University of São Paulo
Institute of Mathematics and Statistics
Bachelor of Computer Science

Explainable AI (XAI) Methods for Convolutional Neural Networks

Antonio Fernando Silva e Cruz Filho
João Gabriel Andrade de Araujo Josephik

Final Essay

mac 499 — Capstone Project

Supervisor: Prof. Dr. Nina S. T. Hirata

São Paulo
2024

Acknowledgements

I dedicate this work to my family, especially my mother, whose constant support has helped me pursue my dream of studying at USP from the very beginning. I am very grateful to my advisor, Nina, for her guidance, not only in this project but also for introducing me to the amazing world of Machine Learning. I also want to thank my girlfriend, Giovanna, for always being by my side, supporting me during tough times, believing in my dreams, and inspiring me with new ideas. Finally, I dedicate this work to my amazing friends from BCC, who always motivate and inspire me to become a better person and reach new goals. - Fernando Cruz

Resumo

Antonio Fernando Silva e Cruz Filho
João Gabriel Andrade de Araujo Josephik. **Métodos de IA explicável (XAI) para Redes Neurais Convolucionais**. Monografia (Bacharelado). Instituto de Matemática e Estatística, Universidade de São Paulo, São Paulo, 2024.

Com a ascensão do uso de Aprendizado de Máquina para problemas de Visão Computacional, o uso de Redes Neurais Convolucionais (CNNs) se mostrou uma peça fundamental para a criação de modelos estado da arte em tarefas como classificação, detecção de objetos e até mesmo segmentação. No entanto,em muitos casos, a simples obtenção do resultado de uma predição não é suficiente, sendo necessária uma justificativa para as decisões do modelo. Utilizando IA Explicável (XAI), podemos encontrar possíveis explicações para predições de modelos complexos como Redes Convolucionais. Nesse trabalho, foram estudadas diversas técnicas de Explicabilidade aplicadas a CNNs, utilizando de técnicas como GradCam e Visualização de Características. Além disso, foram conduzidos experimentos com cada técnica abordada visando avaliar a eficácia na interpretação dos modelos de Visão Computacional.

Palavras-chave: IA. Aprendizado de Máquina. IA Explicável. XAI. Visualização de Características. GradCam. LIME.

Abstract

Antonio Fernando Silva e Cruz Filho
João Gabriel Andrade de Araujo Josephik. **Explainable AI (XAI) Methods for Convolutional Neural Networks**. Capstone Project Report (Bachelor). Institute of Mathematics and Statistics, University of São Paulo, São Paulo, 2024.

With the rise of Machine Learning in Computer Vision problems, the use of Convolutional Neural Networks (CNNs) has proven to be a fundamental component in developing state-of-the-art models for tasks such as classification, object detection, and even segmentation. However, in many cases, simply obtaining the result of a prediction is not sufficient; a justification for the model's decisions is necessary. By employing Explainable AI (XAI), it is possible to identify potential explanations for the predictions of complex models such as Convolutional Neural Networks. In this study, various explainability techniques applied to CNNs were analyzed, utilizing methods such as Grad-CAM and Feature Visualization. Additionally, experiments were conducted with each technique to assess their effectiveness in interpreting Computer Vision models.

Keywords: AI. Machine Learning. Explainable AI. XAI. Feature visualization. GradCam. LIME.

List of Abbreviations

AI	Artificial Intelligence
ML	Machine Learning
XAI	Explainable AI
MLP	Multilayer Perceptron
CNN	Convolutional Neural Network
Conv	Convolution
IME	Institute of Mathematics and Statistics
USP	University of São Paulo

List of Tables

List of Programs

Contents

Chapter 1

Background

In this chapter, we will introduce important concepts required for the understanding of this study. We begin by introducing Explainable AI (XAI) and its principles. We also introduce Neural Networks and important concepts such as Gradient Descent and Back Propagation. Finally, we introduce Convolutional Neural Networks, the main focus of this study.

1.1 Explainable AI

With the rise of Machine Learning models in the last decade in the business and academic areas, Artificial Intelligence (AI) is becoming increasingly present in important decision-making tasks. However, as AI models have become more sophisticated, particularly with the advent of Deep Learning techniques, their internal workings have often remained opaque. Explainable AI (XAI) aims to make models and their decisions more transparent, interpretable and understandable to both experts and inexperienced users.

1.1.1 What is Explainable AI?

Defining a mathematical formalization to explainability of Machine Learning is a difficult task considering the subjective nature of what one may consider "explainable". In non-mathematical terms, Explainability in AI refers to the capacity to articulate or justify the behavior of a model, focusing on methods that explain a model's decisions after they are made.

Another important concept in the area is Interpretability, which can be defined as "the degree to which a human can understand the cause of a decision" by Miller (2017).[1] In this case, however, a model's decision is understandable entirely by its inherent transparency. In other terms, the model is simple enough to be interpretable by a human directly, without the use of external techniques.

[1] Miller, Tim. "Explanation in artificial intelligence: Insights from the social sciences." arXiv Preprint arXiv:1706.07269. (2017).

Models with low complexity whose decisions are understandable by humans are defined as *Interpretable Models*. Linear Regression, Logistic Regression and Decision Tree models are examples of models classified as *Interpretable Models*. Now, models with a level of complexity that prevents humans from directly understanding their decision-making processes are referred to as *Explainable Models*. Recently popular *Deep Learning Models* are one kind of *Explainable Models* and will be the main focus of this essay, especially *Deep Convolutional Neural Networks*, explored in section 1.3.

1.1.2 Why Explainable AI is Necessary

Creating explanations to a model's decisions can yield many advantages, including more ethical and fair decisions, correctly following regulatory compliances and easier model debugging.

To ensure ethical and fair decision-making, Machine Learning systems must provide justifiable decisions, as they often exploit discriminatory patterns to enhance accuracy, which can perpetuate harmful biases. For instance, the COMPAS algorithm, used in U.S. courts to assess recidivism risk, was analyzed by ProPublica[1] and found to exhibit significant bias against Black defendants, frequently overestimating their likelihood of reoffending compared to their actual risk.

Explainable AI (XAI) is sometimes a mandatory requirement, particularly under regulations like the United Kingdom's General Data Protection Regulation (GDPR). The GDPR mandates that organizations must provide clear and understandable explanations for decisions that significantly impact individuals, especially those made by automated systems commonly powered by Machine Learning algorithms. Without XAI, high-stakes decisions cannot leverage such models in the United Kingdom, highlighting the crucial role of explainability in enabling the broader adoption of Machine Learning for real-world applications while ensuring compliance and fairness.

When debugging Machine Learning models, their behavior can often be unpredictable, revealing biases that may not have been initially apparent to humans. These biases can result in high performance on training, validation, or even test datasets but lead to poor performance in real-world deployment. For instance, consider training an image classifier to differentiate between dog and cat images. The model may achieve impressive accuracy on images of dogs in green fields. However, upon examining the regions of the image the model relies on for its predictions, researchers might discover that it focuses on the background rather than the animals themselves. This happens because dog owners are more likely to photograph their pets outdoors, leading to an unintended association between dogs and green backgrounds. Techniques from Explainable AI, such as Grad-CAM (Selvaraju *et al.*, 2019) and Gradient Saliency methods, enable researchers to visualize these image regions, providing critical insights into model behavior and helping to address such biases.

[1] *"How We Analyzed the COMPAS Recidivism Algorithm", by ProPublica: https://www.propublica.org/article/how-we-analyzed-the-compas-recidivism-algorithm*

1.2 Gradient Descent

Let $f : A \to B$ where $A \subseteq \mathbb{R}^n$ for $n \in \mathbb{N}$ and $B \subseteq \mathbb{R}^+$. Suppose we want to find the solution to the optimization problem

$$\underset{x \in A}{\operatorname{argmin}} f(x) \tag{1.1}$$

when $\frac{\partial f}{\partial x}$ is known for any value of x. Considering that the vector $\frac{\partial f}{\partial x}$ points to the direction of the steepest ascent of the function, the vector $-\frac{\partial f}{\partial x}$ will point to the steepest descent from the given point x. Therefore, one can define an initial random value for x and update x using $-\frac{\partial f}{\partial x}$ and a scaling factor η in order to find a local minimum of f and an approximation to the solution of given optimization problem.

We can define such method using the following formula, where x_t represents the value of x at iteration t of the algorithm:

$$x_{t+1} = x_t - \eta \frac{\partial f(x_t)}{\partial x_t}. \tag{1.2}$$

The term η is often called the *learning rate* used in the Gradient Descent method and is often defined manually by the user.

The Gradient Descent method can be used to optimize a neural network's parameters to solve a given problem using a *loss function*.

1.3 Neural Networks

Neural Networks are proven to be universal approximators.[2] That means that Neural Networks are Machine Learning models capable of representing any continuous function, therefore making Neural networks adept at modeling a range of different complex problems. This class of models have seen a growing presence across both academic and industry landscapes. However, given the architecture of multiple hidden layers of Neural Networks creating complex internal patterns, such models are classified as Explainable Models.

In this section, the inner workings of Neural Networks will be explained, starting with the *Perceptron*, considered the fundamental building block of Neural Networks.

1.3.1 Perceptron

A Perceptron is a Machine Learning model inspired by how biological neurons work. It is a simple binary linear classifier that defines its parameters by linear combinations of points in the dataset. The Perceptron model can be described by Figure 1.1.

[2] Hornik, K., Stinchcombe, M., White, H. Multilayer feedforward networks are universal approximators. https://doi.org/10.1016/0893-6080(89)90020-8

[3] https://towardsdatascience.com/what-the-hell-is-perceptron-626217814f53

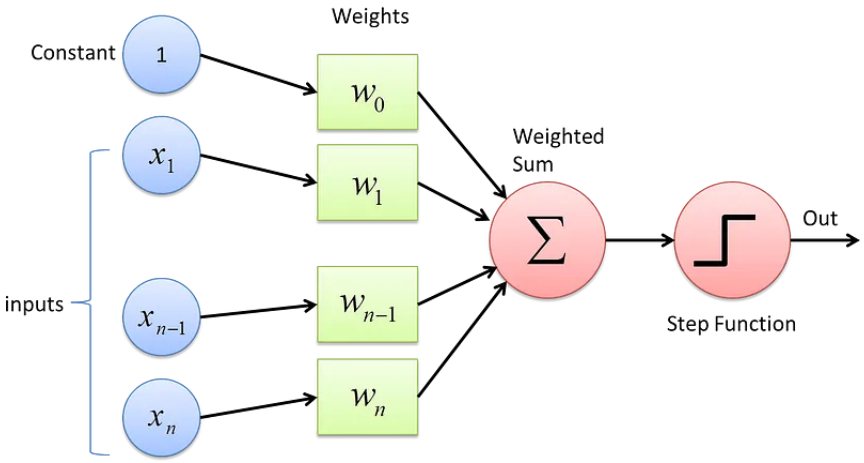


Figure 1.1: *Perceptron Architecture. Font:* Towards Data Science[3].

Where the weights w_i for $i \in \{0, 1, \cdots, n\}$ are trainable parameters and the step function can be defined as $\sigma : \mathbb{R} \to \{0, 1\}$ such that

$$\sigma(x) = \begin{cases} 1 & \text{if } x \geq 0 \\ 0 & \text{if } x < 0. \end{cases} \tag{1.3}$$

Therefore, the Perceptron model can be defined as the function[4] $f : \mathbb{R}^n \to \{0, 1\}$ where

$$f(x) = \sigma(w_0 + \sum_{i=1}^{n} w_i x_i). \tag{1.4}$$

The Perceptron model updates its parameters using each sample (x, y) of the dataset with the rule

$$w_i^{t+1} = w_i^t + \eta \, (y - f(x)) x_i \tag{1.5}$$

for $i \in \{1, \cdots, n\}$ and

$$w_0^{t+1} = w_0^t + \eta \, (y - f(x)), \tag{1.6}$$

where η is the *learning rate* hyperparameter and t is the update iteration number.

As a linear model, the Perceptron can only model linear problems, which only represent a small subset of real world problems. As a solution, researchers started combining Perceptrons in a layered structure, called Multilayer Perceptron, also famously known as *Neural Networks*.

