation, we multiply each channel in the feature map array by "how significant this channel is" regarding the top predicted class, then add up all the channels. Finally, we add scaled heatmaps to the relevant dictionaries in *pred_indces* and return them.

Generating Class Saliency Map

To compute the class saliency map we call the **vis_sm(data, load)** function, with the parameters *data* and *load*, which are the same as in the previous two visualizations. This method ensures the model is set and the desired image is obtained and pre-processed. By computing the gradient with respect to the top

class score, we can identify the most contributing pixels in the image. We take the maximum absolute gradient values along each RGB channel to obtain the saliency map. At last, we return the original image, normalized saliency map, and relevant description.

Generating Adversarial Examples

To generate an adversarial example, we call the function **vis_ae(load, method, epsilon, targe, ord)**, with the parameters and their default values:

- **load -** a path to the image we want to load and use in visualization. Default: None.
- **method -** a method to be used for finding an adversarial pattern. Default: "FGSM."
- **epsilon -** a constant used in finding the adversarial pattern. Default: 0.1.
- **target -** a class targeted with the adversarial attack or None. Default: None.
- **ord -** order of the norm used. Default: infinity.

As with the Grad-CAM, this method ensures the model is set, and the desired image is obtained and pre-processed. It also assigns relevant descriptions to the original image. Then the computation is done by the **adversarial_example(image, method, target, epsilon, ord)** function.

In the method **adversarial_example(image, method, target, epsilon, ord)**, we use the Cleverhans functions to generate adversarial examples with the FGSM and PGD techniques. We also generate random white noise to see what effect, if any, it has on the classification of the image. Then we return the adversarial pattern and perturbated image together with an appropriate description.

4.2.2 Pages

The `pages` python file holds classes representing each visualization method. As we have stated earlier, it is in charge of the graphical side of the project.

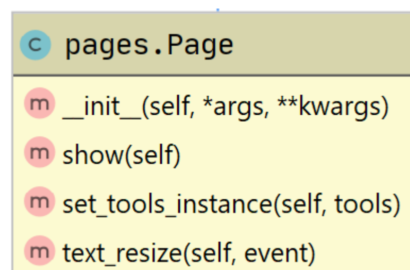

Figure 4.2: The `Page` class and its methods.

The `Page` class is derived from Tkinter Frame, so it groups widgets forming a scene in the application window. The `Page` class holds the fonts used. The methods of `Page` class (see figure 4.2) consists of:

- **show()** - displays the given `Page` instance in the window application.
- **set_tools_instance(tools)** - sets the `Tools` instance for computational functions.
- **text_resize(event)** - bound to a resizing event, it resizes displayed text in proportion to the window size.

Each visualization is represented by a subclass of `Page`. The structure of `Page` dependencies can be seen in figure 4.3. The widgets needed for displaying the page are initialized in a constructor of each `Page` subclass. Then each class representing a visualization has a set of methods specific for a given visualization. The methods all of the classes share are:

- **on_resize(event)** - resizes the frames containing the visualization in proportion to the window size.
- **init_data(*args)** - sets the parameters of the class according to the received arguments.
- **new_computation()** - clears the widgets for a new visualization.
- **select_file()** or **load_images()** - opens a file dialog for loading data for the visualization.
- **change_scene()** - updates the widgets according to the visualization.
- **show_img()** - displays the visualization.

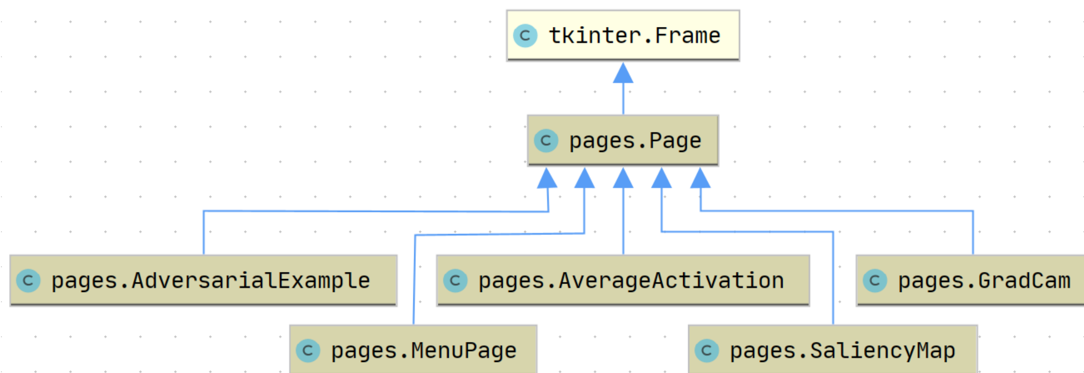

Figure 4.3: The dependencies of the Page class.

AverageActivation

The class `AverageActivation` represents the visualization of average activation. It displays three images for each filter and layer consisting of the two average activations and their difference. Instead of the activation difference, there is an option to display a histogram of the two current average activations, or for each layer, the average difference, or even the graph of the norms of differences across the layers. The methods executing the above activities and more are:

- **save_data(data, path) -** saves given data to a provided path.
- **save_computed() -** saves all of the computed activation averages.
- **load_precomputed() -** loads previously computed and saved activation averages.
- **reset_buttons() -** sets the state of the buttons to a default.
- **difference_graph() -** shows or hides the graph of average/max/min norms of differences across the layers.
- **create_histogram(recreate) -** shows or hides the histogram for the current average activations displayed.
- **layer_average() -** show or hides the average activation difference of the current layer.
- **build_filter_slider(range) -** create a slider for selecting filters (only for convolutional layers).
- **change_layer(layer) -** updates the widgets for a new layer.
- **vis_average_activation() -** initializes a new visualization by calling the **vis_aa(data, n, classes, ord, load)** function from the `Tools` instance.

The overview of the `AverageActivation` class with all of its methods can be seen in the figure 4.4.

pages.AverageActivation	
__init__(self, *args, **kwargs)	init_data(self, data=None, mode=None, model=None, text=None)
new_computation(self)	create_histogram(self, recreate=False)
on_resize(self, event)	layer_average(self)
load_images(self, controller)	show_img(self)
load_precomputed(self, controller)	build_filter_slider(self, range)
save_data(self, datas, path)	change_layer(self, layer=0)
save_computed(self, controller)	change_scene(self)
reset_buttons(self)	vis_average_activation(self, controller)
difference_graph(self)	

Figure 4.4: The `AverageActivation` class and its methods.

GradCam

The class `GradCam` represents the visualization of Grad-CAM. It displays two images: the original image and the image with overlayed heatmap. These are its methods:

- **validate(P) -** check if the target entry contains only digits.
- **set_radio_buttons() -** creates buttons for displaying heatmap for each desired class.

- **overlay_heatmap(heatmap, image, alpha) -** super-imposes the generated heatmap onto the original image.

- **vis_gradcam() -** initializes a new visualization by calling the **vis_gc(data, load, class_indices, n)** function from the `Tools` instance.

The overview of the full `GradCam` class can be seen in the figure 4.5.

pages.GradCam

__init__(self, *args, **kwargs)
validate(self, P)
init_data(self, model=None, data=None, text = None)
on_resize(self, event)
new_computation(self)
select_file(self)
change_scene(self)
set_radio_buttons(self)
overlay_heatmap(self, heatmap, image, alpha=0.5)
show_img(self)
vis_gradcam(self, controller)

Figure 4.5: The `GradCam` class and its methods.

