UNIVERSITÀ DI PERUGIA

DIPARTIMENTO DI MATEMATICA E INFORMATICA

BACHELOR'S THESIS IN COMPUTER SCIENCE

Web Application For Adversarial Attacks Through Image Filters

Advisors

Prof. Valentina Poggioni

Ph.D. student Alina Elena Baia

Ph.D. student Gabriele di Bari

Candidate

Luca Ceccagnoli

Academic Year 2018-2019

Contents

Introduction 4

1 Background **5**

1.1 Machine Learning 5

1.1.1 The learning process 5

1.1.2 Classifiers 7

1.1.3 Optimization 7

1.1.4 Generalization 9

1.2 Artificial Neural Networks 11

1.2.1 Forward propagation 12

1.2.2 Backpropagation 14

1.2.3 Batch Normalization 16

1.3 Convolutional Neural Networks 18

1.3.1 Convolutional layers 19

1.3.2 Pooling Layers 22

1.4 Neural Network Models 24

1.4.1 VGG 24

1.4.2 Inception 25

1.4.3 ResNet 29

1.4.4 MobileNet 31

1.4.5 DenseNet 34

2 Adversarial Machine Learning **38**

2.1 Adversarial threat model 39

2.2 Generating adversarial examples 41

2.2.1 Fast Gradient Descent Method(FGSM) 41

2.2.2 Basic Iterative Method(BIM) 42

2.2.3 DeepFool(DF) 42

2.2.4 Carlini & Wagner(CW) 42

2.3 Defenses 43

2.4 Universal Unrestricted attacks with AGV 44
2.4.1 Three-step process 44

3 AGV Web Application **46**
3.1 Tools 46
3.2 The ImageNet dataset 47
3.3 Image Filters 48
3.4 Graphical User Interface 52
3.4.1 Prediction using preset images 53
3.4.2 Prediction using custom images 54
3.4.3 Attacking with preset models 54
3.4.4 Creating custom attacks 55
Conclusions 57
References 58

Introduction

Deep learning is a branch of machine learning based on deep neural networks (DNNs). It has become a popular field of study in the last twenty years, with applications in many real-world scenarios. Deep neural networks architectures have been employed with great results in tasks like computer vision, speech recognition, natural language processing, cancer detection[60], crop yield forecasting[5] or self-driving cars[16]. Notably, for *image classification* tasks, **convolutional neural networks(CNNs)** spiked in popularity after the great results achieved by *AlexNet* on the *ImageNet* dataset[33]. Networks developed in the later years even managed to surpass human accuracy[43]. Despite this, recent studies regarding the security aspects of neural networks have shown that they can be easily fooled by specifically crafted inputs, resulting in mistakes that would not be normally made by humans. As neural networks become an increasingly present component of our lives, it is easy to see how exploiting these weaknesses could lead to dangerous effects. Because of this, the field of **adversarial machine learning** has been actively researching algorithms to generate malicious inputs, and ways to counter them. In regards to computer vision, most attack algorithms create adversarial images by using noise or small, single pixel perturbations. These techniques produce unnatural images, that risk being detected by humans, if the perturbations are too intense. Inspired by widespread image filters from applications like *Instagram*, we investigate ways to generate continuous perturbations that maintain an image's semantic features, making them less susceptible to human awareness. We present a web application that acts as an interface to **the *AGV* algorithm**[7], allowing an user to apply sequences of *filters* on a given image, and use it to attack popular convolutional neural networks used in computer vision. The user will also be able to create custom attack models through an appropriate editor. By having access to a visualization of the filtered image and the output of the network, we will be able to compare its decisions with those of a human observer.

1 Background

This chapter presents some theoretical background to understand the subject of this thesis. Starting from *machine learning* concepts, then we will move our attention to *artificial neural networks*, more specifically to *convolutional neural networks*, that are the most prominent in the field of computer vision.

1.1 Machine Learning

"Machine learning is a field of study where computers are instructed to learn without being explicitly programmed to do so" -Arthur Samuel(1959)

Machine learning is a subfield of Artificial intelligence, the study of computer programs that automatically improve with experience[39]. In traditional programming, a computer is given input data and a program to calculate an output. In machine learning, the machine develops its own algorithm given input data and an error measure to evaluate its performance. The generated algorithm is referred to as a **Model**.
If we describe the desired solution to our problem as a **target function** $\hat{\boldsymbol{g}}$, the objective of our model is to generate a function $\boldsymbol{g}$ that *approximates* $\hat{\boldsymbol{g}}$, and it does so through an iterative training process, during which it tries to *minimize* its error measure. The closeness of this approximation determines the model's ability to *generalize*.

1.1.1 The learning process

In order to write a good algorithm, a machine learning model requires a large amount of information. This collection of input data is called **data set**: a collection of structured data objects, called **examples**, described by a number of *attributes*(also called *features*), which capture basic characteristics of an object.
Based on the form of the input data, we can define three main learning methods:

- **Supervised learning**: refers to any situation in which both the inputs and outputs of a component can be perceived[51]. In this case, the data set will be composed of labeled examples, that include the attribute values that our model *should* predict. Examples of supervised learning methods are *classification* and *regression*.

- **Unsupervised learning**: In unsupervised learning we do not have any information about the expected output, so the training set is unlabeled and the model is tasked with finding previously undetected patterns in the data. Examples of supervised learning methods are *clustering* and *association*.

- **Reinforcement learning**: In reinforcements learning, the model is not trained using a static dataset, but rather through dynamic interactions with an environment, by attributing a score or reward to certain actions. Through a trial and error process, the model will learn a function that chooses actions that maximizes the reward, given different options.

In this thesis we will be only looking at models that are generated following the supervised learning approach: given a set of n training examples in the form $\{(x_i, \hat{y_i}), i \in [1, \ldots, n]\}$, such that x_i is the feature vector of the i^{th} example and $\hat{y_i}$ is its target (or label), the algorithm seeks a function $g : X \longrightarrow Y$ such that $g(x_i)$ approximates $\hat{y_i}$ in some space of possible functions G, called the *hypothesis space*. A machine learning algorithm is generally divided in two phases:

- **Training phase**: the model can read the expected labels and *learn* by confronting them with its own predictions. We can say that training is performed by **induction**. The result of this phase is the model itself.