[4] The independent term w_0 is usually called the *bias* of the Perceptron (or neuron)

1.3.2 Multilayer Perceptron (MLP)

By stacking multiple Perceptrons into multiple layers, one can build more complex decision boundaries and model more complex functions. For example, by using the Multilayer Perceptron, one can model a XOR function:

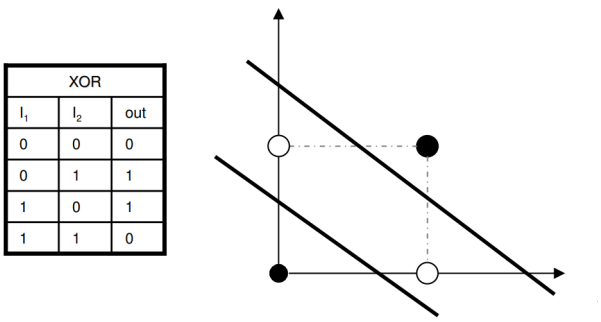


Figure 1.2: *The XOR Problem Font:* Kevin Swingler Lecture Notes.

The XOR problem can not be solved by using a single Perceptron, since it is not a linear problem. However, by using two Perceptrons (which corresponds to the two lines in the figure), one can model the non-linear XOR problem. In this specific example, one can define the uppermost line as the decision boundary of the Perceptron $f_1(x) = \sigma(-x_1 - x_2 + 1.5)$ and the lowermost line as the decision boundary of the Perceptron $f_2(x) = \sigma(x_1 + x_2 - 0.5)$. Considering the Perceptrons f_1 and f_2, we can create a new Perceptron that receives the outputs of those Perceptrons and returns the result of the XOR function. For example, we can define the Perceptron $g(x) = \sigma(x_1 + x_2 - 1.5)$, generating the following results:

x_1	x_2	$f_1(x)$	$f_2(x)$	$g(f_1(x), f_2(x))$
0	0	1	0	0
0	1	1	1	1
1	0	1	1	1
1	1	0	1	0

Table 1.1: *Perceptron output for binary combinations of x_1 and x_2.*

Showing that the non-linear problem can be successfully be solved by the Multilayer Perceptron.

Although the Multilayer Perceptron has the perk of being able to model complex functions, we are still limited by how to model is trained, since it cannot be trained by using the update rule of the traditional Perceptron.

By using a technique known as Backpropagation, the Multilayer Perceptron can be trained using gradient-based update rules, like the Gradient Descent.

1.3.3 Backpropagation

In order to find the weight's gradients of our MLP, one can use Backpropagation, a technique that involves computing those gradients by performing a *Forward Pass* and a *Backward Pass* on the Neural Network.

Forward Pass

The Forward Pass basically consists in computing the input through the network and comparing the output with the expected value using a loss function $\mathcal{L}$. During the forward pass, the output of each layer is stored in memory to be later used in the Backward Pass.

Backward Pass

In the Backward Pass, the gradients of the loss with respect to the weights are calculated to update the network. By using the chain rule, one can start off by calculating the gradients of the last layer of the network, and then use the result to calculate the gradients of the weights of the previous layer, *going on the opposite direction of the Forward Pass.*

Updating Weights

With the gradients of the loss with respect to the weights calculated, now the weights are updated in an iterative process, until a satisfiable loss/accuracy is achieved or new model/dataset tunning is necessary.

1.4 Convolutional Neural Networks

1.4.1 Convolutions

First, it is important to define what a convolution is. Given two discrete one-dimensional signals f and g, their convolution $f * g$ is defined as:

$$(f * g)[n] = \sum_{i=-\infty}^{+\infty} f[n]g[n-i]$$

Given two discrete two-dimensional signals f and g, their convolution $f * g$ is calculated as:

$$(f * g)[m][n] = \sum_{i=-\infty}^{+\infty} \sum_{j=-\infty}^{+\infty} f[m][n] \cdot g[m-i][n-j]$$

In practice, the signal g is represented by a window (or kernel), usually square and of odd size. Thus, we can abstract convolution as the multiplication of a sliding window. The picture below illustrates this process. It is important to note that the window needs to be flipped during the convolution, although this is not illustrated in the picture.

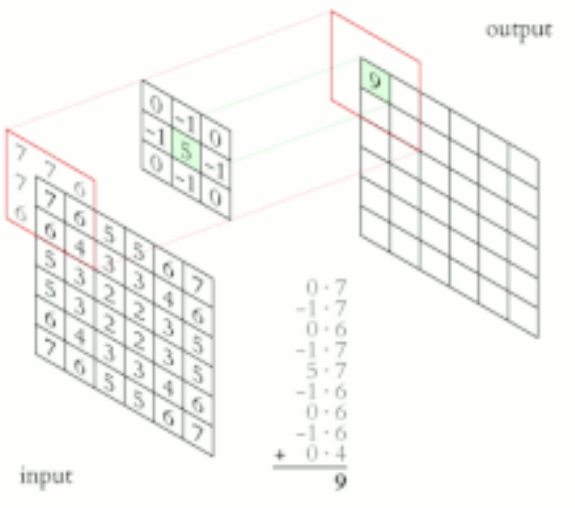


Figure 1.3: *2D Convolution*

This process can be used to apply different filters to images. For example: using a 3×3 window with all weights equal to $\frac{1}{9}$, we can generate a filter that blurs the image (moving average). Below is an example of applying this filter:

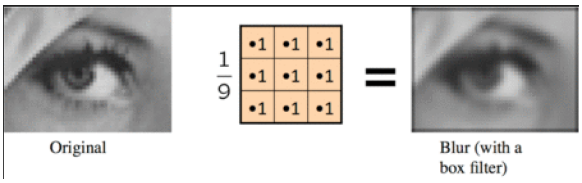


Figure 1.4: *Filtered Image (3x3 Mean Filter)*

It is also important to note that convolution is commonly implemented in machine learning contexts as "cross-correlation," which is a very similar operation but without the flipping of the window. Note that, since the weights are learned in our case, there is no difference. Therefore, in our context, convolution and cross-correlation are synonymous.

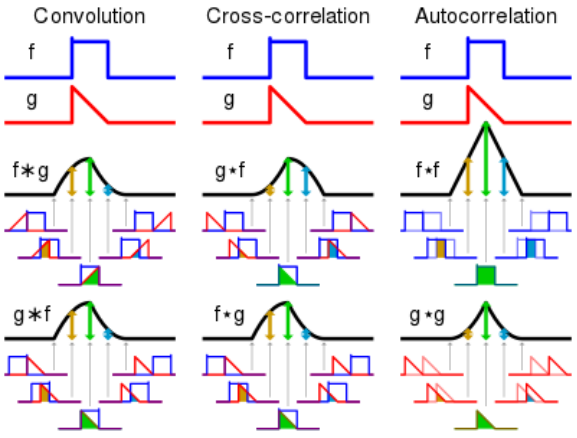


Figure 1.5: *Comparison with Cross-Correlation*

A pertinent question that can be asked is what happens at the edges of the image.

When the window is sliding over them, what happens to the missing pixels? The process of filling in these pixels is called padding. Padding can be done with zeros, the nearest pixel, or not be done at all. Note that when there is no padding, the image decreases in size after convolution.

Another important hyperparameter that can be adjusted is the "stride." This defines how many positions the window is moved at a time. That is: a stride value different from 1 also implies a decrease in image size after convolution.

1.4.2 Convolutional Layer

In order to understand the need for convolutional networks, we have to understand why it's impractical to use fully-connected networks to process images. Let's walk though an example to see that.

Consider a color image with dimensions 512x512. If we were to process this image with a conventional neural network, the input layer would have $3 \cdot 512 \cdot 512 = 786432$ dimensions. Assume a hidden layer with only 128 neurons (which is relatively small). Just between these two layers, there would be 100663296 parameters! This is highly inefficient.

The solution to this problem is to extract features from the image, which will serve as input to the network. These features could include various aspects such as symmetry, black levels, contrast, presence or absence of patterns, etc. All of these features will serve as input to the network. As a result, we can reduce the input layer's dimensionality from several hundred thousand to just a few dozen.

However, a challenge still remains: how do we select these features? We can apply convolutions to the image to calculate interesting features, and these convolutions can be learned alongside the rest of the network! It is important to understand some essential details about these networks before proceeding. Each convolutional layer has three dimensions: height, width, and the number of channels. The input layer typically has one channel for black and white images, or three channels for color images.

Each channel in each convolutional layer combines all the channels from the previous layer. In other words, the "windows" used have weights for all the channels. These windows slide over the data from the previous layer to generate **one channel** in the next layer.

Let us consider an example. Suppose we have a network that processes color images of size 128×128 pixels. This network has 3 convolutional layers with 16, 32, and 64 channels per layer, respectively. Assume a window size of 3 for all layers. In this case, we have:

- First layer: window size is $3 \times 3 \times 3$. With 16 output channels, we will have

$$16 \cdot 3 \cdot 3 \cdot 3 = 432 \text{ parameters.}$$

- Second layer: window size is $3 \times 3 \times 16$. With 32 output channels, we will have

$$32 \cdot 3 \cdot 3 \cdot 16 = 4608 \text{ parameters.}$$

- Third layer: window size is $3 \times 3 \times 32$. With 64 output channels, we will have

$$64 \cdot 3 \cdot 3 \cdot 32 = 18432 \text{ parameters.}$$

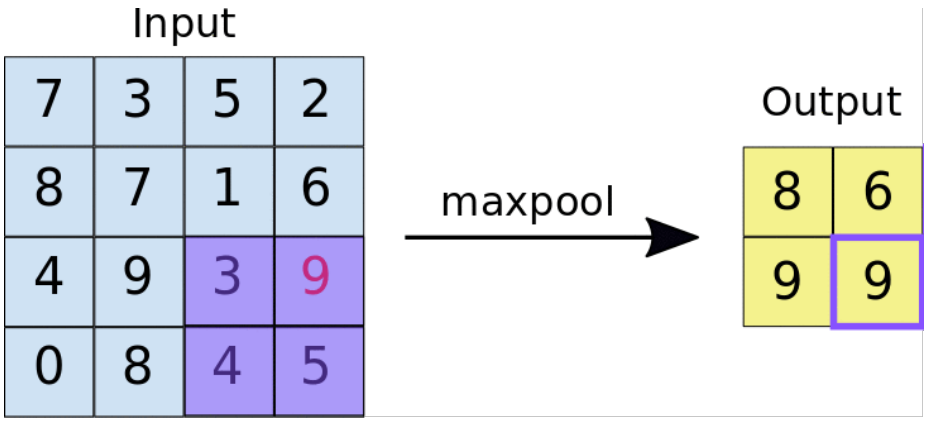
Another important detail is the output dimension of each layer. This depends on whether or not **padding** is used. **Padding** refers to how the layer behaves at the image edges. We can complete the image with zeros, the nearest pixel value, or the pixel value from the opposite edge of the image. If padding is not used, the output dimensions will decrease by $2\lfloor \frac{W}{2} \rfloor$, where W is the window size. For instance, with a window size of 3, each layer will reduce the image size by 2 pixels. If the input is 128×128, the output of the first layer will be 126×126, the second layer will output 124×124, and so on.

1.4.3 Pooling

Remember, each convolution extracts a feature from the image. Therefore, when we perform another convolution using the outputs from the previous layer, we are combining features extracted from the image to compute new features. As a result, deeper layers extract more complex features from the image. For instance, the first layer may extract features like the presence of vertical straight lines, while the tenth layer may extract features like "presence of dog snouts."

Thus, the features involved gradually become less localized and more global (pertaining to the entire image). This is why it is useful to summarize information into smaller dimensions as the network deepens.

To accomplish this, we use "pooling" layers. These layers work similarly to convolutions: windows slide over the data and compute an output based on nearby pixels. However, this time, a function is used to aggregate these data. Common functions include "max" (maximum value) and "avg" (average value).

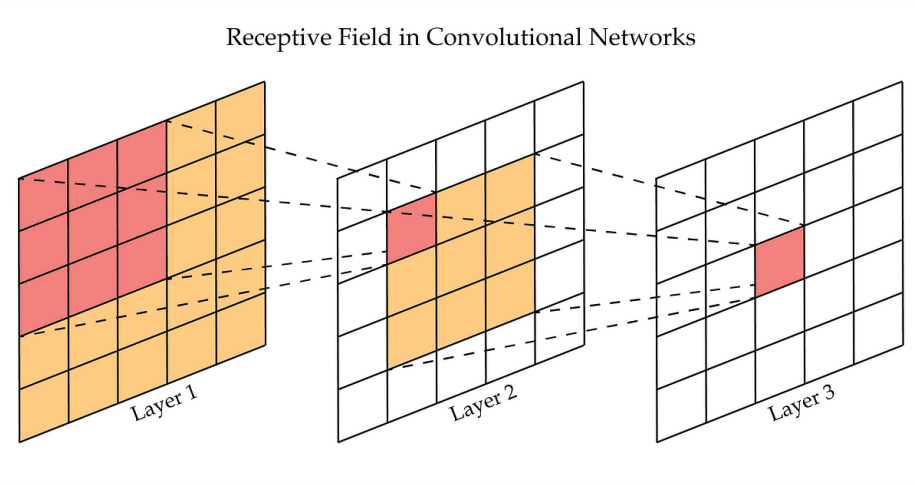


1.4.4 Receptive Field

An important concept for understanding the power of **deep convolutional networks** is the **receptive field**. This concept relates to the power that chained convolutions have.