SaliencyMap

The class `SaliencyMap` represents the visualization of Grad-CAM. It displays two images: the original image and the class saliency map. The only unique method in this class is vis_saliency_map() for initializing a new visualization by calling the **vis_sm(data=None, load=None)** function from the `Tools` instance. The overview of all methods of `SaliencyMap` can be seen in the figure 4.6.

pages.SaliencyMap

__init__(self, *args, **kwargs)
init_data(self, model=None, data=None, text = None)
on_resize(self, event)
new_computation(self)
select_file(self)
change_scene(self)
show_img(self)
vis_saliency_map(self, controller)

Figure 4.6: The `SaliencyMap` class and its methods.

AdversarialExample

The class `AdversarialExample` represents the visualization of an adversarial example. It displays three images: the original image, the adversarial example, and its pattern. These are its methods:

- **validate(P) -** same as in the `GradCam` class.
- **comp_activation() -** sends the original and perturbated image to be visualized with the average activation method.
- **comp_gradcam() or comp_smap() -** sends either the original or the perturbated image to be visualized with the grad-CAM or saliency map method
- **vis_adversarial_example() -** initializes a new visualization by calling the **vis_ae(load, method, epsilon, targe, ord)** function from the `Tools` instance.

The overview of the `AdversarialExample` class with all of its methods can be seen in the figure 4.7.

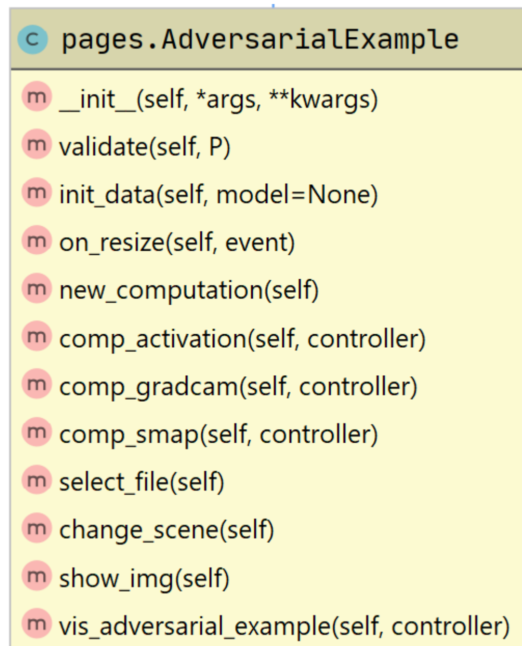

Figure 4.7: The `AdversarialExample` class and its methods.

MenuPage

The class `MenuPage` is also derived from `Page`. It acts as the introductory page and displays the menu. On this page, the user sets the *data_folder* and *model* parameters of the `Tools` instance. The model can be set to the provided models or to a loaded one. The methods executing the above activities are:

- **load_model() -** opens a file dialog and loads the model.

- **set_model()** - sets the *model* parameter.
- **select_dir()** - opens a folder dialog window and sets the *data_folder* parameter to the selected directory.

The overview of the `MenuPage` class can be seen in the figure 4.8.

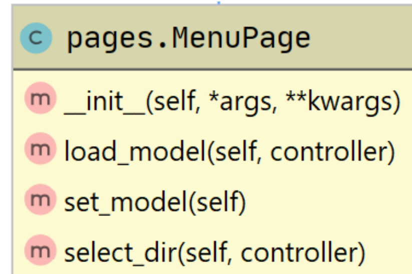

Figure 4.8: The `MenuPage` class and its methods.

4.2.3 Visualizer

The python file `Visualizer` contains the class `VisualizerApp` deriving from the Tkinter Tk. It functions as a master for frames and creates the whole app. The code can be run by running the python file `Visualizer`. The functions implemented inside have the following purpose:

- **select_page(page)** - displays *page*, an instance of `Page` subclass.
- **option_selction()** - calls **select_page(page)** with *page* selected from options menu.
- **set_data(page, *args)** - sets data of *page*, an instance of `Page` subclass, by calling its method **init_data(*args)**.
- **text_resize(event)** - same as in the `Page` class.

The structure of `VisualizerApp` can be seen in the figure 4.9.

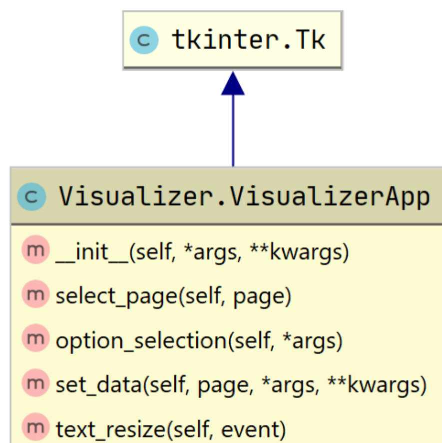

Figure 4.9: The `VisualizerApp` class and its methods.

4.3 Adding more visualizations

In this section, we describe how one could add more visualizations into the *CNN Visualizer* app.

4.3.1 New Visualization method

The new visualization technique should be implemented in the `utils` python file inside the `Tools class`, so it can access the model. For the data needed for the visualization, the class provides sufficient methods on how to acquire them. Therefore only the visualization function is needed.

4.3.2 New Page

To display the new visualization, it is needed to create a new subclass of `Page`. Each page is divided into two Tkinter Frames, one contains the images with descriptions, and the second one contains the control buttons. Then it is important to decide whether two or three images are to be displayed and create widgets accordingly. Each visualization needs an individual approach, but one can take inspiration from the implemented visualizations. The classes `AverageActivation` and `AdversarialExample` implement visualization with three images, and classes `GradCam` and `SaliencyMap` with two.

After creating a new subclass of `Page` and adding all necessary functions into the `Tools` class, they need to be united with the rest of the app. The new page needs to be constructed in the class `VisualizerApp` and also added to the options menu so we can access it. For example, in the following code we can see constructed a page `NewPage` called *NEW PAGE* incorporated to the `VisualizerApp` class.

```
# The options menu after adding the new page:
options = ["Menu",
           "Average Activation",
           "Adversarial example",
           "Saliency Map",
           "Grad-CAM",
           "NEW PAGE"]

# Constructing the pages together with the new class:
for O, P in zip(options, [pages.MenuPage,
                          pages.AverageActivation,
                          pages.AdversarialExample,
                          pages.SaliencyMap,
                          pages.GradCam,
                          pages.NewPage]):
    # Initialize each page object.
    page = P(self)
    page.set_tools_instance(tools_instance)
    self.pages[O] = page
    page.place(in_=frame, x=0, y=0, relwidth=1, relheight=1)
```

5. User Guide

In this chapter, we give detailed instructions on the installation and usage of our visualization tool.

5.1 Installation

In the electronic attachment, there is an executable application included. To run the visualization *CNN Visulizalizer*, Windows 10 or 11 is required. After downloading the attachments, the visualizer can be launched via `CNN_Visualizer.exe`.

It is important to keep the `mnist` folder, containing the `mnist.npz` and `mnist_model.h5`, in the `data` folder. And the `data` folder in the same folder as the `CNN_Visualizer.exe` file. Otherwise, the application will not be able to find the provided MNIST model and dataset.

5.2 Main Menu

The application consists of five different pages. After running the application, we are greeted by the menu page. The page consists of several buttons configuring the visualizer. We can choose one of the provided pre-trained models with buttons (A) (see figure 5.1). The button (B) loads a model from the disk to be used in the visualizations. The button (C) selects a directory from where the visualizations load data. It is advised to set this directory right away.