- **Test phase**: where the model is unable to read the data labels, and simply tries to label the examples using the *experience* it developed in the previous phase, which is also called a process of **deduction**.

Testing a model on the same data it was trained on would be trivial, thus the dataset is split into **training data** and **validation data**, each to be used in its respective phase. In machine learning, we refer to **parameters** as all the values which are automatically adjusted by the model during training, while **hyperparameters** are arbitrarily set by humans, and are a mean of controlling the learning process.

1.1.2 Classifiers

Classification is a type of supervised learning that consists in the task of assigning a class to unlabeled examples. By class we refer to a categorical variable, which can assume one from a finite set of distinct values. The mapping function, called **classifier**, will be defined as $g : X \longrightarrow Y$ where Y is a set of k categorical labels. The model will map each attribute set x to a predefined class in Y, or to a probability distribution describing how likely x is to belong to each class, like in Artificial Neural Networks, which are the topic of section 1.2.

1.1.3 Optimization

As we said before, machine learning models try to minimize an error measure to find the best **mapping function** $g(x)$. The employed error measure is conversely referred to as a **loss function**, defined as $L : (\hat{Y}, Y) \longrightarrow R$, that associates a *cost* to the discrepancy between the target label and the predicted one. The minimization process happens during training, and qualifies the model's task as an **optimization** problem.
The loss function will be described by a curve, where we can find a local minimum by using a technique called **gradient descent**.
In a multi-variable function, the gradient $\nabla L(\theta)$ is a vector that contains the curve's partial derivative $\nabla L(\theta)_\theta$ for each parameter θ, where by *parameter* we refer to any value that is learned by the model during training. The gradient tells us in which direction the function increases faster, or, in mathematical terms,

its *directional derivative*. By considering the *negative* gradient $-\nabla L(\theta)$, we can instead move towards a **local minimum**. The vector's direction will tell us *which* parameters have to change in order to reach the minimum, while its magnitude will tell us by *how much*.

Gradient descent is an iterative algorithm, where at each step (Figure 1.1), every parameter is updated using the following equation:

$$\theta = \theta - \lambda \nabla(\theta)_\theta L(\theta) \tag{1.1}$$

Where λ is the learning rate, an *hyperparameters* with a value between 0 and 1, which controls how strongly changes to the gradient at each step will impact the parameters. Hyperparameters are arbitrarily set before training, and are not changed by the model. The algorithm stops when changes to the gradient become negligible, meaning that it has reached a local minimum in the function, or after a set number of iterations.

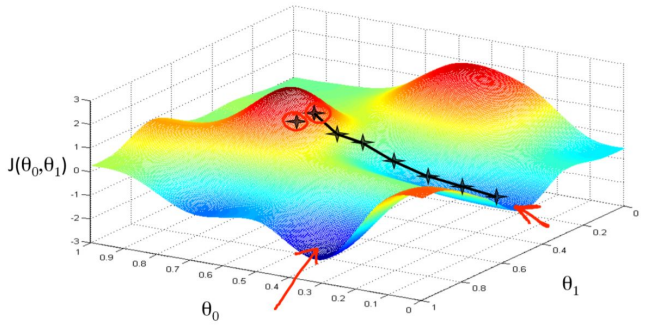
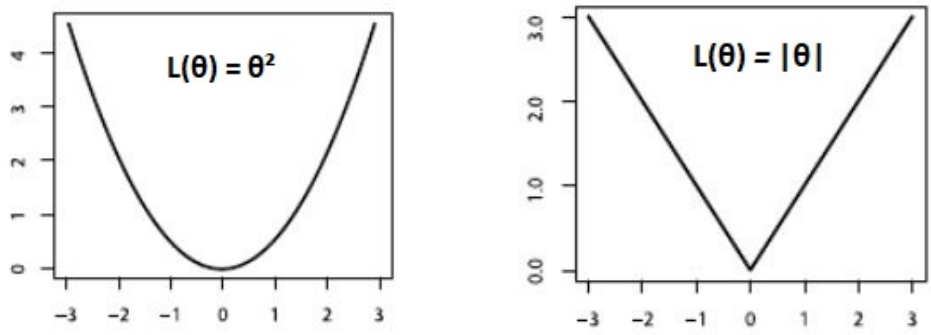


Figure 1.1: Gradient descent over a two-parameter loss function. θ is initialized with random values and approaches a local minimum with each step[44].

As gradient descent works with derivatives, the loss fuction we pick should be globally *continuous* and *differentiable*. Some examples are the **squared loss** $L(\theta) = \theta^2$ and **absolute loss** $L(\theta) = |\theta|$(Figure 1.2).



(a) **(b)**

Figure 1.2: The squared loss(a) and absolute loss(b) functions[41].

1.1.4 Generalization

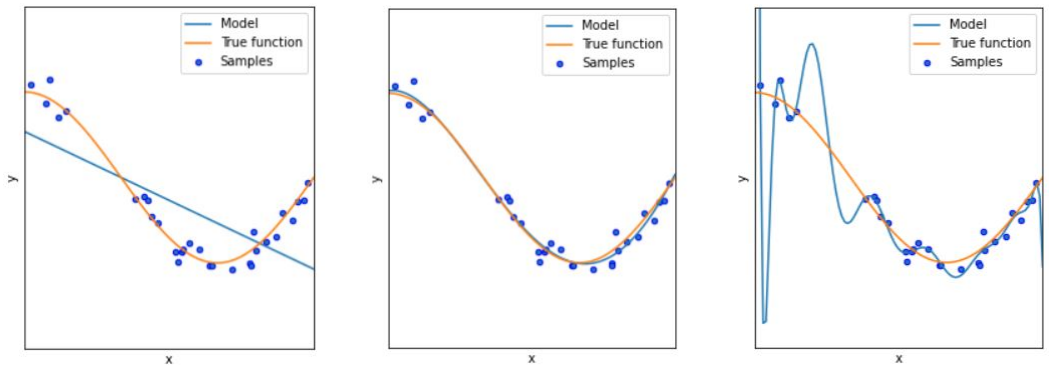
The final result of our optimization process should be a model capable of correctly predicting new, unseen data, other than the examples it has been trained on. This determines its ability to **generalize**.