Consider an input image. Apply a 3×3 convolution to it. Now, apply another 3×3

convolution to the output of the first convolution. Observe this output image. How much information does each pixel contain about its neighbors?



The answer is that each pixel contains information from a region of size 5×5 around it! This is the receptive field of these neurons.

A common misconception is that, since the receptive field of two 3×3 convolutions is 5×5, two 3×3 convolutions have the same **expressive power** as a 5×5 convolution. This **is not** true. A 5×5 convolution has 25 parameters, while two chained 3×3 convolutions only have 18 parameters.

Chapter 2

Local Interpretable Model-agnostic Explanations (LIME)

Local Interpretable Model-agnostic Explanations (LIME) (Ribeiro *et al.*, 2016) is a tool used to visualize the importance of features on the result of a model's prediction. A score is given to each feature fed to the model, making it possible to understand a black-box model's decision based on its inputs.

In this chapter, we will discuss how LIME works, how can one use it on image classification models, we will present our implementation of the method and show some experiments done using the technique.

2.1 How it Works

LIME works by training an interpretable model (1.1.1) to mock the complex *black-box* model over a region of the model's domain. The underlying idea is that while the model's decision boundary across the entire domain may be complex, it tends to be simpler within smaller, localized regions.

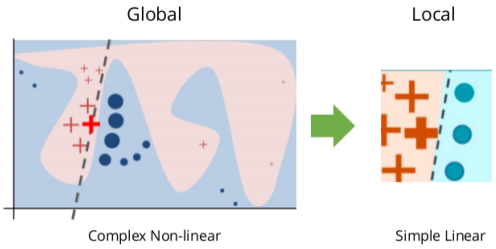


Figure 2.1: *Smaller regions of model's decion boundary tend to have simple, linear behaviour. Font: C3.ai*[1]

[1] https://c3.ai/glossary/data-science/lime-local-interpretable-model-agnostic-explanations/

To train the interpretable model, a single sample is selected and small perturbations are applied to that data point in order to create a dataset, consisting of the sample image and its perturbations. Using the dataset, the interpretable model is trained by minimizing the optimization problem bellow:

$$g^* = \underset{g \in G}{\operatorname{argmin}} \mathcal{L}(f, g, \pi_x) + \Omega(g). \tag{2.1}$$

Where g^* is the final trained interpretable model, G is a set of interpretable models, f is the black-box model, Ω is a function that maps a model's complexity to a number, with higher complexity yielding higher numbers (Used to penalize complexity in models used to mock the black-box model), π_x is a function to penalize samples in the dataset "too far" from the original sample x and $\mathcal{L}$ is a cost function to quantify the similarity between the interpretable model's decisions and the complex model's decisions, defined by the expression bellow:

$$\mathcal{L}(f, g, \pi_x) = \sum_{z, z' \in Z} \pi_x(z)(f(z) - g(z'))^2. \tag{2.2}$$

Where Z is the artificial dataset created from the sample, with datapoints z - a point with the original model's features, and z', a data point that represents a transformation applied to those features, like using a subset of inputs or attributes created by *feature engineering*.

The function π_x can be defined arbitrarily, provided that it satisfies the condition $\pi_x(z) \in [0, 1]$ for all possible samples z in the artificial dataset. It must also ensure that $\pi_x(x) = 1$ for the original sample x, with scores decreasing as data points move further away from x, while higher scores are assigned to points closer to x

With the interpretable model trained, one can analyze its structure to interpret the complex model locally. For example, a Linear Regression model could be interpreted by analyzing its weights. A positive weight value would mean that a feature had a positive impact in the prediction made by the complex model, while a negative weight associated to a feature would represent a negative impact to the model's inference.

Because of the huge ammount of variables in a image classification task, directly using an image consisting of hundreds of thousands of pixels in an interpretable model would yield poor results, since analyzing each individual pixel's contribution to a prediction is not well aligned with a human interpretation of image features. In order to generate more valuable results, a feature engineering technique will be proposed on the next section to model LIME for Image classification tasks.

2.2 LIME on Image Models

In order to use image models on LIME, one must find a way to transform the features into a more valuable and human-aligned metric. To achieve better results, instead of using granular structures such as pixels, we can use *segmentation algorithms* to create *superpixels* of images. A *superpixel* of an image is a set of pixels in a continuous region of the image.

For example, the use of the quickshift segmentation algorithm would yield the following superpixel configuration, where the yellow lines represent borders of superpixels:

(a) *Original Image* **(b)** *Quickshift segmentation*

Figure 2.2: *Superpixel generation using Quickshift*

To train a model using superpixels, a new dataset would be generated by selecting specific superpixels while discarding others. Each data point in this dataset would consist of the original image with certain superpixel regions removed. Below is an example of images from this newly created dataset:

Figure 2.3: *New dataset samples*

These images would then be fed into a complex model to assess the significance of each superpixel in the final prediction. Superpixels that play a more crucial role in the model's decision would cause a greater impact on its output.

The interpretable model would be trained using a binary vector $v \in \{0,1\}^n$ as input, where each element indicates the presence (1) or absence (0) of a superpixel in the sample, with n representing the total number of superpixels. The output of this interpretable model would be the predicted probability of the desired class.

In the scenario where a Linear Regression model is used, each weight related to each superpixel would represent its importance to the prediction of the desired class. A high positive weight means that a superpixel positively impacts a probability value, meaning that the portion of the image occupied by the superpixel represents valuable information for the final prediction and should be aligned with the corresponding class, if the model is well trained.

An image explanation is shown in image 2.4 for the class "Golden Retriever", using a VGG16 model:

(a) *Original Image* **(b)** *LIME Visualization of image*

Figure 2.4: *Visualization of top 15 most important superpixels for "Golden Retriever" ImageNet Class in image*

2.3 Implementation

Using the theory presented in previous sections, an implementation using Pytorch for loading pretrained CNNs, Scikit-Learn for Linear models and Scikit-image for image segmentation using Quickshift.

Our implementation focused on the VGG16 CNN trained on ImageNet, using a Linear Regression model for LIME explanations. We can load both models with the code bellow:

```
1  from torchvision.models import vgg16, VGG16_Weights
2  from sklearn.linear_model import LinearRegression
3
4  model = vgg16(weights=VGG16_Weights.IMAGENET1K_V1)
5  interpretable_model = LinearRegression()
```

The VGG16 model can be seamlessly swapped with other models since LIME operates as a black-box approach, focusing solely on inputs and outputs rather than the model's internal workings.

After the models are loaded, an image to be explained by LIME needs to be loaded. In this example, the image *"dog.jpg"* will be loaded.

```
1  from PIL import Image
2  from torchvision.transforms import ToTensor
3
4  to_tensor = ToTensor()
5
6  image = Image.open("dog.jpg")
7  image = to_tensor(image)
```

Next, we will generate the superpixels of the image using Quickshift, specifying values for `kernel_size`, `max_dist` and `ratio`:

```
import torch
from skimage.segmentation import quickshift

superpixels = torch.tensor(quickshift(
   image.numpy(),
   kernel_size=kernel_size,
   max_dist=max_dist,
   ratio=ratio
))
```

With the superpixel mask created, we will generate a new dataset to train our linear model by modifying the original image, chosing a subset of the superpixel set per sample.

```
# create samples tensor
probability_full_tensor = torch.full((num_samples, num_superpixels),
    probability)
samples = torch.bernoulli(probability_full_tensor).to(dtype=torch.int)

# create masks for image to create dataset

image_masks = []
for sample in samples:
   sample_indexes = torch.nonzero(sample == 1, as_tuple=True)
   mask = torch.isin(superpixels, torch.cat(sample_indexes))
   image_masks.append(mask)

image_masks = torch.stack(image_masks)

# apply each mask to the image, creating a new image per mask
dataset = image.unsqueeze(0) * image_masks.unsqueeze(1).expand(-1, 3, -1, -1)
```

Now, we extract predictions of the model over each image in the dataset, using a predefined class to visualize called `explained_class`:

```
model.eval()
with torch.no_grad(): # avoid gradient computation
   predictions = model(dataset)[:, explained_class]
```

As the final step before training the interpretable model, we compute the distance of each sample in the new dataset from the original image. This ensures that the linear model training accounts for the variation between the original image and its modified versions.

```
cosine_similarity_evaluator = torch.nn.CosineSimilarity(dim=1)

full_ones_samples = torch.ones_like(samples).float()
cosine_similarity = cosine_similarity_evaluator(full_ones_samples, samples)

sample_weight = torch.sqrt(torch.exp(- cosine_similarity**2 / kernel_samples
    **2))
```

Here, the sample weight function can be defined as the expression

$$\texttt{sample\_weight} = \sqrt{\exp(\texttt{cosine\_similarity}^2/\texttt{kernel\_samples}^2)} \tag{2.3}$$

where `cosine_similarity` is the cosine similarity between the samples tensor and a all-ones tensor and `kernel_samples` is a hyperparameter in the interval $(0, 1)$.

Finally, the linear model can now be trained:

```
interpretable_model.fit(
   X=samples,
   y=predictions,
   sample_weight=sample_weight
)
```

By extracting the interpretable model's weights, we can find the most important regions in the image:

```
from numpy import argsort

weights = interpretable_model.coef_
top_features = torch.tensor(argsort(coefs)[-num_selected_weights:]) # select
    top "num_selected_weights" weights

masked_superpixels = torch.isin(superpixels, top_features)
visualization_image = image * masked_superpixels
```

Generating a LIME visualization for our desired image.

2.4 Experiments

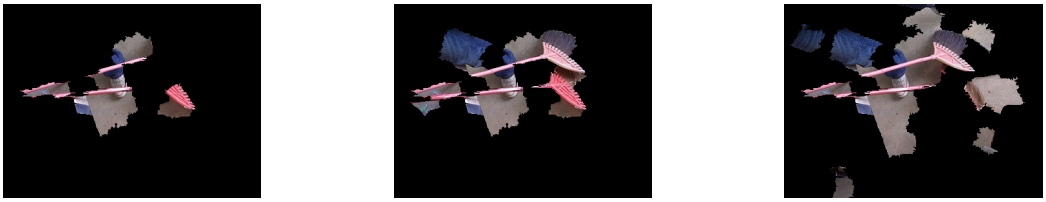
In this section, different hyperparameter configurations on different images using LIME were explored. The experiments were conducted on the VGG16 CNN architecture trained on ImageNet. The Quickshift algorithm was used for the images feature engineering. The distance function π_x used in the experiments is the same as the one used in 2.3.

We will examine instances where the complex model failed to predict the expected class, as well as cases where it accurately identified the correct label.

2.4.1 Successful Predictions

(a) *Original Image* **(b)** *Superpixel segmentation*

Figure 2.5: *Broom image with superpixel segmentation*



(a) *Top 5 Superpixels, Sampling probability of 40% and distance kernel of 0.5* **(b)** *Top 10 Superpixels, Sampling probability of 50% and distance kernel of 0.5* **(c)** *Top 15 Superpixels, Sampling probability of 80% and distance kernel of 1*

Figure 2.6: *Broom Visualizations with different hyperparameter configurations*

(a) *Original Image* **(b)** *Superpixel segmentation*

Figure 2.7: *Golden Retriever image with superpixel segmentation*

(a) *Top 5 Superpixels, Sampling probability of 40% and distance kernel of 0.5*

(b) *Top 10 Superpixels, Sampling probability of 60% and distance kernel of 0.75*

(c) *Top 15 Superpixels, Sampling probability of 80% and distance kernel of 1*

Figure 2.8: *Golden Retriver Visualizations with different hyperparameter configurations*

Our experiments confirm that the complex model aligns well with human judgment, primarily focusing on highly distinctive features, such as the head of a broom and the head and body of a Golden Retriever.

Hyperparameters can significantly influence the outcome of a LIME visualization. For instance, in the broom example, one broom head appears in the Top 5 superpixels but is absent from the Top 15 superpixels when using a different hyperparameter configuration. The same goes for the dog's visualization, with its frontal legs present in the Top 5 but absent on the Top 15 of another hyperparameter configuration.