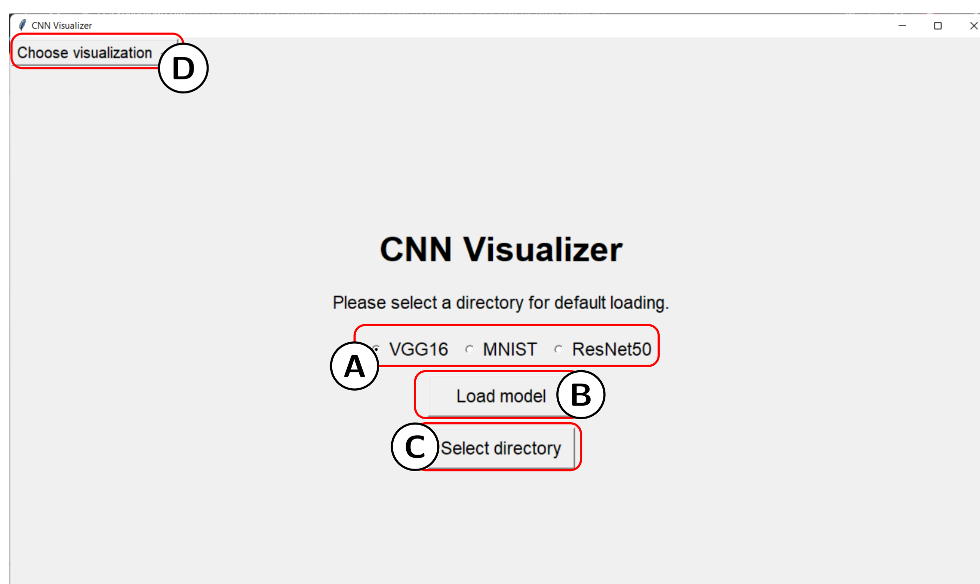

Figure 5.1: The main menu.

After clicking on (D), a collapsible options menu occurs. The options menu is accessible from anywhere inside the application. It is used to navigate between pages. Option (A) in figure 5.2 always takes us back to the main menu, and the rest of the options display a relevant visualization. Option (B) starts the average activation visualization; option (C) starts the visualization of adversarial

example; option (D) starts the class saliency map visualization; option (E) starts the Grad-CAM visualization.

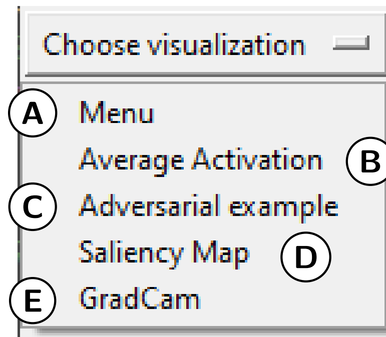

Figure 5.2: The collapsible options menu.

5.3 Average Activation

The Average Activation is visualization described in section 3.1. In the figure 5.3 we can see its introductory page with various settings. The button (A) runs the visualization. The button (B) loads previously saved visualization, and button (C) loads images for the visualization. Note that only `JPG` and `JPEG` files are allowed. They both open a dialog window. Always load a sufficient amount of images. For the **images** mode two images, and for the **classes** mode at least one image from two different classes. The buttons in (D) determine the mode, and the buttons in (E) determine the order of the norm used in the visualization.

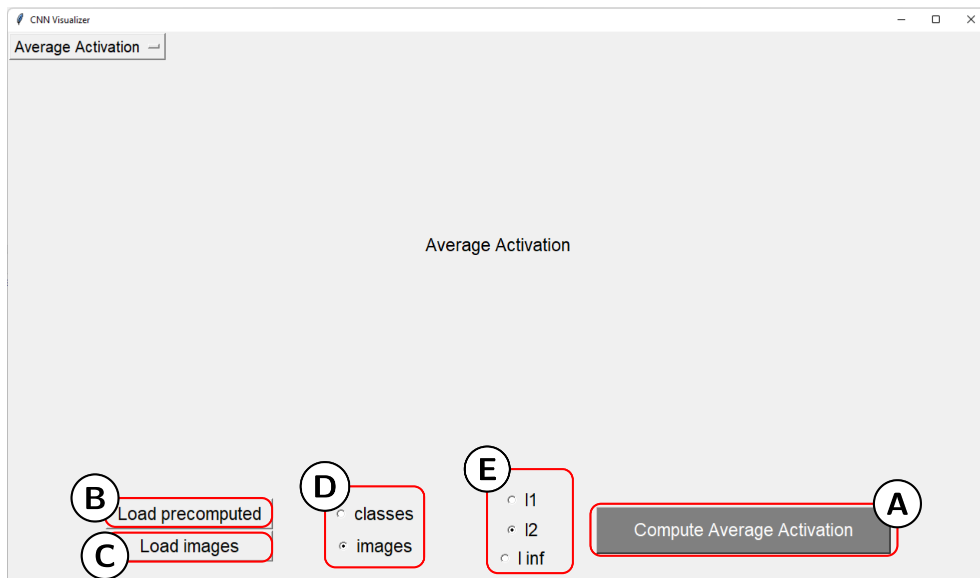

Figure 5.3: The introductory page of the Average Activation Visualization.

After running the visualization with the button (A) from figure 5.3, we can see the results as shown in the figure 5.4. The displayed images on the sides are the two average activations for a given layer and filter, and the image in the middle is their difference. The highlighted area (A) contains the displayed

images' descriptions. The slider (B) is used to navigate across the layers of the used CNN. The layers are ordered in ascending order as they occur in the network. The slider (C) navigates across the filter of each layer. The filters are ordered in descending order with respect to the norm of the activation difference. The real number of the filter can be found in the displayed description.

The buttons ($D - F$) display an image instead of the activation difference. After clicking the button, the text on it will change to *"Hide ..."*, and then when it is clicked, the average difference will be displayed again. The button (D) displays the average difference activation of a current layer, and the button (E) displays the graph of average, minimum, and maximum differences in each layer. The button (E) displays a histogram for the displayed average activations, and when the (F) is checked, a logarithm is applied to the graph's y-axis.

After clicking the (I) button, a pop-up message window asks if you are sure. The reason is that the (I) button saves the current visualization, but the data to be saved can be extensive. When we are ready for a new visualization, we can press button (H), which throws away the current one and takes us back to the introductory page of Average Activation visualization from figure 5.3.

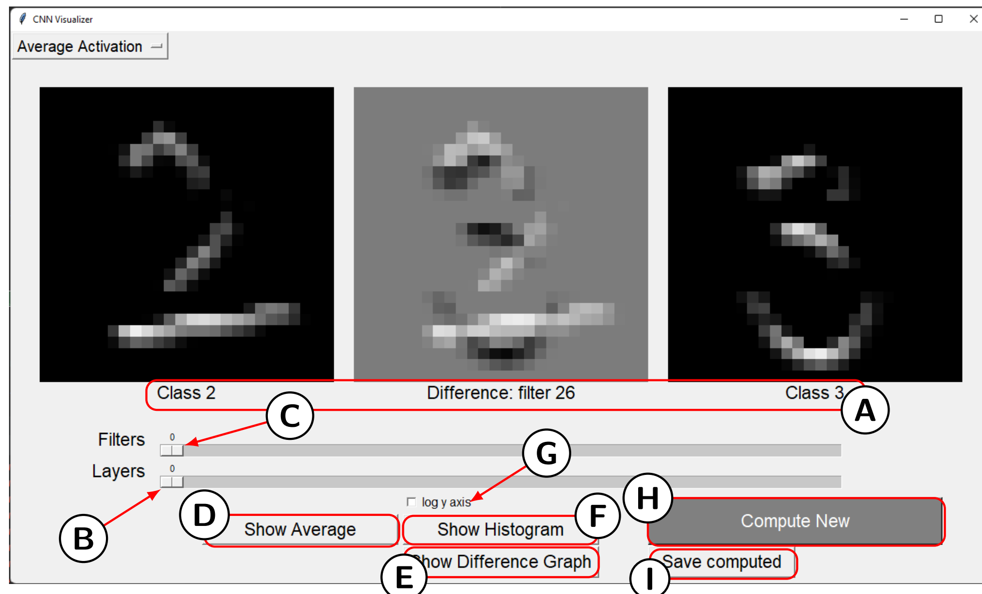

Figure 5.4: Output of Average Activation Visualization.