We define **error rate** as the amount of correctly labeled examples compared to the whole data. In the training phase, this gives us the **training error**, which estimates the effectiveness of our training process. Conversely, from the test phase we gather the **validation error**, also called **generalization error**, as it evaluates the model's generalization performance. When both these values are low, the model is said to be a *good fit*, and the curve of $g(x)$ is very close to that of $\hat{g}(x)$.

If, during the training of our model, both the training and test error are too high, it means that our model is **underfitted**(Figure 1.3a), either too simple or too inexperienced to understand the data. Underfitting is easy to detect, and can sometimes be fixed by simply training the model longer, or increasing its complexity (for example, by adding more layers to a neural network), other times we might need to use another model altogether.

On the other hand, **overfitting**(Figure 1.3c) is a much more subtle issue: by training *too much*, the model becomes very proficient at recognizing its training data, but unable to generalize over new data. The training error keeps improving,

while the test error starts to plateau and even increase. Chance of overfitting increases as a model becomes more complex, so when trying to choose a correct model for our task, we follow the principle of parsimony[1] according to which, given two models with the same generalization error, the simpler one is preferred [42]. Overfitting is a wide topic, and it goes beyond the scope of this thesis to discuss its solutions.



(a) Underfitting **(b)** Good fit **(c)** Overfitting

Figure 1.3: Graphs showing labeled data(blue dots), their associated target function $\hat{g}(x)$(orange) and the model $g(x)$. An underfitted model(a) does not understand the curve of the data, while an overfitted one(c) becomes too specific and will not generalize well. A model that is a good fit(b) will presente a curve that is very close to the target one[57].

[1] Also called Occam's razor, formulated by Wilhelm Occam in the 14^{th} century

1.2 Artificial Neural Networks

Artificial Neural Networks(ANNs) are a type of machine learning model inspired by biological brains. They are composed of computational units called **neurons**, organized in layers, that process and propagate data to other neurons in different layers. There are three types of layers:

- The **input layer** is the first layer of every network, initialized with one or more examples.
- The **output layer** is the last one, and returns the final computation of the entire network.
- Every layer between the input and output layers is called a **hidden layer**. They gather inputs from the previous layer and send their outputs to the next one.

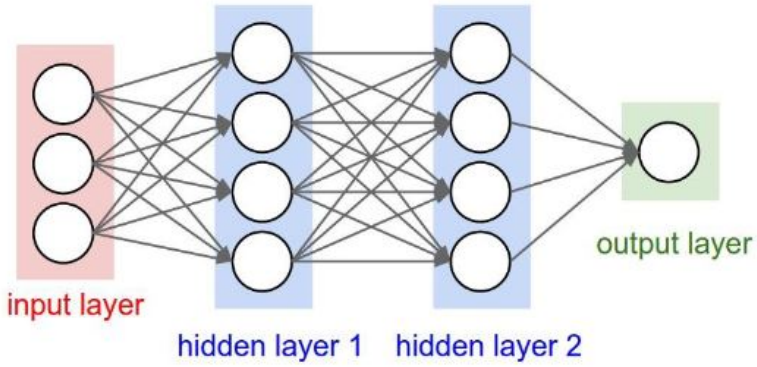


Figure 1.4: An artificial neural network[18].

Every neural network must contain one input layer, one output layer and 0 or more hidden layers, each with an arbitrary number of neurons.
Neural networks can be portrayed a *weighted graphs*, where neurons constitute the *nodes*, while their connections are the *edges*. Every edge has an associated **weight**, which quantifies of the importance of that connection, and can be seen as the

strength of a synaptic link in a biological brain(Figure 1.5). Weights constitute the network's *parameters*, that are usually initialized randomly and adjusted during the training phase.
We can label network architectures based on how their layers are connected:

- **Feed-forward** networks, where connections between neurons form a directed acyclic graph, opposed to **recurrent** networks, where connections form a directed graph.
- **Fully connected** networks, where every neuron in a layer is connected to every neuron in its successor, opposed to **convolutional** networks, were every neuron is connected to a small part of its previous layer. Convolutional networks are discussed in detail in section 1.3.

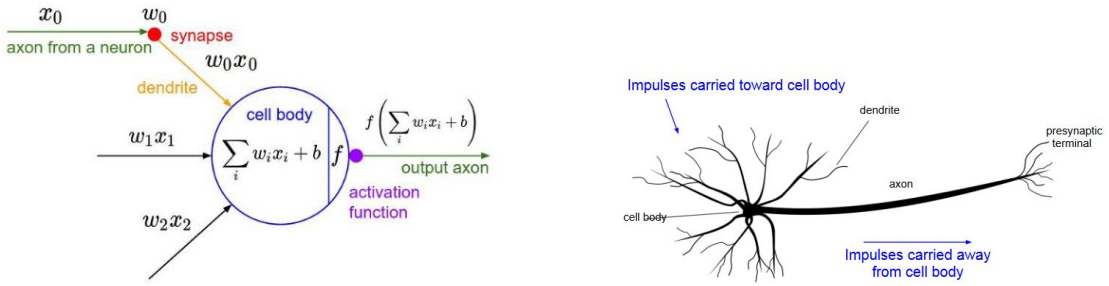
All architectures consisting of at least one deep layer are categorized as **deep neural networks(DNNs)**, or deep learning models. The number of layers in a DNN is referred to as its **depth**.