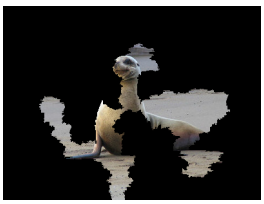
2.4.2 Failed Predictions

(a) *Original Image*

(b) *Superpixel segmentation*

Figure 2.9: *Sea Lion image with superpixel segmentation*

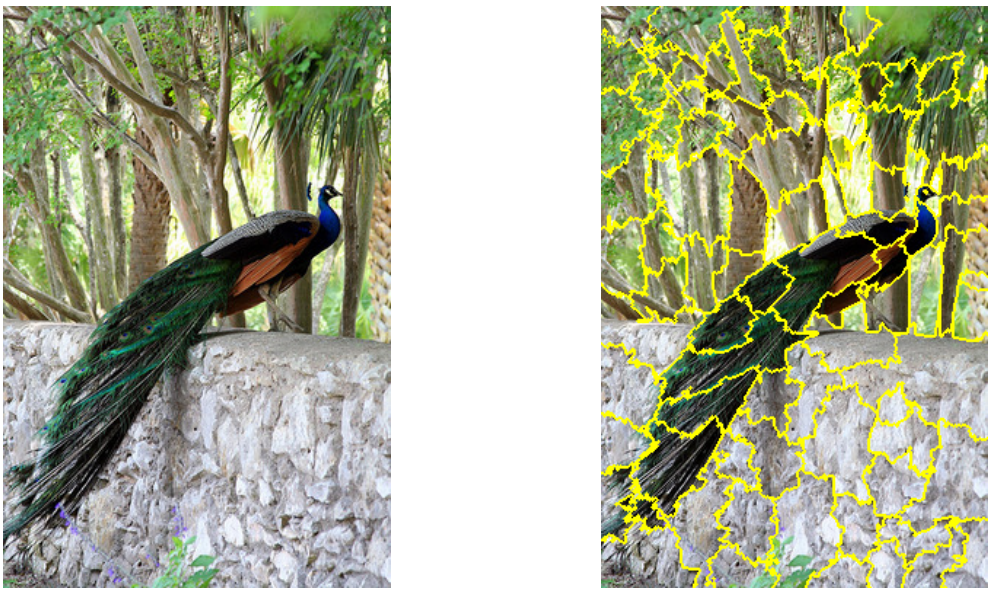
(a) *Top 5 Superpixels, Sampling probability of 40% and distance kernel of 0.5*



(b) *Top 10 Superpixels, Sampling probability of 60% and distance kernel of 0.75*

(c) *Top 15 Superpixels, Sampling probability of 80% and distance kernel of 1*

Figure 2.10: *Polar Bear Visualizations with different hyperparameter configurations (Expected: Sea Lion)*



(a) *Original Image* **(b)** *Superpixel segmentation*

Figure 2.11: *Peacock image with superpixel segmentation*

(a) *Top 5 Superpixels, Sampling probability of 40% and distance kernel of 0.5* **(b)** *Top 10 Superpixels, Sampling probability of 50% and distance kernel of 0.75* **(c)** *Top 15 Superpixels, Sampling probability of 80% and distance kernel of 0.75*

Figure 2.12: *Mosquito Net Visualizations with different hyperparameter configurations (Expected: Peacock)*

Analyzing the failed experiment cases reveals that the model may have misinterpreted a peacock's tail as a mosquito net. Additionally, in the sea lion image, the complex model appears to focus on both the sea lion and the sandy ground, likely leading to a misclassification due to the shared habitat of polar bears and sea lions.

Chapter 3

GradCAM

3.1 Introduction

One of the most proeminent model-specific methods to acquire explanations for classification with CNNs is GradCAM. The intuition for the method is simple. At the last conovlutional layers, we have several channels that represent each a different feature. Those features are used by the next part of the network to produce the final output. If we want to know which parts of the image are being more useful to the network, we can look at the feature maps and observe which parts of the image are generating the signal used by the rest of the network.

The problem with this approach is that the features have informations about all the output classes. How we know what features are more important to the decision? The idea behind GradCAM is to **average the feature maps weighted by the gradient** of each channel with respect to a specific class.

However, this will still highlight the regions that have a negative influence to the decision. To filter out those regions, the result is passed through ReLU. The result is a coarse heatmap of the image highlighting important regions.

The formula for the heatmap with regard to the class c is:

$$H = \text{ReLU}(\sum_k \alpha_k^c A^k) \tag{3.1}$$

Where α_k^c, the weight of the k-th feature map for the class c, is defined as:

$$\alpha_k^c = \frac{1}{Z} \sum_i \sum_j \frac{\partial y^c}{\partial A_i^k j} \tag{3.2}$$

Where A^k is the k-th feature map, y^c is the output for the c class, and Z is the number of neurons in each map.

To visualize the process, we will use the ResNet-18 He *et al.*, 2015 trained on the ImageNet dataset, with the weights available at PyTorch. We will use as input two images of airplanes.

Figure 3.1: *Input images*

The first step is to store the activations of the last convolutional layer of the network for each of the images. For this network, we have 512 channels at the last layer.

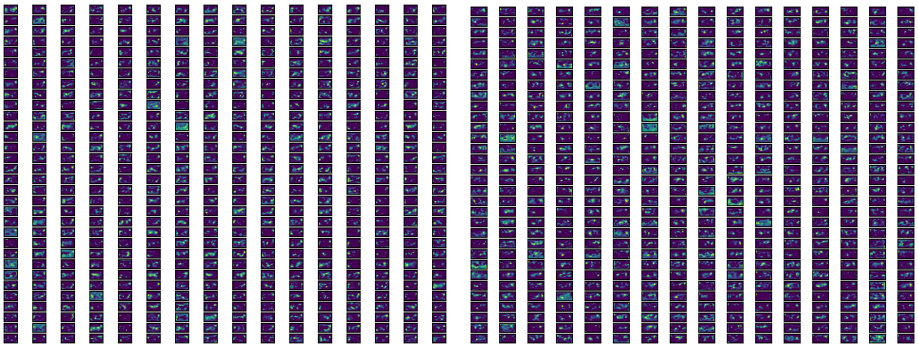


Figure 3.2: *Activations of last convolutional layer*

The next step is to calculate the weight of each channel, according to **eq:alpha**. We can then visualize the weights as a heatmap (each vertical line represents a channel):

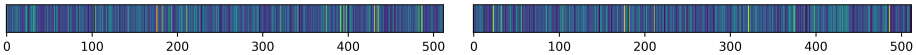


Figure 3.3: *Weight of each channel of the last convolutional layer*

By calculating the average of the activations found at **fig:actiovations** weighted by , we can achieve a heatmap highlighting important regions of each image:

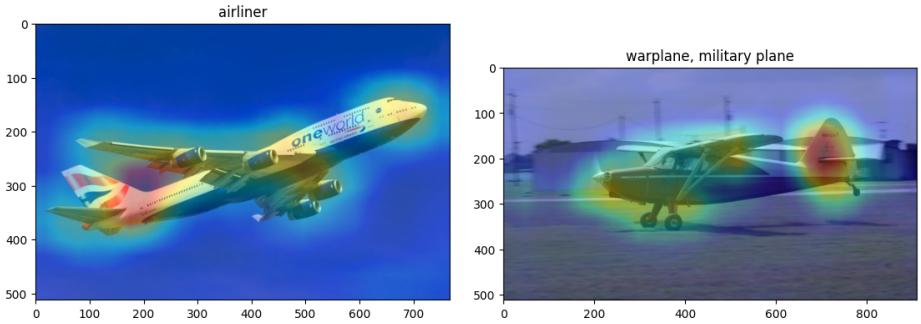


Figure 3.4: *Result of GradCAM*

3.2 Guided Backpropagation

Guided Backprogation is one of the most proeminent way of creating saliency maps. Following the same idea behind the application of the ReLU function at **eq:heatmap**, we filter out negative gradients through the network, allowing the visualization of only positive influences.

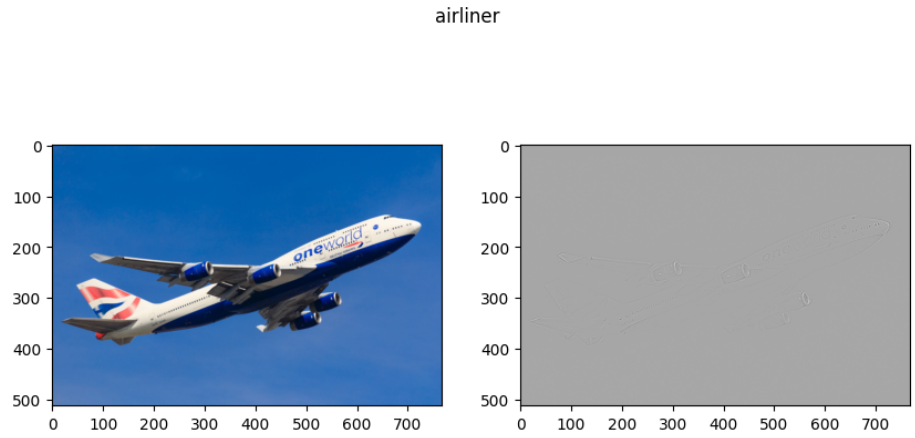


Figure 3.5: *Results of Guided Backpropation*

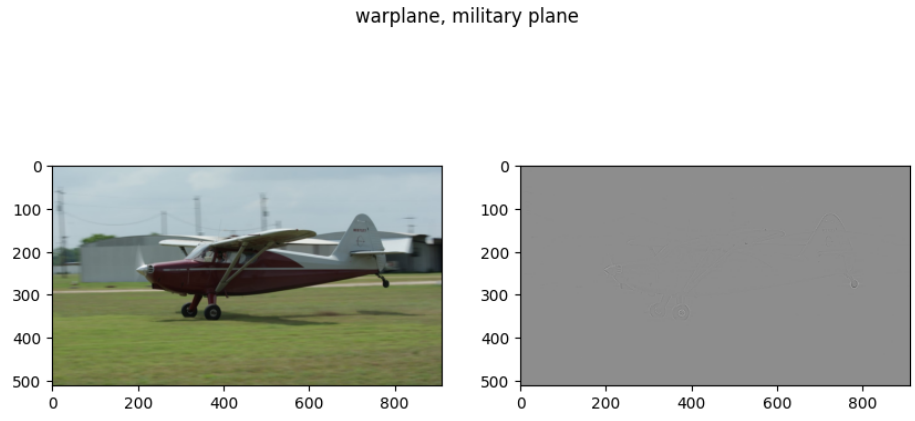


Figure 3.6: *Results of Guided Backpropation*

3.3 Guided GradCAM

The output of GradCAM has the dimensions of the last convolutional layer of the network, and has to be upsampled to be overlayed on top of the input image. This results in a very coarse heatmap, with rough borders and lost details. To solve this, we can multiply (pixel by pixel) the heatmap with other simpler method, as guided-backpropagation.

Figure 3.7: *Results of Guided GradCAM*

3.4 Implementation

All these images were generated using the Python library Pytorch. The implementation used a feature of the library called hooks. Hooks are functions that can be connected to the network. Those hooks will be called either at the forward pass (for forward hooks), or at the backward pass (for backward hooks). We used those hooks to register the activations and the gradient at GradCAM, and to filter out negative gradients at Guided Backpropagation.