5.4 Adversarial Example

This visualization generates and displays an adversarial example, the pattern, and the original input image. In the figure 5.5 we can see its introductory page with various settings. The button (A) runs the visualization, and the button (C) loads an image for the visualization. The entry (C) only takes a digit, the targeted class. The not targeted attack is performed when (C) is left empty. The buttons in (E) determine the method of the attack: either FGSM, PGD (both described in 2.2) or random white noise. The slider (D) defines the epsilon parameter of the methods generating the adversarial example.

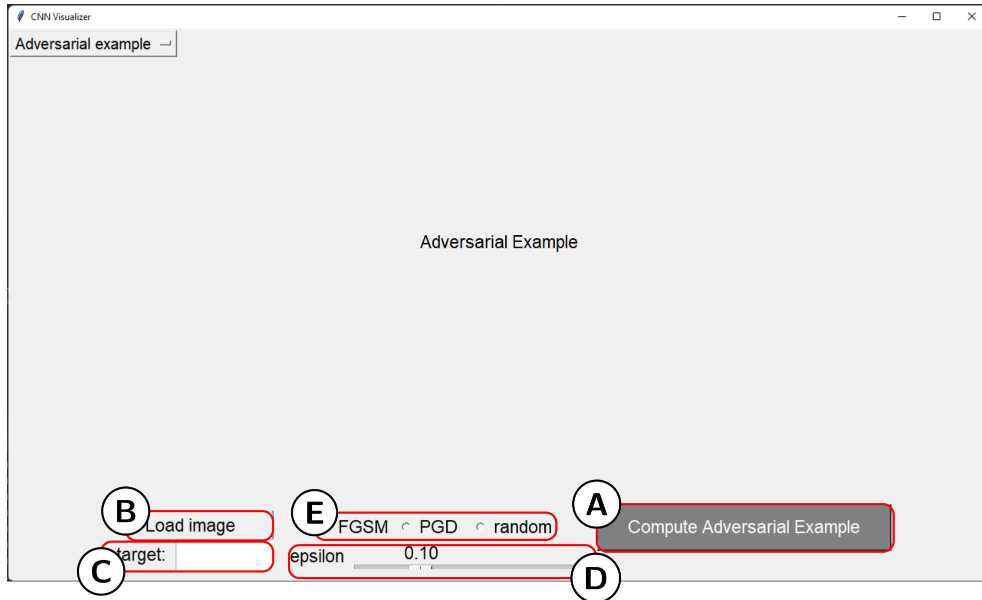

Figure 5.5: The introductory page of the Adversarial Example Visualization.

The figure 5.6 depicts the output adversarial example. The highlighted area (A) contains the description. The buttons $(B - D)$ sends the current data for further visualizations. The button (B) sends both the original image and the perturbated image to the Average Activation visualization. The buttons in (E) selects whether the original or the perturbated image is to be sent further. The buttons (C) and (D) sends the selected image to the Saliency Map and Grad-CAM visualizations respectively. The button (H) throws away the current visualization and takes us back to the introductory page of Adversarial Example visualization.

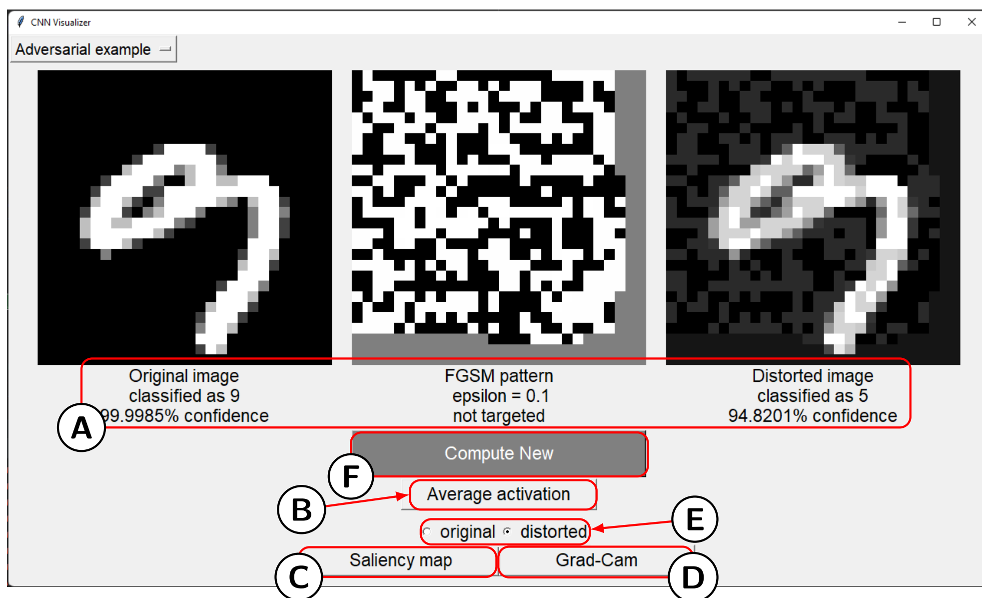

Figure 5.6: Output of the Adversarial Example Visualization.

5.5 Grad-CAM

The Grad-CAM visualization (as described in section 3.3) displays the input image and a heatmap of predictions for the top five classes and an optional sixth one. In the figure 5.7 we can see its introductory page with the button (A) for running the visualization and button (B) for loading an input image. The entry (C) again only takes digits and determines the optional sixth class.

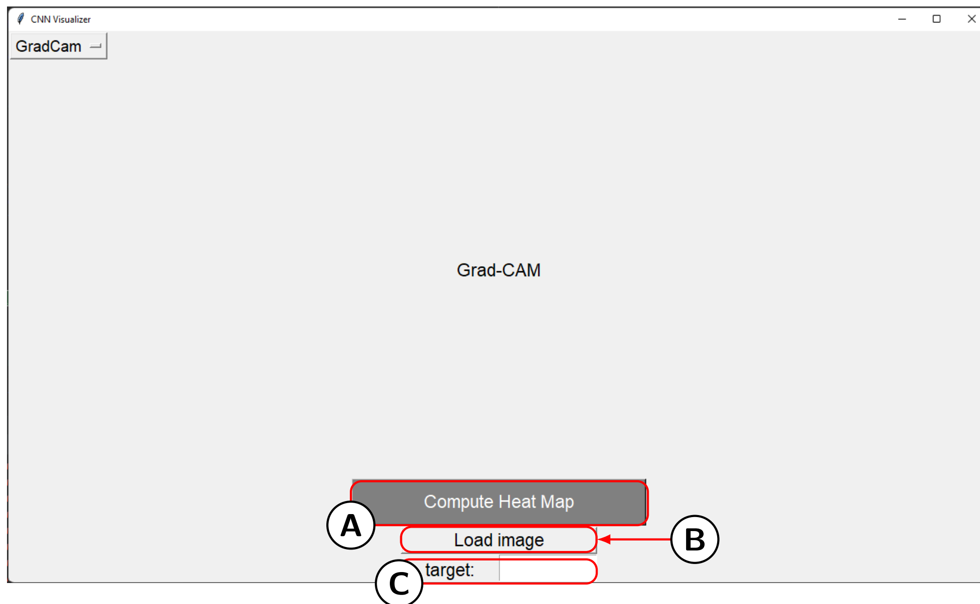

Figure 5.7: The introductory page of the Grad-CAM Visualization.