1.2.1 Forward propagation

Neurons receive multiple inputs but produce a single output. A neuron calculates the weighted sum of its inputs and processes them with a function, called **activation function**. If the final value exceeds a certain *threshold*, the neuron is said to *activate*, propagating its output to other neurons, who will repeat the process up to the final layer.
given a neuron N in layer (d), a vector X containing the outputs of its connected neurons form layer $(d-1)$ and a vector W containing their weights and a function σ, its output $\mathbf{y}$ will be computed as:

$$\mathbf{y}(X, W) = \sigma(X \cdot W + b) \tag{1.2}$$

where b is a **bias**, a parameter that sets the activation threshold, and σ is the



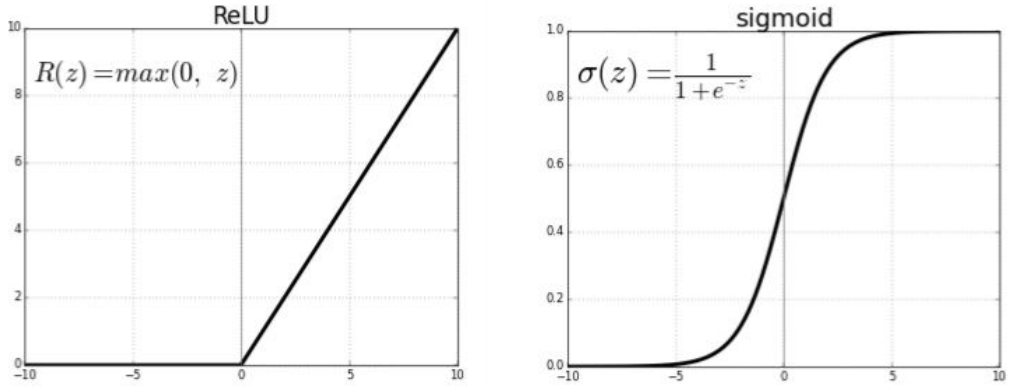
(a) **(b)**

Figure 1.5: parallel between a neuron in an ANN(a) and in a biological brain(b). We can consider dendrites as the inputs from previous neurons, and the axon as the single output, propagated to other neurons through the synapses[18].

activation function. If the output **y** is greater than the bias threshold b, it will be propagated to the connected neurons at layer $(d + 1)$. Neural networks can use many types of activation functions [4]. Linear functions are easier to optimize but are only able to solve problems with linearly-separable solution spaces[38].
With a non-linear activation function,a two-layer neural network has been proven to be an universal function approximator[13]: by stacking at least two non-linear layers, we can approximate any possible continuous function. For the purpose of this thesis, the actifation function we are mostly interested in is the **rectifier**(Figure 1.6a): a piecewise function defined as the positive part of its argument.

$$\sigma(x) = \mathbf{max}(0, x) \tag{1.3}$$

It returns the provided input if it is positive, otherwise 0. The rectifier presents the benefits of a linear function without actually being one. It is simple to calculate and can output true 0 values(with negative inputs), which makes a network more discriminating[10]. Layers that employ a rectifier are called **Rectified Linear Units(ReLU)**, and they have been crucial to the development of very deep neural networks in the last decade[21].
We will also look at the **softmax** function, which is a generalization of the *logistic function*(Figure 1.6b): it takes an input vector of real numbers and normalizes it



(a) **(b)**

Figure 1.6: The ReLU(a) and sigmoid(b) activation function. Other than being the base function for softmax, the sigmoid was one of the more popular activation functions before ReLU. [49]

into a probability distribution proportional to the exponents of the input numbers:

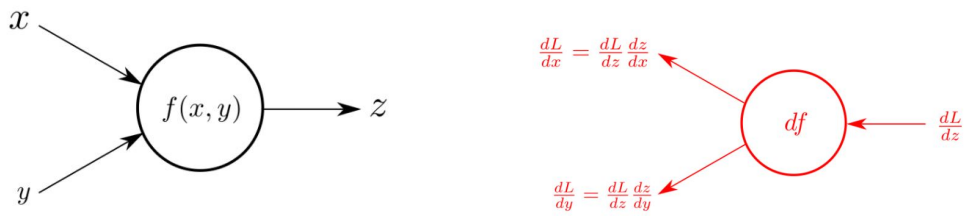
$$\sigma(\vec{x}) = \frac{e^{x_i}}{\sum_{j=1}^{K} e^{x_J}} \tag{1.4}$$

The elements of the input vector will always be normalized to a value between 0 and 1. It is typically used in the output layer: neural networks return unnormalized real numbers as class scores, so a softmax layer is added to normalize these values into a range that makes them interpretable as probabilities.

1.2.2 Backpropagation

Backpropagation is a supervised learning algorithm used to train the *weights* of feed-forward neural networks. Starting from the output layer, weights are recursively updated backwards using gradient descent. It is inspired by *Hebb's Rule*[35] and dynamic programming.

Like in other machine learning models, the objective of the training is to minimize a loss function, by calculating its gradient in the weight space. This is simple in the output layer, where we have direct access to the outputs, but in the hidden layers



(a) **(b)**

Figure 1.7: forward-propagation(a) and back-propagation(b), which shows the use of the *chain rule* to calculate gradients[24].

those values depend on the output of the previous neurons. The entire network can be seen as a composed function, whose gradient can be computed using the *chain rule*(Figure 1.7). This way, the partial derivative of the loss function, with respect to each weight, can be recursively calculated from the previous layer, and used to adjust the parameters. the gradient with respect to a single weight can be calculated as:

$$\frac{\partial L}{\partial w_{jk}^{(d)}} = \frac{\partial L}{\partial z_j^{(d)}} \frac{\partial z_j^{(d)}}{\partial w_{jk}^{(d)}} \tag{1.5}$$

After obtaining the previous gradient, weights can be updated using gradient descent:

$$w_{jk}^{(d)} = w_{jk}^{(d)} - \lambda \frac{\partial L}{\partial w_{jk}^{(d)}} \tag{1.6}$$

Where λ is the learning rate, $w_{jk}^{(d)}$ refers to the weight of the connection from neuron j at layer $d-1$ to neuron k at layer d, and $z_j^{(d)} = w_{jk}^{(d)} y_j^{(d)} + b_j^{(d)}$.

We define one **epoch** as one forward pass and one backward pass of all the training data. Training a neural network usually involves multiple epochs.

In **stochastic gradient descent**, backpropagation is executed after once for every example in an epoch, which is a fast and lightweight method, but computationally intensive, and is not guaranteed to reach a local minimum.

batch gradient descent runs backpropagation once per epoch, after processing all the training data and using an average of all the gradients. It is more stable

and converges more reliably, but is too memory intensive for large datasets.
In practice, training data is usually grouped into multiple batches, and backpropagation is run after processing an entire batch of examples. This is called **mini-batch gradient descent**, which gives us a good middle ground between the two other methods (Figure 1.8). Given a dataset with X examples and a **batch size** of B, backpropagation will run $\frac{X}{B}$ times per epoch. Batch size and number of epochs are both *hyperparameters*.