Program 3.1 GradCAM Class

```
1   class GradCAM:
2
3   def __init__(self, model):
4     self.activations=None
5     self.gradients=None
6     self.model=model
7     self.model.eval()
8     def forward_hook(module, input, output):
9       self.activations = output
10
11    def backward_hook(module, input, output):
12      self.gradients = output[0]
13
14
15    model._modules['layer4'].register_forward_hook(forward_hook)
16    model._modules['layer4'].register_backward_hook(backward_hook)
17
18
19  def forward(self, im):
20    self.model.zero_grad()
21    output = self.model(im)
22    label = output.argmax().item()
23    output[0,label].backward()
24    alpha = self.gradients.squeeze(0).mean(dim=(1, 2))
25    print(categories_MNIST[label])
26    heatmap = (self.activations.squeeze(0) * alpha.view(-1, 1, 1)).sum(dim=0)
27    heatmap = transforms.Resize(im.shape[2:])(heatmap.unsqueeze(0))/heatmap.max
          ()
28    return (torch.clamp(heatmap, min=0).cpu(), categories_MNIST[label])
```

Program 3.2 GradCAM Class

```
1
2
3   class GuidedBackPropagation:
4     def __init__(self, model):
5       self.activations=None
6       self.gradients=None
7       self.model=model
8       self.model.eval()
9
10      def backward_hook(module, input, output):
11        if isinstance(module, torch.nn.ReLU):
12          return (torch.clamp(input[0], min=0),)
13
14
15
16      for i, module in enumerate(model.modules()):
17        if isinstance(module, torch.nn.ReLU):
18          module.inplace = False
19          module.register_full_backward_hook(backward_hook)
20
21
22    def forward(self, im):
23      self.model.zero_grad()
24      output = self.model(im)
25      label = output.argmax().item()
26      print(label)
27      output[0,label].backward()
28      ret = im.grad.clone()
29      ret = ret.squeeze(0).sum(0)
30      ret = (ret - ret.min())/(ret.max() - ret.min())
31      return (ret.cpu(), categories_MNIST[label])
```

Following is an example of the utilization of those classes:

Program 3.3 Utilization of the classes

```
from util import *
from GradCam import *
from glob import glob

model = torchvision.models.resnet18(weights=torchvision.models.
    ResNet18_Weights.IMAGENET1K_V1).to(device).eval()
inp_trans = transforms.Compose([transforms.Resize(512), transforms.ToTensor(),
     transforms.Normalize(mean=[0.485, 0.456, 0.406], std=[0.229, 0.224,
    0.225]),])
view_trans=transforms.Compose([transforms.Resize(512), transforms.ToTensor()])


files=glob('./imagens/*')
images=[Image.open(f) for f in files]
tensors = [inp_trans(im).unsqueeze(0).to(device) for im in images]
view_tensors = [view_trans(im).unsqueeze(0).to(device) for im in images]
print(len(tensors))

gr = GradCAM(model)
heatmaps=[gr.forward(tensor) for tensor in tensors]

gb = GuidedBackPropagation(model)
bp = [gb.forward(tensor.requires_grad_()) for tensor in tensors]
```

3.5 Experiments

In order to illustrate the usefulness of GradCAM, we conducted some experiments inducing variations in the input image. Those variations were made in three ways: noise, brightness and contrast. The GradCAM was calculated for the top class and the original class.

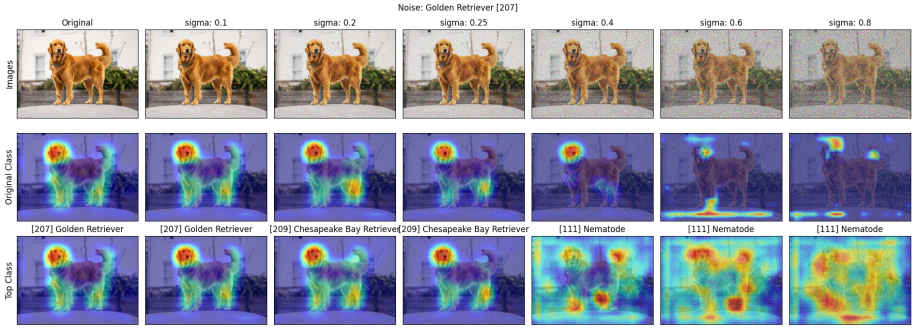


Figure 3.8: *Noise variation*

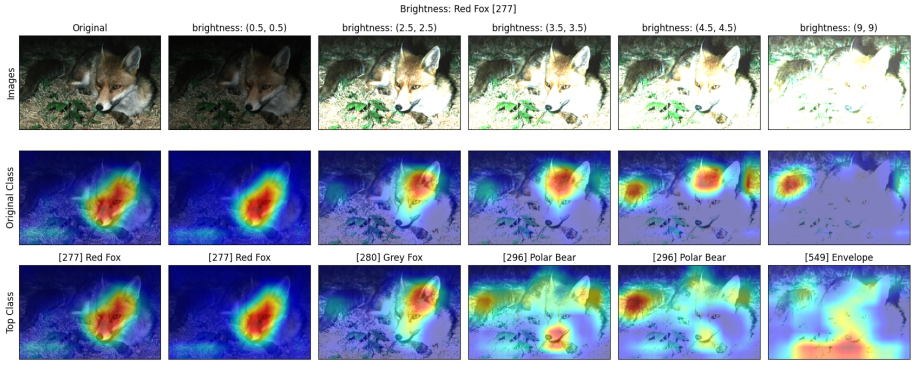


Figure 3.9: *Brightness variation*

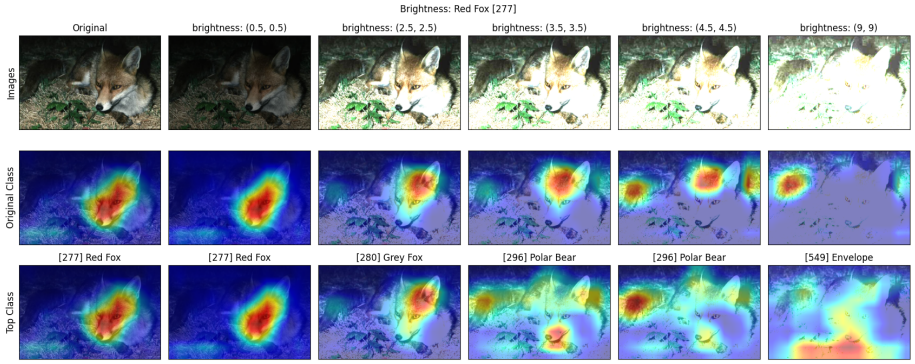


Figure 3.10: *Contrast variation*

In the three figures, it's possible to observe how the GradCAM outputs become dispersed as the image degrades. At the first experiment, for instance, the focus of the network gets more dispersed as the noise increases. At the maximum level of noise, the GradCAM outputs a heatmap focusing heavily on the ground and on the background.

The experiments also gives us some insights in what can be causing wrong outputs. At the red fox experiment, at the last stage of the stage of degradation, the output is "envelope". This seems absurd and completely nonsensical, but by examining the heatmap, it's clear that the network is being mislead by the vast amount of completely white regions at the image.

Chapter 4

Feature Visualization

4.1 The Optimization Problem

Feature Visualization is a technique that involves maximizing values of neurons, sets of neurons or even layers of a Neural Network in order to understand concepts learned by the model. By maximizing a neuron's value, we can better understand what set of features each part of our network is learning to capture and verify if the network is aligned with human judgement. In simple terms, Feature Visualization can be described by the following optimization problem:

$$\text{img}^* = \underset{\text{img}}{\text{argmax}}\, h_{n,x,y,z}(\text{img}) \tag{4.1}$$

where h represents the activation of a neuron, img is the input of the network, x and y represent the spatial position of the neuron, n is the layer of the network and z is the channel index. This expressions represents the problem of maximizing a value of a single neuron. For a set of neurons, the problem can be described as the formula:

$$\text{img}^* = \underset{\text{img}}{\text{argmax}} \sum_{(n,x,y,z) \in A} h_{n,x,y,z}(\text{img}) \tag{4.2}$$

where $A \subseteq \mathbb{N}^4$ is a set of combinations of network layer, channel index and spatial position vectors.

In order to find a solution to the optimization problem, we can use the Gradient Descent technique presented in Chapter 1. However, instead of minimizing an optimization problem, we are looking for a solution that maximizes Equation 3.1 or Equation 3.2. Therefore, instead of subtracting the partial derivative term of Equation 1.2 we just need to add the partial derivative multiplied by the *learning rate* (We will call the technique *Gradient Ascent*). Thus, the following formula is derived for Feature Visualization:

$$\text{img}_{t+1} = \text{img}_t + \eta \sum_{(n,x,y,z) \in A} \frac{\partial h_{n,x,y,z}(\text{img}_t)}{\partial \text{img}_t} \tag{4.3}$$

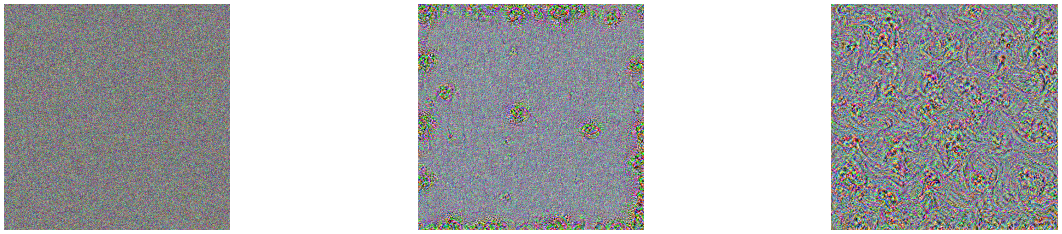
where t is the iteration t of the algorithm.

The initial image img_0 can be defined in two ways:

- A completely random initialization, with $\text{img}_0(c, x, y) = U[0, 1]$ for each channel c and image spatial positions x and y.
- A user defined image, normally a real world image.

The choice between these two approaches depends on the researcher's ultimate objective when using the algorithm. Providing an initial non-random image allows researchers to focus on specific features within the image and examine their effects on a particular set of neurons. In contrast, using a random image may be better suited for discovering unknown features.

Like most optimization problems, the Feature Visualization problem doesn't strictly have a single optimal solution, but rather multiple viable solutions that maximize the given set of neurons. For example, a set of seemingly random images may activate a neuron fully while at the same time an image of a cute dog may also activate this same neuron to its maximum value. Directly applying *Gradient Ascent* to random images without additional techniques often produces images that poorly align with human perception. This outcome occurs not because human-aligned images fail to maximize the neuron's value, but because solutions comprising seemingly random pixels are closer to the image's initial state.



(a) *Original Image*

(b) *Feature Visualization of Convolution layer 10 of VGG16*

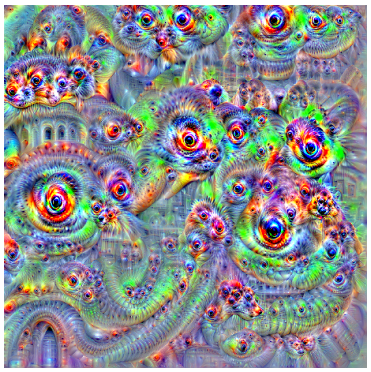
(c) *Feature Visualization of Pug (ImageNet index 254) neuron of VGG16*

Figure 4.1: *Feature Visualization images using solely Gradient Ascent. Little human-recognizable features are present in the resulting images*

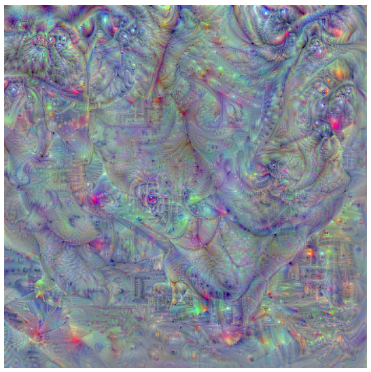
To identify human-aligned features in our models, we need effective techniques for generating images that closely correspond to those features. A closer examination reveals that Feature Visualization using solely Gradient Ascent typically produces features that occupy only small portions of the final image.

To expand these features across a larger area and create more intricate forms, we can downscale the image and apply the Gradient Ascent step. Once the downscaled image has been refined, it can be upscaled by a specific factor, and Gradient Ascent can be reapplied.

Repeating this process until the image reaches its original size enables the generation of more detailed and compelling results, as demonstrated below:



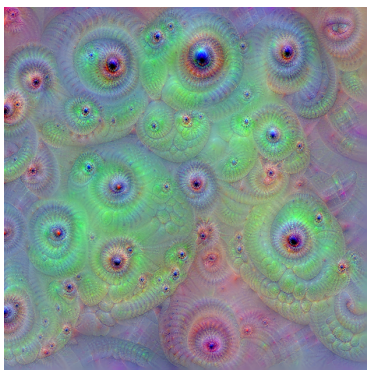
(a) *Feature Visualization of Convolution layer 10 of VGG16*



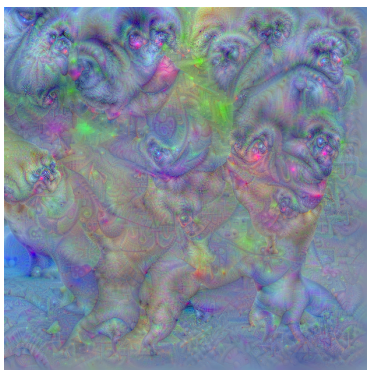
(b) *Feature Visualization of Pug (ImageNet index 254) neuron of VGG16*

Figure 4.2: *By employing Gradient Ascent across multiple image scales, we achieve results that align more closely with human perception. In Subfigure 4.2a, eye-like structures frequently emerge in the generated images, while in Subfigure 4.2b, fur-like textures (top-left portion of image) reminiscent of a Pug's facial coat are present.*

Some features aligned with human perception can now be visible by applying this method on random images. However, high frequency noise is still very present in the generated images, a feature not very common in real life pictures. In order to mitigate the high frequency noise, we may apply regularization techniques to the random image on every iteration of the Feature Visualization algorithm. For example, by applying Gaussian Blur, high frequency regions are dimished because Gaussian blur is a low-pass filter.