An example of the output of Grad-CAM can be seen in the figure 5.8. In the area (A) is pointing to, the description is displayed. Buttons in the area (C) select the class for which the heatmap is displayed, and button (B) takes us back to the introductory page of Grad-CAM for a new visualization.

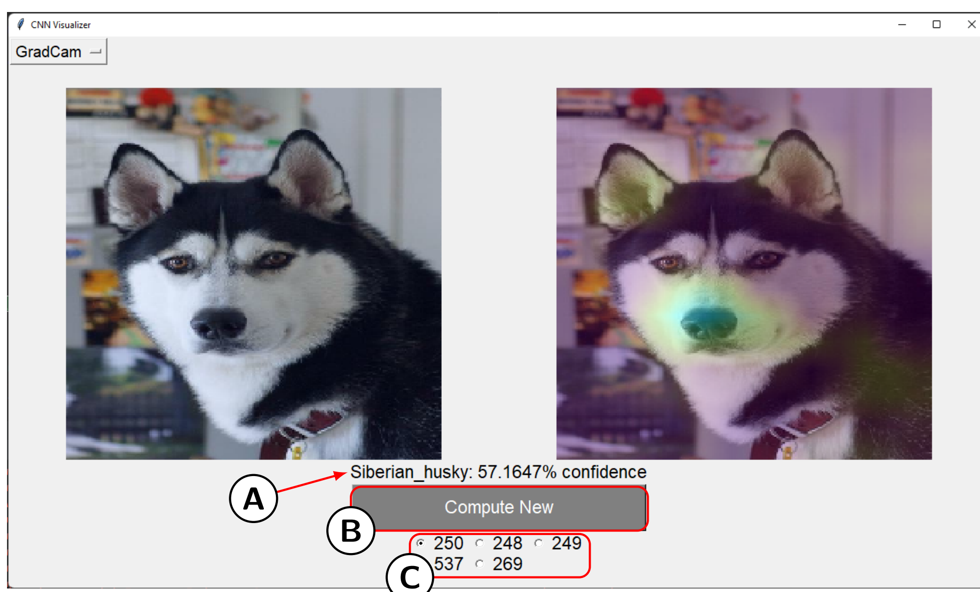

Figure 5.8: Output of Grad-CAM Visualization.

5.6 Saliency Map

The last visualization is Saliency Map. It is the most simple one. It only displays the input image and its saliency map. The introductory page of the Saliency Map visualization can be seen in the figure 5.9, where the button (A) runs the visualization and the button (B) loads an input image.

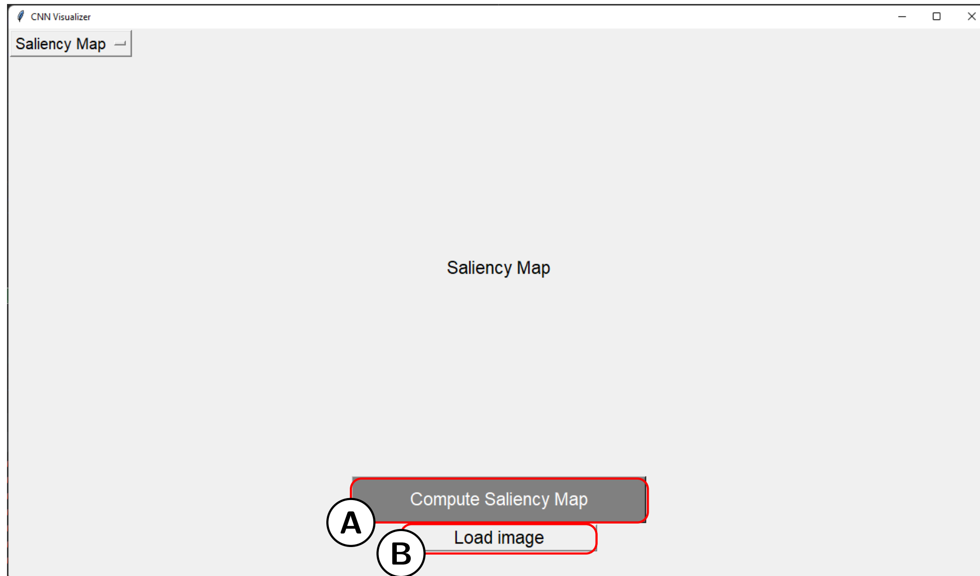

Figure 5.9: The introductory page of the Saliency Map Visualization.

The figure 5.6 depicts an example of the Saliency Map output. The arrow (B) points to the description, and the button (A) takes us back to initialize a new visualization on the introductory page.

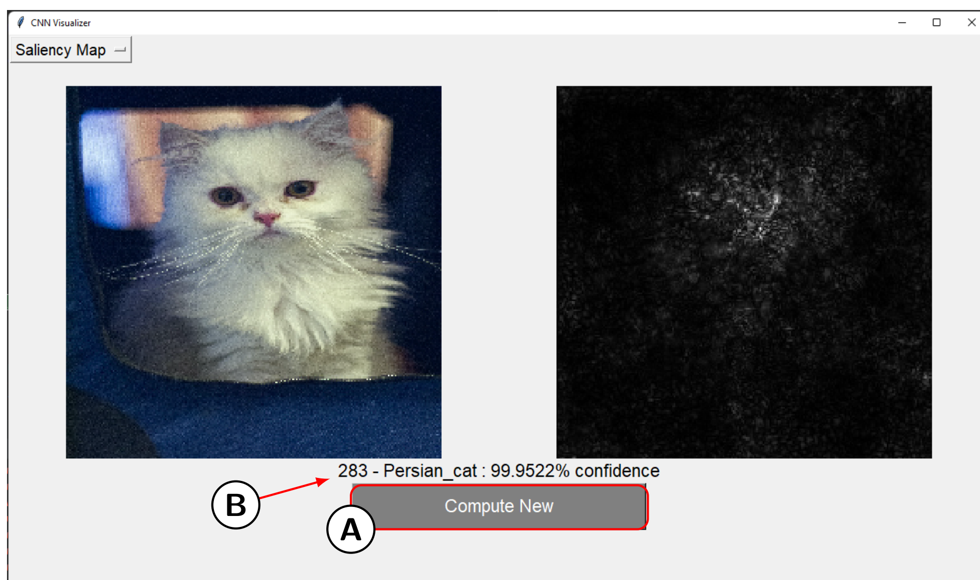

Figure 5.10: Output of Saliency Map Visualization.

6. Experiments

In this chapter, we describe the behavior of CNNs in various scenarios as observed by the visualization tool *CNN Visualizer*. The main goal is to show the variety of possible visualizations and analyses of networks inside the application. All of the images used are from the MNIST dataset [11] or from a subset of ImageNet [14] called Imagenette [37].