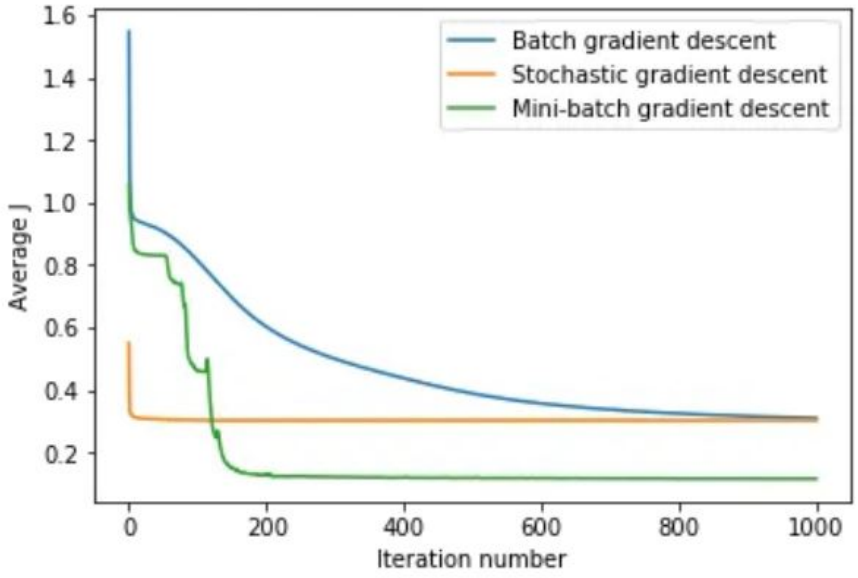


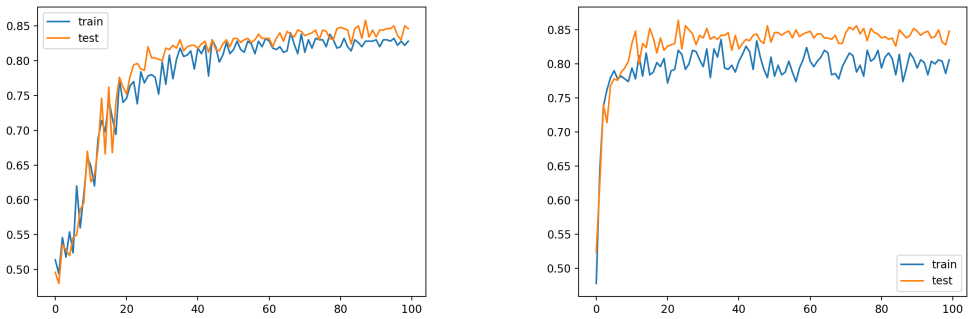
Figure 1.8: The three gradient descent methods. While mini batches do not converge the fastest, they find the better local minimum[6].

1.2.3 Batch Normalization

In backpropagation, gradient descent works under the assumptions that the data distribution does not change in the other layers. In practice, all the layers are updated simultaneously[22], but the distribution likely changes after updating the weights of the previous layer. This variation is called *internal covariate shift*, and can greatly increase the training time. As the inputs to each layer are affected by parameters in the previous layers, the phenomenon is amplified as the network

becomes deeper. **Batch Normalization** was proposed in [31] to help coordinate the update of multiple layers in a model, by sequentially *standardizing* the input of the nodes in each layer. To standardize the inputs means to change their distribution such that it has a *mean* of 0 and a *standard deviation* of 1. Their values can be calculated for each layer per mini-batch, or by maintaining an average of mean and standard deviation across mini-batches.

Batch normalization has been shown to improve **training time** significantly(Figure 1.9), but the actual reasons for this are still unclear. Instead of reducing the internal covariate shift, it has been argued to simply *smooth* the objective function[47], improving performance.



(a) Without batch normalization **(b)** With batch normalization

Figure 1.9: Plots showing effects of batch normalization on a *multilayer perceptron(MLP)*.The x axis shows the number of **epochs**, the y axis shows the **accuracy**. While not increasing overall, the peak accuracy is reached much faster by the model employing batch normalization(b)[11].

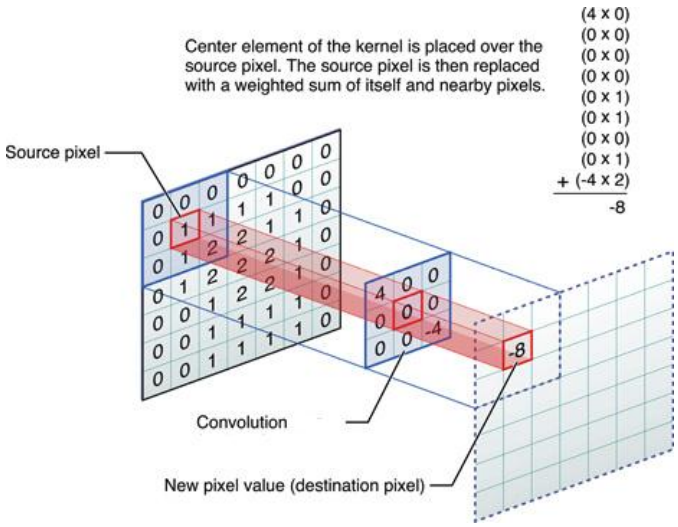


Figure 1.10: convolution example: each sum between the kernel and its projection over the input produces a single value. [9]

1.3 Convolutional Neural Networks

Convolutional Neural Networks(CNNs) are based on **convolutional operations** instead of standard matrix multiplication, and have proved very succesful in computer vision tasks over the last decade. Each neuron only receives input from a subset of the previous layer neurons, called *receptive field*, and data is represented as a volume instead of a vector, where the first two dimensions denote the *size* ($height \times width$), while the third one is the *depth*, which refers to the number of channels or feature maps.

CNNs were inspired by biological research on the animal visual cortex, started by the studies of David Hubel and Torsten Wiesel[29], who observed a hierarchical structure in visual neurons, where deeper cells are more complex and are able to recognize higher level patterns.