(a) *Feature Visualization of Convolution layer 10 of VGG16*



(b) *Feature Visualization of Pug (ImageNet index 254) neuron of VGG16*

Figure 4.3: *Feature Visualization images generated by applying Gaussian Blur besides multi-scale Gradient Ascent. This technique applied on Layer 10 predictably yields an image with much lower frequencies than 4.2a. Also, patterns like snouts or eyes are still present with round and circular features. As for Subfigure 4.3b, pug-like faces are very present in the generated image, unlike what is present in Subfigure 4.2b.*

By incorporating regularization, as shown in Figure 4.3, we generated more human-aligned images with clearly recognizable features. Using the techniques found in this section, an implementation will be proposed in the next section.

4.2 Implementation

Following the theory explored in the last section, an implementation in Python using the library Pytorch will be proposed in this section.

For our experiments, we will use a VGG16 CNN architecture trained on ImageNet. The model is made available by the library Torchvision. We can import the model with pretrained weights with the code bellow:

Program 4.1 Loading pretrained VGG16 model

```
1  from torchvision.models import vgg16, VGG16_Weights
2
3  model = vgg16(weights=VGG16_Weights.IMAGENET1K_V1)
```

Now, we need to define a way to track activations of certain layers or neurons of the network in order optimize the Feature Visualization images. In Pytorch, we can define *Hooks*, which will update every time a forward pass is executed in the network. In our implementation, we defined the following Hook class to track activations:

Program 4.2 Hook Class

```
1  class Hook:
2      def __init__(self, model_layer: Sequential):
3          self.hook = model_layer.register_forward_hook(self.hook_fn)
4
5      def hook_fn(self, _, input: Tensor, output: Tensor):
6          self.input = input
7          self.output = output
8
9      def close(self):
10         self.hook.remove()
```

Where activations can be retrieved after each forward pass by checking the hook's output:

Program 4.3 Hook Usage

```
1  model = vgg16(weights=VGG16_Weights.IMAGENET1K_V1)
2  hook = Hook(model.features[22])
3  model(image) # compute forward pass in model
4  activations = hook.output # retrieve model.features[22] activations
```

Using hooks, the Gradient Ascent step can now be implemented:

Program 4.4 Gradient Ascent Step with Normalization

```
1   def gradient_ascent_step(
2       image: torch.Tensor,
3       model: torch.nn.Module,
4       hook: Hook,
5       learning_rate: float
6       ) -> torch.Tensor:
7
8       blur = GaussianBlur(kernel_size=7, sigma=0.9)
9       image = blur(image)
10
11      image.requires_grad_()
12
13      model(image)
14      activations = hook.output
15
16      loss = (activations**2).sum()
17
18      loss.backward()
19      normalized_grad = (image.grad - image.grad.mean()) / image.grad.std()
20
21      image.grad.zero_()
22      image = image.detach()
23      image += learning_rate * normalized_grad
24
25      return image
```

The algorithm begins by applying a blur to the input image in lines 8 and 9 to regularize the results, producing a smoother and less noisy output, as discussed in the previous section. Next, gradient computation is enabled for the image in line 11, a necessary step for handling PyTorch tensors. Following this, the activations in the layer are calculated and aggregated into a single value in lines 13 to 16, which is then used to compute the gradient. Finally, the gradient is computed in line 18, normalized in line 19, and the image is updated in line 23, completing one step of Gradient Ascent.

Now, we can compute Gradient Ascent through mulitple iterations with multiple image scales, until we are satisfied with the results.

Program 4.5 Feature Visualization

```
1   def feature_visualization(
2       image: torch.Tensor,
3       model: torch.nn.Module,
4       model_layer: torch.nn.Module,
5       pyramid_levels: int,
6       growth_rate: float,
7       steps: int,
8       learning_rate: float
9   ) -> torch.Tensor:
10
11  resizer = Resizer(pyramid_levels, image)
12
13  hook = Hook(model_layer, backward=False)
14
15  for pyramid_level in range(pyramid_levels):
16      image = resizer.resize(image, pyramid_level)
17
18      for _ in range(steps):
19          image = gradient_ascent_step(image, model, hook, learning_rate)
20
21  image = image.clamp(0, 1)
22  return image
```

In simple terms, we create an *Resizer* object responsible to rescale the image on every step of the outer for loop (line 16). Then, we bind the hook to the desired model layer in line 13 and process the image for each image size computed in line 16 and for the desired amount of Gradient Ascent steps for each iteration. After the process, the image is clamped back to the interval $[0, 1]$, since the Gradient Ascent steps can bring the generated image pixels out of the interval.

4.3 Experiments

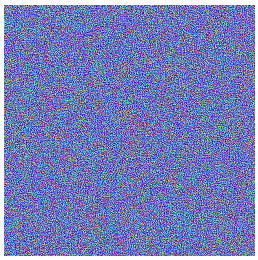
In this section, we present a series of experiments conducted to evaluate and demonstrate the effectiveness of feature visualization techniques. These experiments are designed to highlight the interpretability of the learned features in deep neural networks, analyze their behavior across different layers and architectures, and explore their potential applications. Through these experiments, we aim to provide both qualitative and quantitative insights into the representational power and limitations of feature visualization.

The experiments in this section are performed using the VGG16 convolutional neural network (CNN) architecture, pre-trained on the ImageNet dataset. The hyperparameters will be carefully optimized to produce results that closely align with human interpretability and understanding.

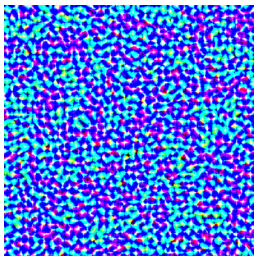
4.3.1 Layer-Wise Visualization

To create feature visualization images that capture the most significant features of an entire layer, we compute the gradient of the sum of all neuron activations within that layer. This approach highlights the collective importance of features across the layer. As an initial example, feature visualization images were generated for Layer 10 of the VGG16 architecture, as shown in Figures 4.2a and 4.3a. Next, we will extend this exploration to various layers of the network to analyze how features evolve throughout the architecture.

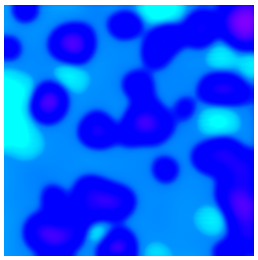
Initial Layers (1-5)



(a) *No Multiscaling, No Blurring, learning_rate* = 0.4

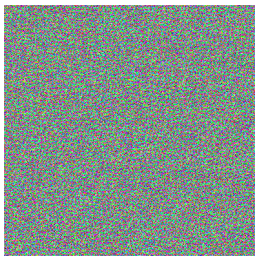


(b) *4 Layer Multiscaling, No Blurring, learning_rate* = 0.04

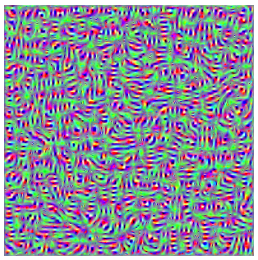


(c) *4 Layer Multiscaling, Blurring, learning_rate* = 0.04

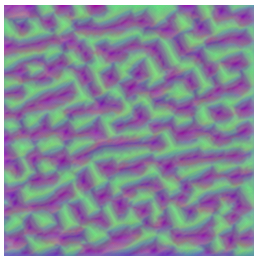
Figure 4.4: *Layer 1*



(a) *No Multiscaling, No Blurring, learning_rate* = 0.4

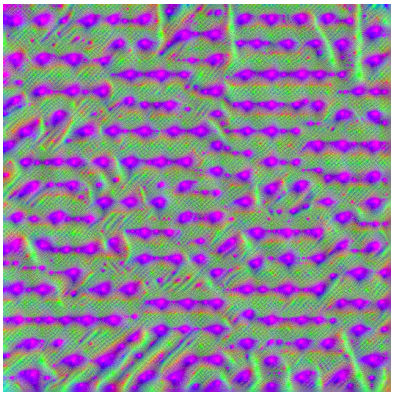


(b) *4 Layer Multiscaling, No Blurring, learning_rate* = 0.04

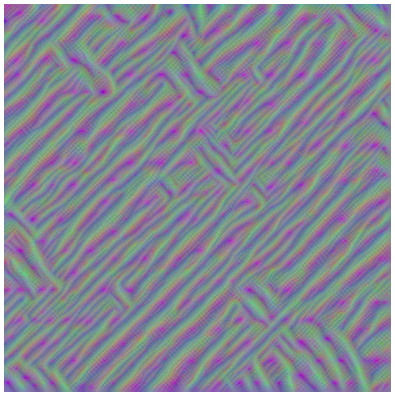


(c) *4 Layer Multiscaling, Blurring, learning_rate* = 0.04

Figure 4.5: *Layer 3*



(a) *4 Layer Multiscaling, Blurring, learning_rate* = 0.22

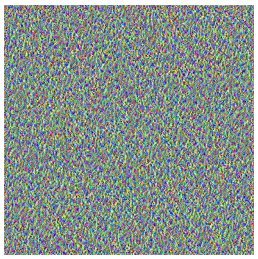


(b) *4 Layer Multiscaling, Blurring, learning_rate* = 0.08

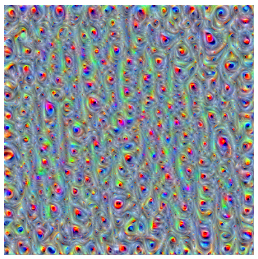
Figure 4.6: *Layer 5*

Intuitively, the initial layers of a CNN capture more simple features from images, like edges and simple formats. By analyzing the generated images, one can clearly notice the complexity enhancement throughout the layers of the network. For layer 1 (Figure 4.4), the generated images mainly focus on maximizing a color value similar to blue, probably correlated to the color distribution of the images on ImageNet. However, in layers 3 (Figure 4.5) and 5 (Figure 4.6) some patterns are already noticeable in the figures, with patterns similar to lines and curves being created.

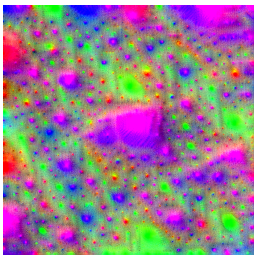
Intermediary Layers (6-9)



(a) *No Multiscaling, No Blurring, learning_rate* = 0.4

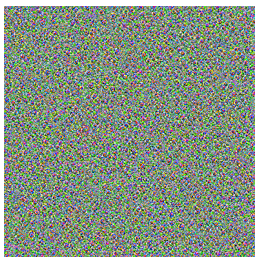


(b) *4 Layer Multiscaling, No Blurring, learning_rate* = 0.04

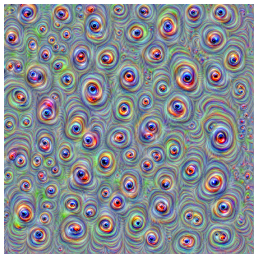


(c) *6 Layer Multiscaling, Blurring, learning_rate* = 0.22

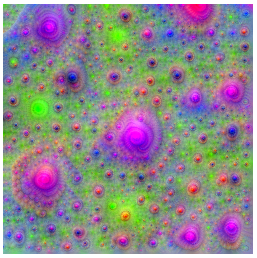
Figure 4.7: *Layer 6*



(a) *No Multiscaling, No Blurring, learning_rate* = 0.4

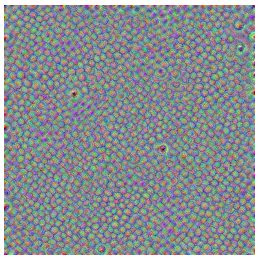


(b) *4 Layer Multiscaling, No Blurring, learning_rate* = 0.04

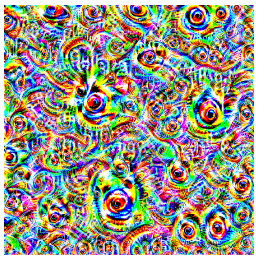


(c) *6 Layer Multiscaling, Blurring, learning_rate* = 0.22

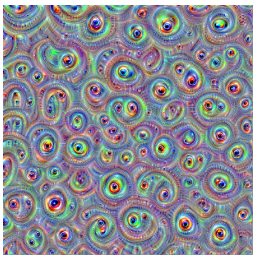
Figure 4.8: *Layer 8*



(a) *No Multiscaling, Blurring, learning_rate* = 0.4



(b) *4 Layer Multiscaling, No Blurring, learning_rate* = 0.4

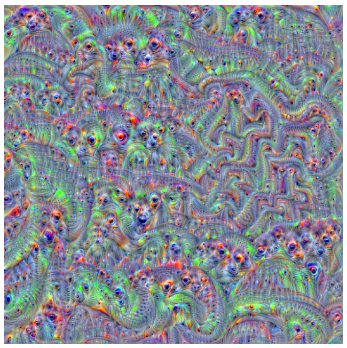


(c) *4 Layer Multiscaling, No Blurring, learning_rate* = 0.04

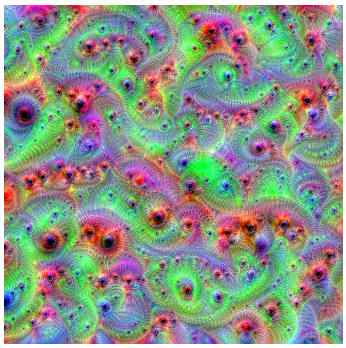
Figure 4.9: *Layer 9*

From our experiments, we can observe that complex features emerge from intermediary layers, with eye-like patterns appearing in Figures 4.8 and 4.9, probably related to the huge amount of animal pictures in the training dataset. Also, curves and circles are way more present in this layer range, showing that the network learned throughout its layers to maximize its activation to more *curvy patterns*, since real life pictures tend to have less linear line segments.