6.1 Average Activation Visualization Results

This section describes the results obtained using the Average Activation Visualization.

6.1.1 Filters Across the Network

As we have mentioned in section 1.1.1 the complexity of the detected features of the image depends on how deep in the network the given convolutional filter lays. This can be observed while sliding across the layers of the network, we can notice how the average feature maps get from clear outlines of the original images to unrecognizable shapes (see figure 6.1).

Figure 6.1: The Average Activation for the filters with the biggest difference from the first, the seventh, and the last convolutional layer respectively in the VGG16 model.

Also, as we dive deeper into the network, thanks to the histogram feature, we can notice that number of pixels with low-value increases. This can indicate that the filters indeed filter out only the learned features and focus on them (see figure 6.2).

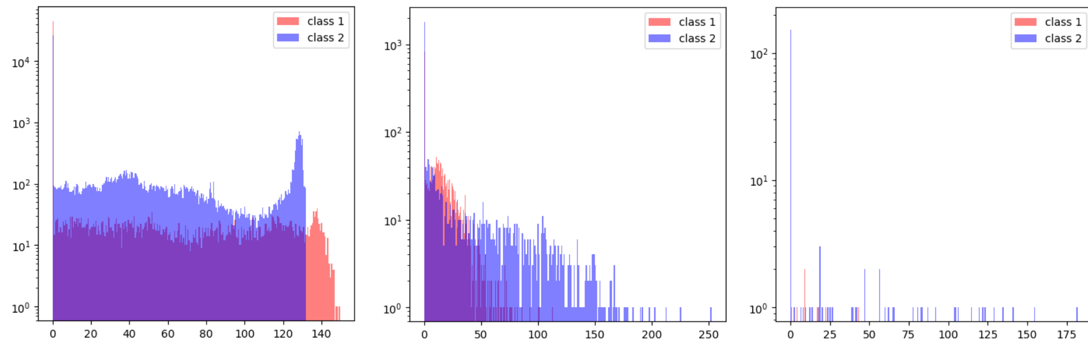

Figure 6.2: The histograms for the images in figure 6.1. From left: first, seventh, last convolutional layer.

Deep inside the network, in the last convolutional layers, we have experienced that two images classified differently, tend to activate different filters. On the other hand, the activated filters for the images placed in the same class significantly overlap.

6.1.2 Prediction

Someone might hope that the visualization of the last layers would indicate how the network chooses a specific classification. This is far from the truth. Usually, the last layers of a CNN are fully-connected, and their output is just a one-dimensional array. In the application, we transform the array into a matrix for a comprehensible visualization, but the resulting image only gives out a noise.

The most legible fully-connected layer is the very last one, used for prediction. In the visualization of the last layer, we can usually clearly see that one pixel is brighter than the others. This brightest pixel corresponds to the predicted class (see figure 6.3).

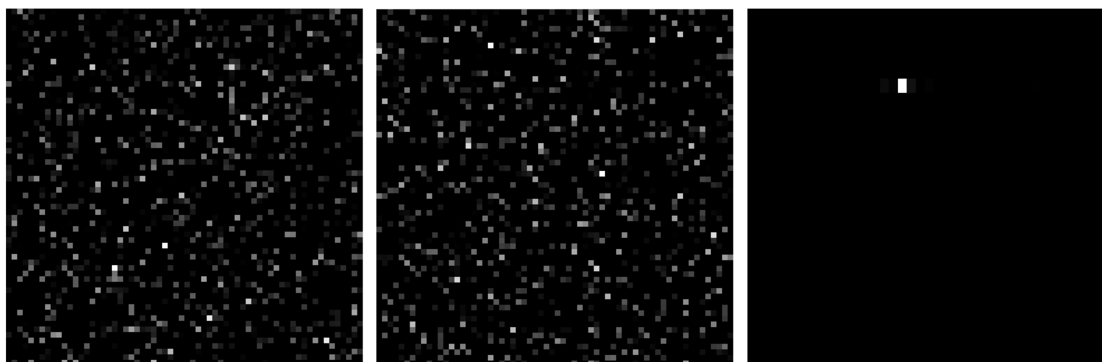

Figure 6.3: The outputs of fully-connected layers of the VGG16 model for an image classified as *English_springer*.

6.2 Fooling CNN

This section looks at what the network focuses on when dealing with an input that is meant to confuse it.

6.2.1 Partially Covered Input

First, we explore what happens when we cover the crucial part of the input image for the classification. Grad-CAM helps us reveal the targeted region.

Figure 6.4: On the left: the heatmap for the *English_springer* prediction, and on the right: the heatmap for the *Welsh_springer_spaniel* in the original image.

Then we cover the highlighted area and the resulting image put into the VGG16 model. The original image was classified as *English_springer* with over 86% confidence. The covered was classified as *Welsh_springer_spaniel*, which originally had under 6% confidence. When we covered other parts of the image, the original classification did not change. It is safe to say that the grad-CAM really returns the crucial region for a classification.

Figure 6.5: On the left: the heatmap for the *English_springer* prediction, and on the right: the heatmap for the *Welsh_springer_spaniel* in the covered image.

When we put the original image into the Average Activation Visualization, we discovered that other than in the covered region, the neurons act more or less the same, especially in the first half of the VGG16 network. In the deeper layers, the change in the image spread across a bigger part of the image.

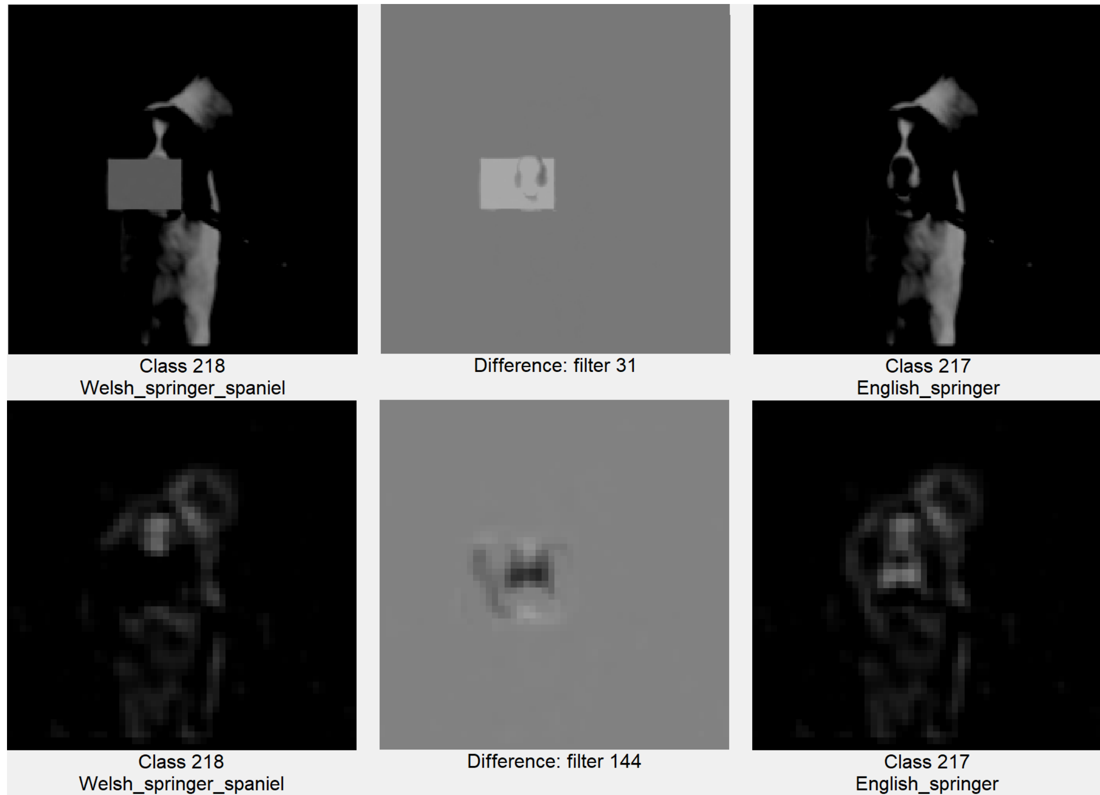

Figure 6.6: The first row is the Average Activation output with the biggest difference from the first layer, and the second is from the seventh convolutional layer.