Stacking convolutional layers allows a model to learn low level features (like lines) in the early layers, and complex features (shapes or objects) in the deeper layers. First, we will look at the basic operation of convolution, then two layer types introduced by the CNN: **convolutional layers** and **pooling layers**.

In image processing, convolution is an iterative operation that involves transforming an image by applying a weight kernel over its pixels(Figure 1.10). The kernel is placed on every pixel in the image where it can fully fit, computing a weighted sum of the pixel and kernel values, which is then assigned to a single pixel in the new image.

1.3.1 Convolutional layers

Convolutional layers operate through the use of **filters**: a filter is a stack of same-sized convolution kernels, one for each input channel, thus having the same depth as the input volume. The size of these filters is determined by an hyperparameter called **size**, and the values stored in each kernel correspond to the weights of the model. Every convolutional layer usually applies multiple filters. Given an input volume X of size $(N_1 \times N_1 \times D)$ and a filter W of size $(F \times F \times D)$, the filter will slide over each $F \times F$ area in X, computing the following dot product:

$$W^T \cdot x + b \tag{1.7}$$

where W^T is the transposed filter, x is the current area and b is the bias. After convoluting, the neurons will apply a non-linear **activation function**[2] (usually ReLU) like in normal neural networks, and construct a new **activation map** of depth 1 (Figure 1.11a). The activation maps computed by all the filters are concatenated (Figure 1.11b), and compose the layer's output volume Y of size $(N_2 \times N_2 \times K)$, where K is the number of filters used.

[2]in some literatures, convolution and activation are considered as two separate layers. This makes no difference in practice, and we prefer to group the operations for simplicity.

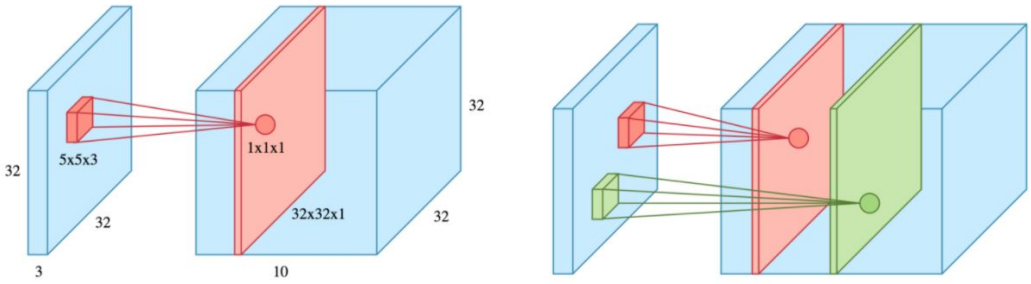


Figure 1.11: Output feature maps of a conv. layer. The red and green colors refer to different filters.(a) shows how a single channel is calculated, (b) how they are stacked to create an output volume. [15]

The output image of a convolution will always be smaller than its input: a 32x32 image convoluted by a 5x5 filter will become a 28x28 image, and subsequent convolutional layers will reduce its size even more, which quickly becomes a problem. This is called **border effect**, as it leads to information loss on the edges of the image (Figure 1.12).
To better explain this issue and how it is handled, we will introduce two more hyperparameters used in convolutional layers:

- **stride**: a positive integer that controls how the receptive fields are allocated over the volume. A stride of $S = n$ means that filters will be translated by n units after each convolution. The number of filter translations that can fit in an input volume is $\frac{(N-F)}{S} + 1$. In practice, a stride greater than 3 is very rare.

- **zero padding**: the operation of artificially adding 0 values to the edges of an output image, which is usually done to prevent the loss of spatial size naturally caused by the convolution. A zero padding of $P = n$ adds a border of n pixels around the image, e.g. an 8x8 image with $P = 1$ becomes a 10x10 image.

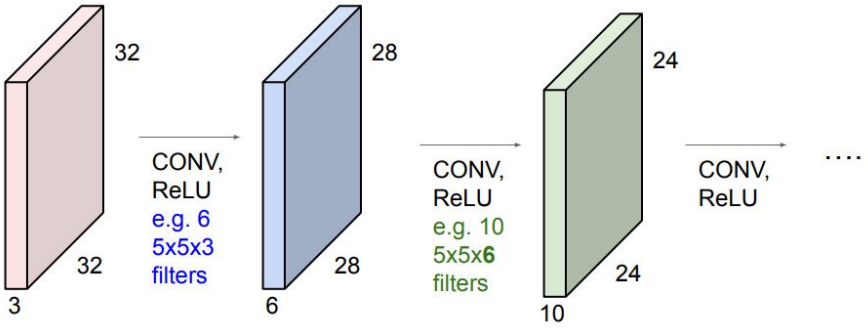


Figure 1.12: The border effect visualized. The depth of the volume remains determined by the number of filters. [18]

So stride directly determines the size of the output, and we can use zero padding to add the lost dimensions back. In practice, the output size can be computed as:

$$N_2 = \frac{N_1 - F + 2P}{S + 1} \tag{1.8}$$

where N_1 is the the input size, F is the filter size, P is the zero padding, S is the stride.

Convolution confers three interesting properties to a machine learning model [22]:

1. **Sparse interactions**: We can map every kernel translation to a neuron, where its receptive field corresponds to the connections it receives from the previous layer. Since a kernel is typically much smaller than the input, we can see that the connection space would result in a *sparse matrix*. Compared to a fully connected layer, this result in much fewer parameters and operations.

2. **Parameter sharing**: Each neuron multiplies its input with the weights stored in its kernel, so neurons at the same depth share the same weight vector, which results in further reduction of the parameter count. The number of parameters for each layer can be calculated as $((F^2 \cdot D) + b) \cdot K$.

3. **Equivariant mapping**: A function $f(x)$ is *equivariant* to a transformation T if changes in its input are mirrored by equal changes in the out-

put: $f(T(x)) = T(f(x))$. Because a single filter always convolutes using the same weights, convolutional layers are equivariant towards *translation*, which means that they will be able to recognize a feature independently from its position in the image(Figure 1.13). Note that this could be undesirable if our task requires features to occupy specific (relative) positions.