Final Layers (10-13)



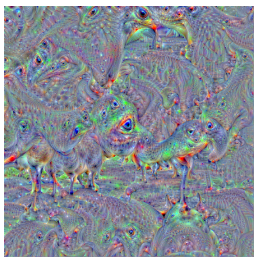
(a) *4 Layer Multiscaling, No Blurring, learning_rate* = 0.04



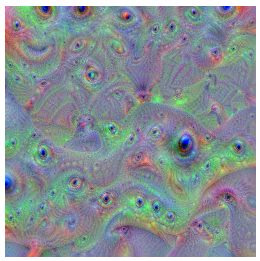
(b) *4 Layer Multiscaling, Blurring, learning_rate* = 0.4

Figure 4.10: *Layer 10*

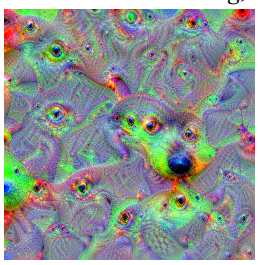
(a) *4 Layer Multiscaling, No Blurring, learning_rate* = 0.4



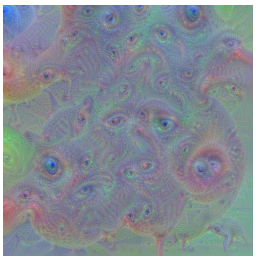
(b) *4 Layer Multiscaling, No Blurring, learning_rate* = 0.04



(c) *4 Layer Multiscaling, Blurring, learning_rate* = 0.22

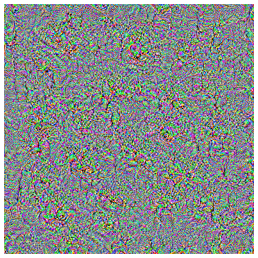


(d) *4 Layer Multiscaling, Blurring, learning_rate* = 0.4

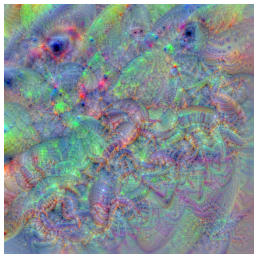


(e) *6 Layer Multiscaling, Blurring, learning_rate* = 0.1

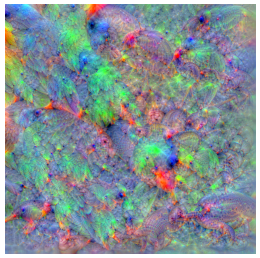
Figure 4.11: *Layer 12*



(a) *No Multiscaling, No Blurring, learning_rate* = 0.4



(b) *6 Layer Multiscaling, Blurring, learning_rate* = 0.08



(c) *6 Layer Multiscaling, Blurring, learning_rate* = 0.1

Figure 4.12: *Layer 13*

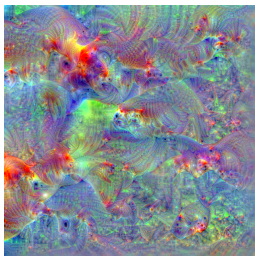
On the final layers of the network, it is possible to see that not only complex shapes are being generated but complete forms that appear like animals are also present. For example, in Figure 4.11, in image (d), a pattern very close to a dog's muzzle is present close to the center of the image. Also, for image (c) in the same layer, patterns like birds faces are also recognizable in the picture.

4.3.2 Class Visualization

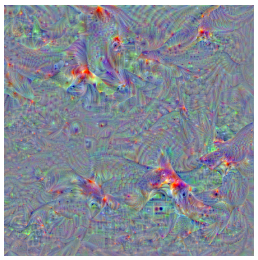
By maximizing the activation of a neuron corresponding to a specific class, we can generate images that visually represent the model's understanding of each class. This approach helps assess whether the model accurately captures the defining characteristics of each class, rather than relying on unrelated noisy biases potentially present in the dataset.

An initial example of this technique is illustrated in Figures 4.2b and 4.3b. Next, we will examine various classes using different hyperparameter configurations to explore the method further.

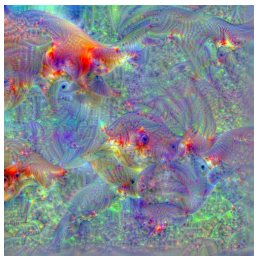
Class 1 - Goldfish



(a) *6 Layer Multiscaling, Blurring, learning_rate* = 0.1



(b) *4 Layer Multiscaling, No Blurring, learning_rate* = 0.04

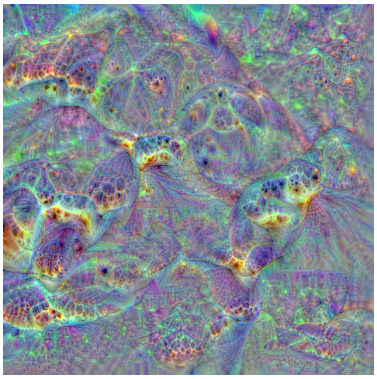


(c) *6 Layer Multiscaling, Blurring, learning_rate* = 0.09

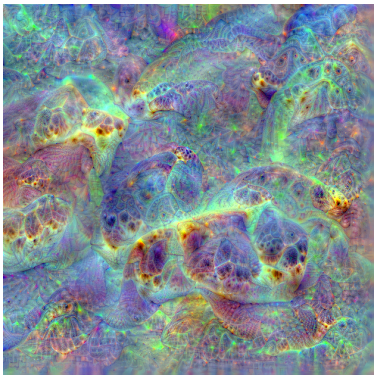
Figure 4.13: *Goldfish Class Feature Visualization for VGG16*

It is possible to notice in the generated images the characteristic orange color of Goldfishes, followed by eye-like shapes and patterns similar to a fish's scales.

Class 33 - Turtle



(a) *4 Layer Multiscaling, Blurring, learning_rate* = 0.1

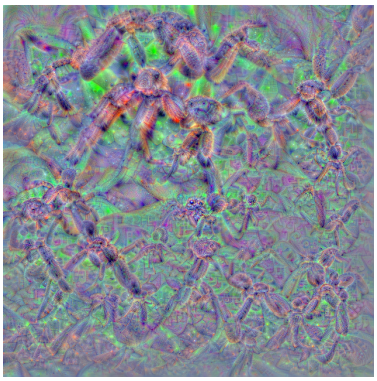


(b) *6 Layer Multiscaling, Blurring, learning_rate* = 0.1

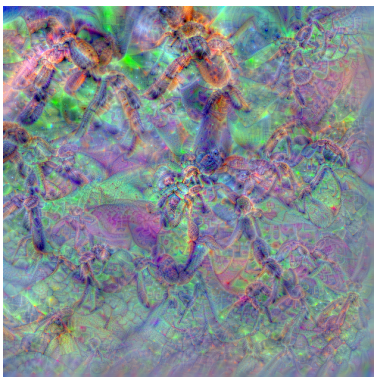
Figure 4.14: *Turtle Class Feature Visualization for VGG16*

The generated images feature the distinctive skin patterns of a turtle, but no recognizable turtle face is visible.

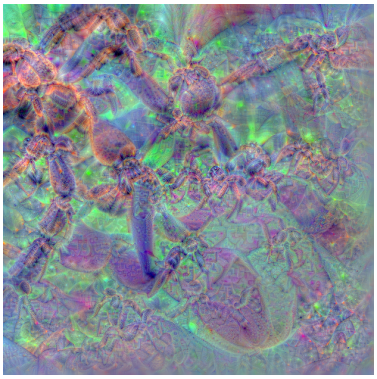
Class 77 - Spider



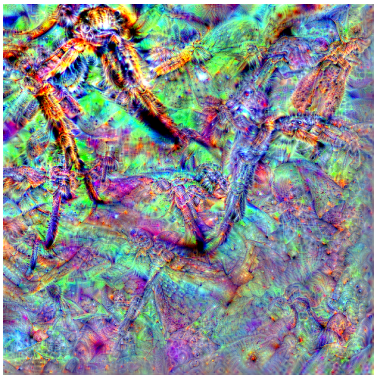
(a) *4 Layer Multiscaling, Blurring, learning_rate* = 0.1



(b) *6 Layer Multiscaling, Blurring, learning_rate* = 0.1



(c) *6 Layer Multiscaling, Blurring, learning_rate* = 0.09

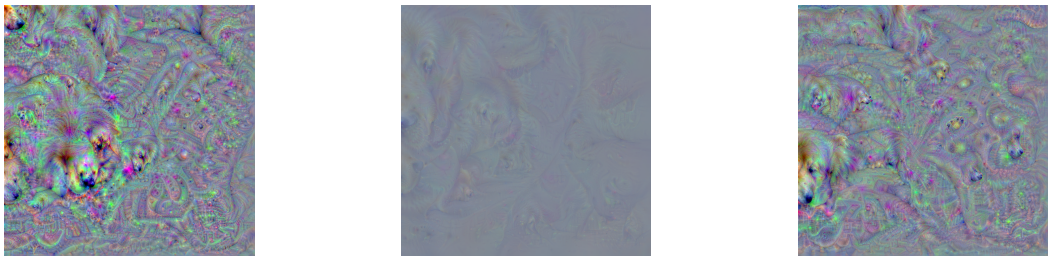


(d) *6 Layer Multiscaling, No Blurring, learning_rate* = 0.09

Figure 4.15: *Spider Class Feature Visualization for VGG16*

A spider's complete anatomy is visible throughout the generated images, with the arachnid's hairy limbs clearly visible at different angles.

Class 207 - Golder Retriever



(a) *4 Layer Multiscaling, Blurring, learning_rate* = 0.1

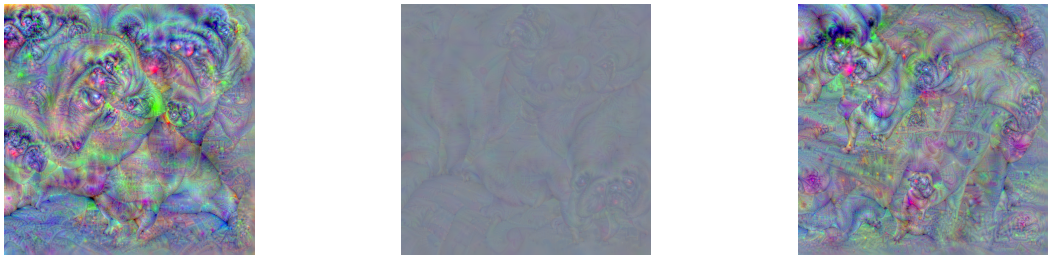
(b) *6 Layer Multiscaling, Blurring, learning_rate* = 0.01

(c) *4 Layer Multiscaling, Blurring, learning_rate* = 0.07

Figure 4.16: *Golden Retriever Class Feature Visualization for VGG16*

The generated images predominantly feature fur patterns and facial details of a golden retriever, with image (b) displaying a structure resembling a complete golden retriever face on the left.

Class 254 - Pug



(a) *6 Layer Multiscaling, Blurring, learning_rate* = 0.1

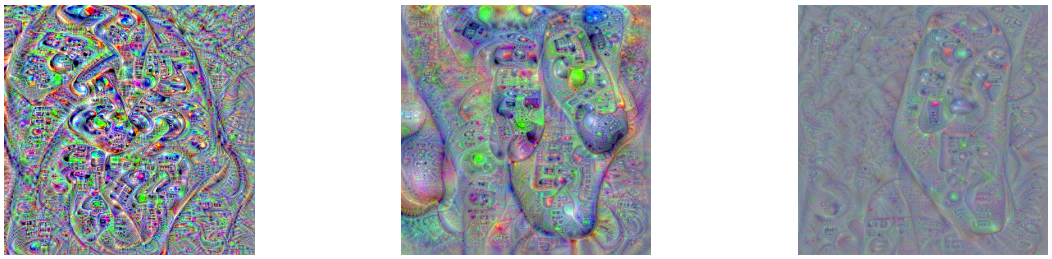
(b) *4 Layer Multiscaling, Blurring, learning_rate* = 0.01

(c) *6 Layer Multiscaling, Blurring, learning_rate* = 0.08

Figure 4.17: *Pug Class Feature Visualization for VGG16*

The generated images exhibit certain facial features of a Pug, including patterns that resemble its distinctive eyes and skin.