6.2.2 Adversarial Example

In this section, we explore how the prediction process differs with an adversarial example from the process with the original input. First, we generate the adversarial example using the FGSM method, with epsilon 0.2. Then we send the data for further visualization.

Figure 6.7: The original image was correctly classified as 0 with 100% confidence, while the adversarial example was classified as 8 with almost 80% confidence

First, we use the Grad-CAM to see the important regions in the images for the classifications. We can see that in the original image, it is the middle part of the number for the correct classification (figure 6.8a). The miss-classified class highlights mostly the background in the original image. In the perturbed image,

the highlighted region for class 8 significantly intensified (figure 6.8b).

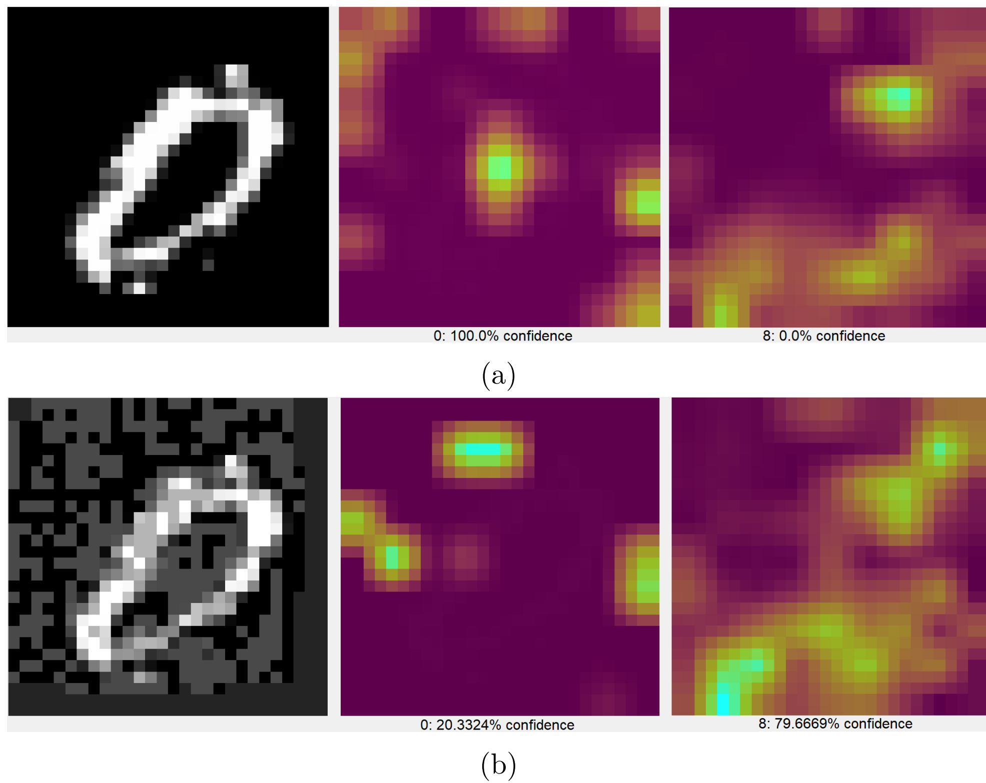

Figure 6.8: Heatmaps of (a) the original image, and the (b) adversarial example for classes 0 and 8.

In the visualization of Average Activation we can see that perturbed image's activation differ from the one of the original image in the first layer. This is not surprising because even if, to a human, the change can be unnoticed, the adversarial example added noise across the whole image. In the figure 6.9 we can see the first layer with the filter with the biggest difference. We can see that the noise is persistent in the activations.

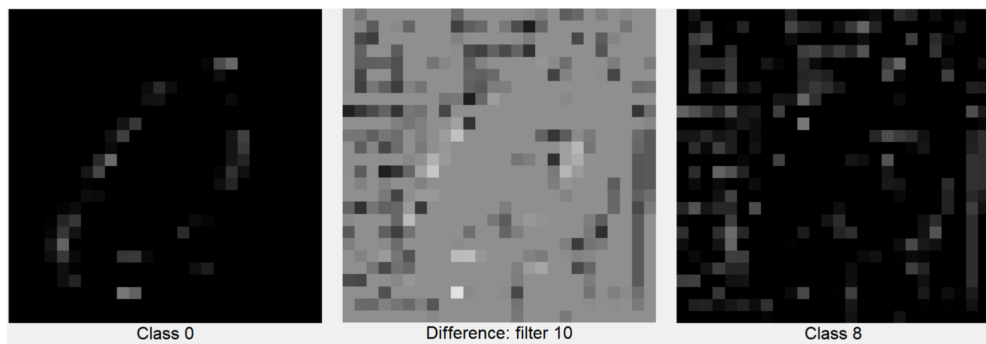

Figure 6.9: The Average Activation output with the biggest difference from the first layer for the adversarial example and the original image from the figure 6.7.

Conclusion

We implemented a tool for visualizing the convolutional neural networks. In the introduction, we listed five requirements the implementation should include, and we designed the application around them. In this section, we describe how we accomplished them.

Activation-based visualization: We have included the Average Activation Visualization, which provides an insight into what features were detected in the decision process of a network. The user has an opportunity to observe the process of transforming the input image through abstract feature maps all the way to the prediction.

Variety of visualizations: Besides the Average Activation Visualization, we included the visualizations of the Class Saliency Map and also the Grad-CAM. They provide a look inside the interconnected workings and the focus of the model.

Not model-specific: We have provided three pre-trained models, VGG16, ResNet50, and a simple model for classifying data from the MNIST dataset. The option to input own pre-trained model is also supported.

Adversarial examples: We implemented an option to generate an adversarial example so that it can be used for more profound analyses of the network. It is connected to all visualizations, so the user has a chance to see them in action.

User-friendly environment: The graphical interface was designed with simplicity in mind. The descriptions at each step guarantee smooth usage of the tool.

All of these goals ensured there are opportunities for broad and precise analyses of CNN models. We have also shown how the tool can be used to understand different prediction scenarios. The Average Activation Visualization is useful for better comprehension of the convolutional filters. But to fully apprehend the working of CNNs, it is best to use a combination of instruments.

To further extend the visualizer, more visualization techniques could be added. The new possible methods could include visualizations of the model's architecture or the updated versions of the implemented methods. One could even implement an option to modify or ultimately create an input for the network.

Bibliography

[1] Keiron O'Shea and Ryan Nash. An introduction to convolutional neural networks. *CoRR*, abs/1511.08458, 2015.

[2] Neural networks - Graphics with TikZ in LaTeX. `https://tikz.net/neural_networks/`. Accessed: July 2022.