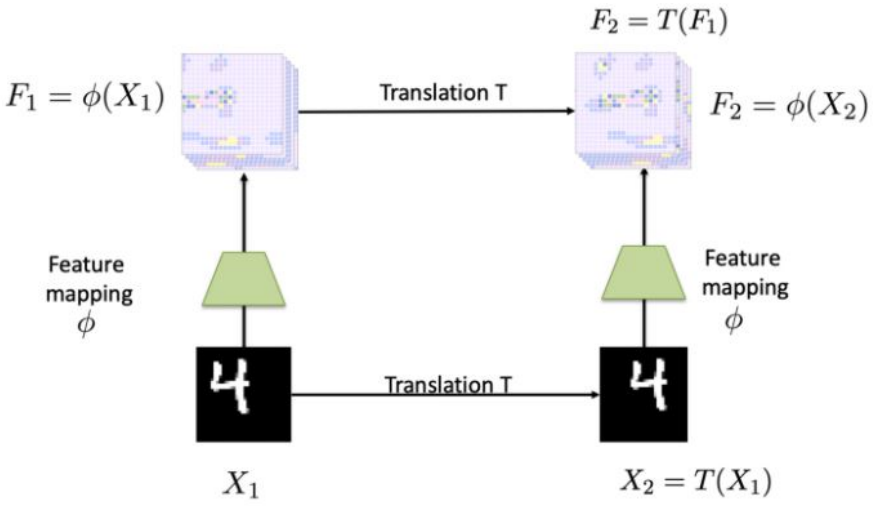


Figure 1.13: Equivariant mapping: translating the digit 4 does not change its representation in memory, even after the convolution ϕ is applied[58].

1.3.2 Pooling Layers

The purpose of pooling layers is to reduce the size of the input volume, while preserving its depth. Pooling consists in an operation of **downsampling** of each feature map: using a kernel to reduce a channel's resolution, by computing a summary value of its regions. The two most common types of pooling are **max pooling** (Figure 1.14), which returns the highest value overlapped by the kernel, and **average pooling**, which calculates an average of those values.

Pooling layers use **kernel size** and **stride** as their hyperparameters. Given a kernel of size K and stride S, the output map size will be $N_2 = \frac{N_1 - K}{S} + 1$. Pooling layers are usually placed after a series of convolutional layers. Pooling reduces the

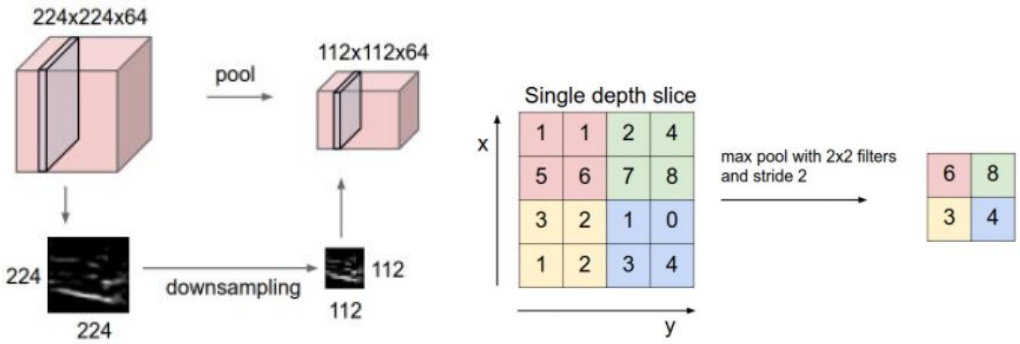


Figure 1.14: Max pooling, from a high level(a) and low level(b) point of view.

number of parameters and computational cost, making the network less prone to overfitting, but also grants some degree of **translation invariance**: a function $f(x)$ is invariant towards a transformation T when the transformation does not affect the output: $f(T(x)) = f(x)$. This means that for *slight* translations in the input, the pooled output will not change. Despite these benefits, pooling is based on the heuristic that the function learned by the layer is invariant, and will cause information loss if this assumption does not hold true [22]. Recent literature discusses better alternatives to pooling, like using higher stride convolutions [52], or *capsule networks* [45], so pooling layers might disappear from future architectures.

1.4 Neural Network Models

This section analyzes the neural networks used to test adversarial attacks in our application, describing their architectures and looking at their main selling points. These models were picked with the purpose of providing an overview of how different architectures evolved, addressing specific issues, rather than showing state-of-the-art results.

1.4.1 VGG

VGG was first presented as a submission for the ILSVRC2014 ImageNet challenge[50], where it achieved first and second place in the localisation and classification tasks, respectively. The idea behind VGG was to increase depth by relying on small, 3x3 convolution filters. VGG was first tested in 5 different configurations, designed from the same architecture, based on the following components:

- 8-16 non-linear convolution layers whose filters use a 3x3 receptive field with 1 pixel stride. The filters start at a depth of 64, which is doubled after every pooling layer, since the reduced resolution can allow for deeper filters without impacting performance.
- Spacial padding to preserve the resolution after convoluting.
- 5 spacial pooling layers, which perform 2x2 max pooling with stride 2.
- 3 fully connected layers, of which 2 with 4096 channels and one with 1000 (corresponding to the 1000 ImageNet classes).
- A final loss layer, using softmax.
- All hidden layers use The ReLU activation function.

The different configurations progressively increase in number of weight layers(from 11 to 19). 5x5 and 7x7 convolutions can be respectively split into 2 and 3 3x3