Class 761 - Remote Control



(a) *4 Layer Multiscaling, No Blurring, learning_rate* = 0.1

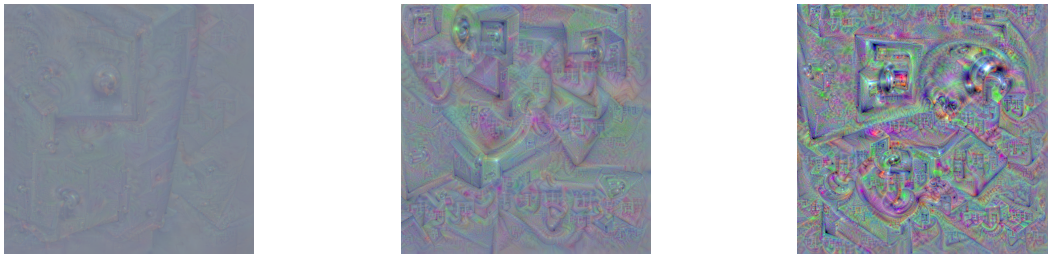
(b) *6 Layer Multiscaling, Blurring, learning_rate* = 0.1

(c) *4 Layer Multiscaling, Blurring, learning_rate* = 0.04

Figure 4.18: *Remote Controller Class Feature Visualization for VGG16*

The generated images do not strongly align with a human's typical perception of a remote controller. However, the convex shape with round patterns visible in image (c) might bear some resemblance to a remote controller.

Class 771 - Safe



(a) *6 Layer Multiscaling, Blurring, learning_rate* = 0.01

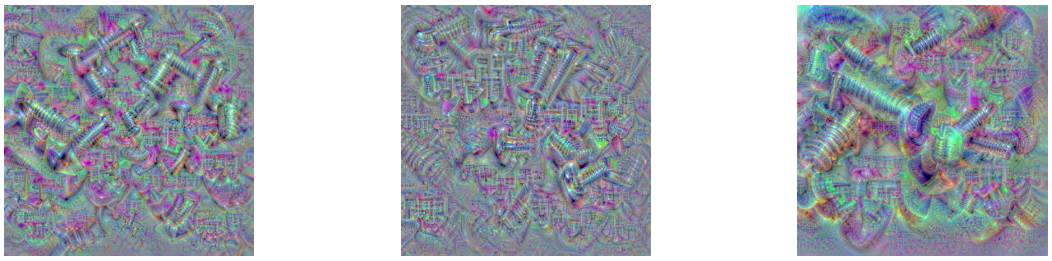
(b) *6 Layer Multiscaling, Blurring, learning_rate* = 0.04

(c) *4 Layer Multiscaling, Blurring, learning_rate* = 0.09

Figure 4.19: *Safe Class Feature Visualization for VGG16*

Image (a) appears to have structures very similar to a safe, with a shape close to a square with a circle inside. The generated images feature multiple straight lines and circular patterns, which are also found in the structures of safes.

Class 783 - Screw



(a) *4 Layer Multiscaling, Blurring, learning_rate* = 0.1

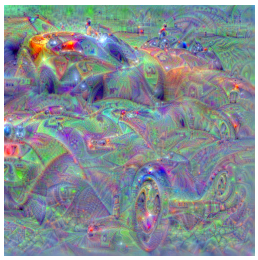
(b) *4 Layer Multiscaling, No Blurring, learning_rate* = 0.04

(c) *6 Layer Multiscaling, Blurring, learning_rate* = 0.09

Figure 4.20: *Screw Class Feature Visualization for VGG16*

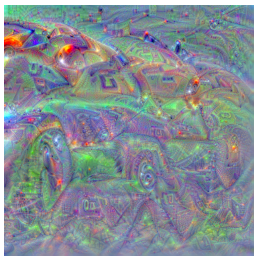
Patterns very closely related to screws are present in the generated images, with screws in multiple positions and angles present in the generated content.

Class 817 - Sports Car



(a) *6 Layer Multiscaling, Blurring, learning_rate* = 0.1

(b) *6 Layer Multiscaling, Blurring, learning_rate* = 0.08

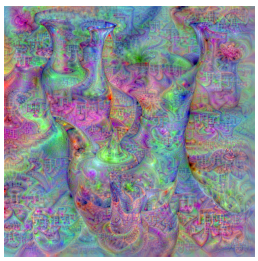


(c) *6 Layer Multiscaling, Blurring, learning_rate* = 0.09

Figure 4.21: *Sports Car Class Feature Visualization for VGG16*

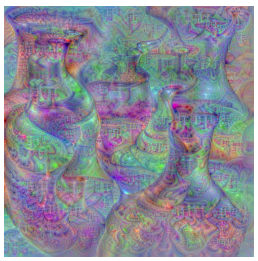
Circles similar to wheels are present in the generated images, but not much resemblance is noticeable in the examples.

Class 883 - Vase



(a) *6 Layer Multiscaling, Blurring, learning_rate* = 0.1

(b) *6 Layer Multiscaling, Blurring, learning_rate* = 0.04



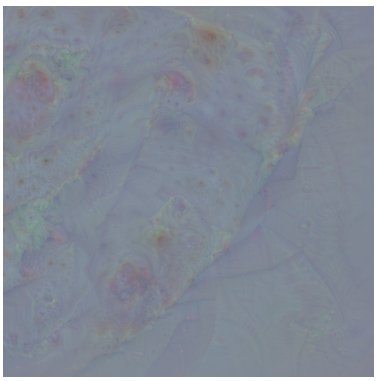
(c) *6 Layer Multiscaling, Blurring, learning_rate* = 0.08

Figure 4.22: *Vase Class Feature Visualization for VGG16*

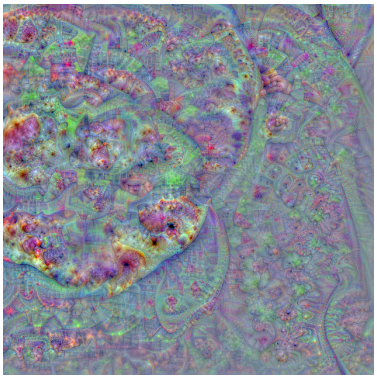
The sinuous shapes present in the images are very similar to a vase's structure. It is possible to notice that the characteristic curve is present in multiple parts of the images.

Class 963 - Pizza

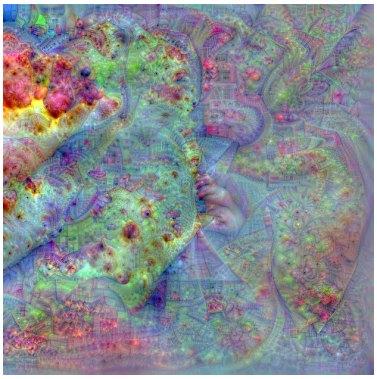
(a) *4 Layer Multiscaling, Blurring, learning_rate* = 0.1



(b) *6 Layer Multiscaling, Blurring, learning_rate* = 0.01



(c) *4 Layer Multiscaling, Blurring, learning_rate* = 0.09



(d) *6 Layer Multiscaling, Blurring, learning_rate* = 0.09

Figure 4.23: *Pizza Class Feature Visualization for VGG16*

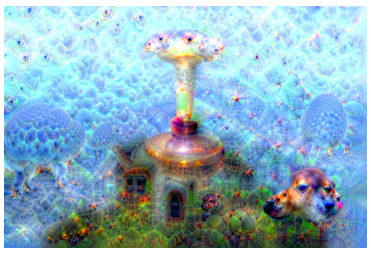
The images feature patterns reminiscent of melted cheese on a pizza, along with circular structures that evoke the classic pizza shape.

4.3.3 Feature Visualization in Non-Random Initial Images

Feature visualization techniques can also be applied to images with non-random initial states. By leveraging the methods discussed in this chapter, it becomes possible to create visuals that appear as if they were pulled directly from a *dream*.[1] The following examples showcase images generated using this technique, illustrating its ability to produce dreamlike and surreal visuals.

[1] The idea of applying Feature Visualization to Non-Random Initial images became popular by a program called *DeepDream*, created by Google

(a) *Original Image*



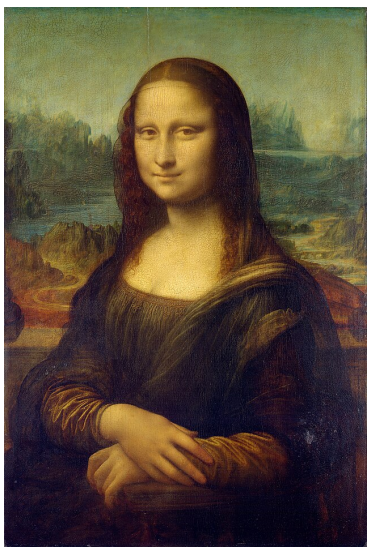
(b) *Feature Visualization of Image with Layer 11*

Figure 4.24: *Christ the Redeemer*

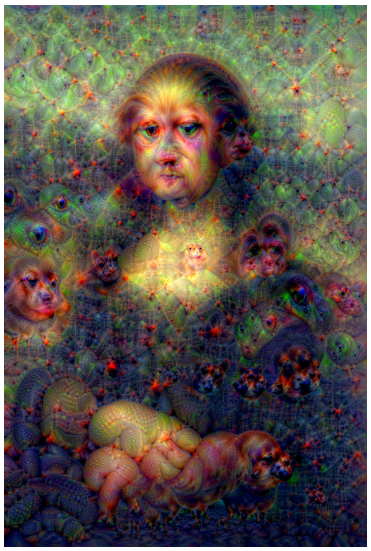
(a) *Original Image*

(b) *Feature Visualization of Image with Layer 11*

Figure 4.25: *IME-USP*



(a) *Original Image*



(b) *Feature Visualization of Image with Layer 11*

Figure 4.26: *Mona Lisa*

(a) *Original Image* **(b)** *Feature Visualization of Image with Layer 11*

Figure 4.27: *The Great Wave off Kanagawa*

Chapter 5

Final Considerations

In this capstone project, various techniques for enhancing the interpretability of Convolutional Neural Networks (CNNs) were explored, including LIME, Grad-CAM, and Feature Visualization.

These methods have proven to be valuable tools for understanding different aspects of a model's decision-making process. LIME and Grad-CAM facilitate the visualization of how a model justifies its predictions by highlighting the most influential regions of an input. Meanwhile, Feature Visualization provides deeper insights into how individual neurons and layers respond to distinct patterns in the input data, revealing the hierarchical representations learned by the model.

Our experiments indicate that Grad-CAM is more effective than LIME for visualizing model explanations. Grad-CAM produces a continuous heatmap over the pixels of an image, whereas LIME generates only binary masks over the original image. Additionally, LIME requires extensive hyperparameter tuning, while Grad-CAM operates without hyperparameter adjustments, making it a more practical choice in many scenarios.

The experiments further demonstrated that Feature Visualization provides meaningful insights into how different layers and class neurons process information, revealing the inner workings of the CNN.

In conclusion, this study contributes to a better understanding of XAI methods in CNNs, offering practical insights into the strengths and limitations of LIME, Grad-CAM, and Feature Visualization in model interpretability.

References

[He *et al.* 2015] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. "Deep residual learning for image recognition". *CoRR* abs/1512.03385 (2015). arXiv: 1512.03385. URL: http://arxiv.org/abs/1512.03385 (cit. on p. 22).

[Ribeiro *et al.* 2016] Marco Tulio Ribeiro, Sameer Singh, and Carlos Guestrin. *"Why Should I Trust You?": Explaining the Predictions of Any Classifier*. 2016. arXiv: 1602.04938 [cs.LG]. URL: https://arxiv.org/abs/1602.04938 (cit. on p. 11).

[Selvaraju *et al.* 2019] Ramprasaath R. Selvaraju *et al.* "Grad-cam: visual explanations from deep networks via gradient-based localization". *International Journal of Computer Vision* 128.2 (Oct. 2019), pp. 336–359. ISSN: 1573-1405. DOI: 10.1007/s11263-019-01228-7. URL: http://dx.doi.org/10.1007/s11263-019-01228-7 (cit. on p. 2).