[3] Saad Albawi, Tareq Abed Mohammed, and Saad Al-Zawi. Understanding of a convolutional neural network. In *2017 International Conference on Engineering and Technology (ICET)*, pages 1–6. IEEE, 2017.

[4] Jiuxiang Gu, Zhenhua Wang, Jason Kuen, Lianyang Ma, Amir Shahroudy, Bing Shuai, Ting Liu, Xingxing Wang, Gang Wang, Jianfei Cai, and Tsuhan Chen. Recent advances in convolutional neural networks. *Pattern Recognition*, 77:354–377, 2018.

[5] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. Adaptive computation and machine learning. The MIT Press, Cambridge, Massachusetts, 2016. `http://www.deeplearningbook.org`.

[6] Salman Khan, Hossein Rahmani, Syed Afaq Ali Shah, and Mohammed Bennamoun. *A guide to convolutional neural networks for computer vision*. Synthesis Lectures on Computer Vision, Number 15. Morgan & Claypool Publishers, Place of publication not identified, 2018.

[7] Sagar Sharma, Simone Sharma, and Anidhya Athaiya. Activation functions in neural networks. *towards data science*, 6(12), 2017. `https://www.ijeast.com/papers/310-316,Tesma412,IJEAST.pdf`. Accessed: July 2022.

[8] Y-Lan Boureau, Jean Ponce, and Yann LeCun. A theoretical analysis of feature pooling in visual recognition. In Johannes Fürnkranz and Thorsten Joachims, editors, *Proceedings of the 27th International Conference on Machine Learning (ICML-10)*, pages 111–118, Haifa, Israel, June 2010. Omnipress.

[9] Mohit Sewak, Md Rezaul Karim, and Pradeep Pujari. *Practical convolutional neural networks: implement advanced deep learning models using Python*. Packt Publishing Ltd, 2018.

[10] Li Deng. A tutorial survey of architectures, algorithms, and applications for deep learning – erratum. *APSIPA Transactions on Signal and Information Processing*, 3(1), 2014.

[11] The MNIST database. `http://yann.lecun.com/exdb/mnist/`. Accessed: 2022.

[12] Li Deng. The MNIST Database of Handwritten Digit Images for Machine Learning Research [Best of the Web]. *IEEE Signal Processing Magazine*, 29(6), 2012.

[13] Hugo Larochelle, Y. Bengio, Jérôme Louradour, and Pascal Lamblin. Exploring Strategies for Training Deep Neural Networks. *Journal of Machine Learning Research*, 1, 01 2009.

[14] Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. ImageNet: A large-scale hierarchical image database. In *2009 IEEE Conference on Computer Vision and Pattern Recognition*, pages 248–255, 2009.

[15] Olga Russakovsky, Jia Deng, Hao Su, Jonathan Krause, Sanjeev Satheesh, Sean Ma, Zhiheng Huang, Andrej Karpathy, Aditya Khosla, Michael Bernstein, et al. ImageNet Large Scale Visual Recognition Challenge. *International Journal of Computer Vision*, 115(3):211–252, Dec 2015.

[16] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. *arXiv 1409.1556*, 2014.

[17] Max Ferguson, Ronay ak, Yung-Tsun Lee, and Kincho Law. Automatic localization of casting defects with convolutional neural networks. In *2017 IEEE International Conference on Big Data (Big Data)*. IEEE, 12 2017.

[18] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 770–778, 2016.

[19] Sai Kumar Basaveswara. CNN architectures, a deep-dive. *Medium*, May 2021.

[20] Christian Szegedy, Wojciech Zaremba, Ilya Sutskever, Joan Bruna, Dumitru Erhan, Ian Goodfellow, and Rob Fergus. Intriguing properties of neural networks. 2013.

[21] Ian J. Goodfellow, Jonathon Shlens, and Christian Szegedy. Explaining and harnessing adversarial examples. 2014.

[22] Nicolas Papernot, Patrick McDaniel, Arunesh Sinha, and Michael Wellman. Towards the science of security and privacy in machine learning. 2016.

[23] Anirban Chakraborty, Manaar Alam, Vishal Dey, Anupam Chattopadhyay, and Debdeep Mukhopadhyay. Adversarial attacks and defences: A survey. 2018.

[24] Alexey Kurakin, Ian Goodfellow, and Samy Bengio. Adversarial examples in the physical world. 2016.

[25] Aleksander Madry, Aleksandar Makelov, Ludwig Schmidt, Dimitris Tsipras, and Adrian Vladu. Towards deep learning models resistant to adversarial attacks. 2017.

[26] Christoph Molnar. *Interpretable Machine Learning*. Independently published, 2 edition, 2022.

[27] Chris Olah, Alexander Mordvintsev, and Ludwig Schubert. Feature visualization. *Distill*, 2017. https://distill.pub/2017/feature-visualization.

[28] Karen Simonyan, Andrea Vedaldi, and Andrew Zisserman. Deep inside convolutional networks: Visualising image classification models and saliency maps. 2013.

[29] Ramprasaath R. Selvaraju, Michael Cogswell, Abhishek Das, Ramakrishna Vedantam, Devi Parikh, and Dhruv Batra. Grad-cam: Visual explanations from deep networks via gradient-based localization. In *Proceedings of the IEEE International Conference on Computer Vision (ICCV)*. IEEE, Oct 2017.

[30] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dandelion Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. Software available from tensorflow.org.

[31] François Chollet et al. Keras. `https://keras.io`, 2015. Accessed: July 2022.

[32] Charles R. Harris, K. Jarrod Millman, Stéfan J. van der Walt, Ralf Gommers, Pauli Virtanen, David Cournapeau, Eric Wieser, Julian Taylor, Sebastian Berg, Nathaniel J. Smith, Robert Kern, Matti Picus, Stephan Hoyer, Marten H. van Kerkwijk, Matthew Brett, Allan Haldane, Jaime Fernández del Río, Mark Wiebe, Pearu Peterson, Pierre Gérard-Marchant, Kevin Sheppard, Tyler Reddy, Warren Weckesser, Hameer Abbasi, Christoph Gohlke, and Travis E. Oliphant. Array programming with NumPy. *Nature*, 585(7825):357–362, September 2020.

[33] Alex Clark. Pillow (PIL Fork) documentation, 2015. `https://buildmedia.readthedocs.org/media/pdf/pillow/latest/pillow.pdf`.

[34] G. Bradski. The OpenCV Library. *Dr. Dobb's Journal of Software Tools*, 2000.

[35] J. D. Hunter. Matplotlib: A 2d graphics environment. *Computing in Science & Engineering*, 9(3):90–95, 2007.

[36] Nicolas Papernot, Ian Goodfellow, Ryan Sheatsley, Reuben Feinman, and Patrick McDaniel. cleverhans v1.0.0: an adversarial machine learning library. *arXiv preprint arXiv:1610.00768*, 2016.

[37] Jeremy Howard. Imagenette. `https://github.com/fastai/imagenette/`.

A. Attachments

A.1 Contents of electronic attachment

- data .... a directory containing data
  - mnist .... a directory containing mnist data
    - mnist.npz .... MNIST dataset
    - mnist_model.h5 .... MNIST model
  - test .... a directory containing data used in exmeriments
- src .... a directory containing source codes
  - pages.py .... python file containing the page classes
  - requirements.txt ... required packedges for running the source codes
  - Vysualizer.py .... python file building the application
  - utils.py .... python file containg the tools class
- thesis.pdf .... electronic version of this thesis
- CNN_Visualizer.exe .... the application