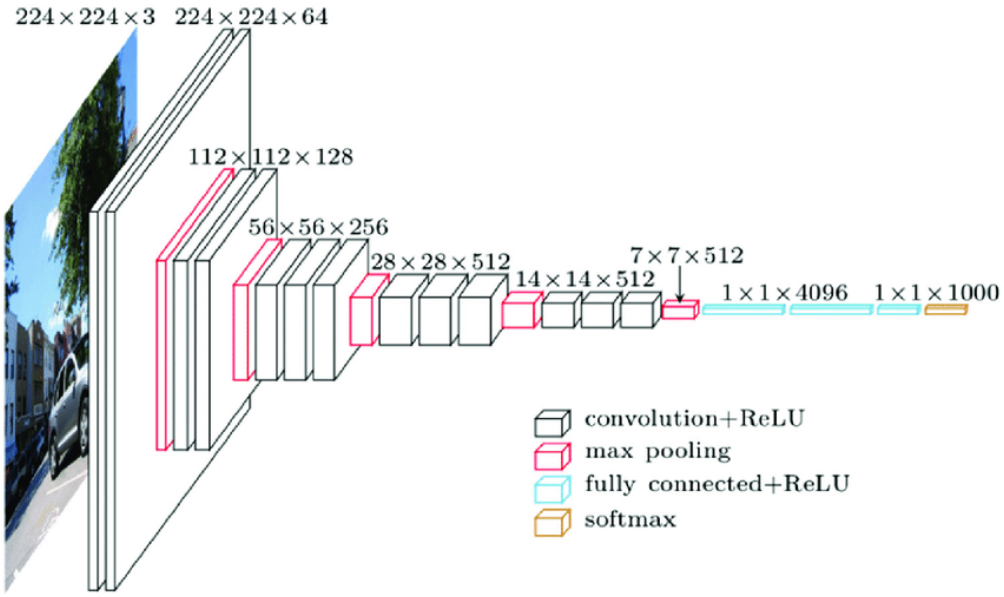


Figure 1.15: The VGG-19 architecture[60].

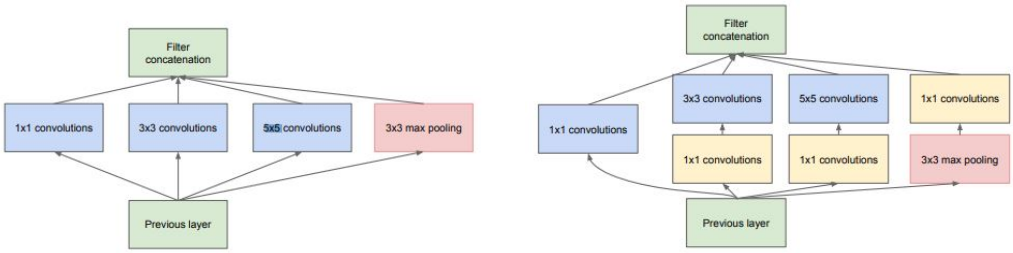
convolution layers, which makes the decision function more discriminative and reduces the number of parameters.

We will only look specifically at the last of the configurations, labeled with the letter "E", also called ***VGG-19***, which is the one used in our project (figure 1.15).VGG-19 achieved the best performance among the other configurations in regards to both top-1 error rate (25.5 %) and top-5 error rate(8.0 %) on the ILSVRC2012 dataset. It uses an input size of (224x224) and expects images with pixel values $\in [0, 255]$.

While VGG performed well against its competitors, it is a very slow model to train. Furthermore, its Keras implementation weights 549 MB [3], which makes it unsuited for some enviroments.

1.4.2 Inception

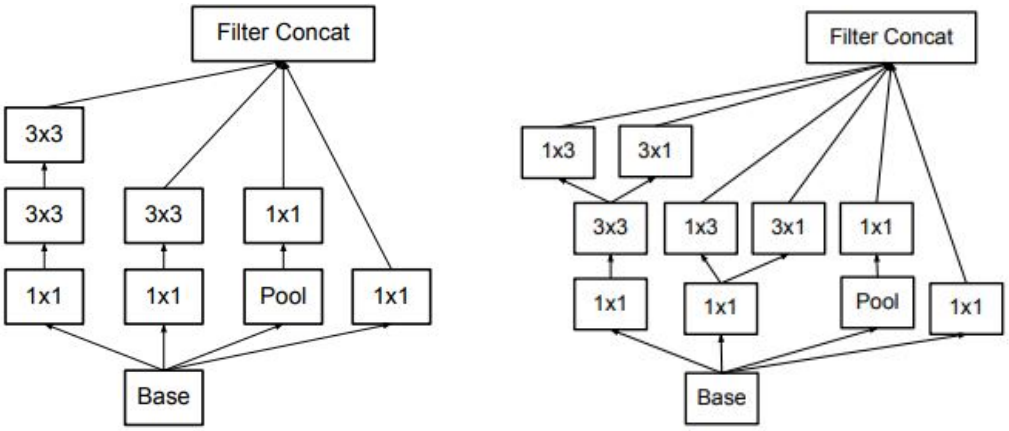
Inception's first iteration, called GoogLeNet(or InceptionV1), was another submission for the ILSVRC2014 challenge, winner of the classification task. In its paper[53] it is suggested that while increasing depth and width of a model is



(a) näive version **(b)** with dimension reductions

Figure 1.16: Layout of the **Inception Module**.

the safest way to improve its accuracy, it also increases the number of parameters, which lead to a higher risk of overfitting, and increased use of computational power. The long-term solution to this problem would be moving from *fully connected* to *sparsly connected* architectures, that would allow us to increase depth and size without spiking computational requirements. However, sparse matrix operations are handled very inefficiently by today's computing infrastructures. The core idea behind the Inception architecture is finding out how to approximate a local sparse structure in a convolutional network using readily available dense components.
This has been achieved with a layer structure called **Inception module**, consisting of 3 filters: 1x1, 3x3 and 5x5, followed by a 3x3 max pooling, all used in parallel in a single layer, whose outputs are then concatenated.
The different-sized filters allow the module to capture features of different sizes and at different abstraction levels. In its näive form(Figure 1.16a), the 5x5 filters are problematic as they are very expensive for a convolutional network with a high number of filters. Thus, linear convolutions are applied before 3x3 filters and after pooling(Figure 1.16b) to reduce dimensionality before more complex operation, while also adding linear activation to the modules.
Inception modules are stacked on top of each other, occasionally separated by max pooling layers with stride of 2 to halve the resolution. All convolutions use the ReLU activation function.
This architecture allowed to further increase the depth of the network without



(a) The previous 5x5 filter is factorized into two consecutive 3x3 filters.

(b) Expanding filter banks in width avoids representational bottlenecks.

Figure 1.17: The updates to the inception module made in **InceptionV3**.

causing computational overloads. The final GoogLeNet network was composed of 27 weight layers, which made it susceptible to the *vanishing gradient problem*[3]. The issue was addressed by adding two **auxiliary classifiers** connected to the outputs of two inception modules, one lower in the network, the other higher.

InceptionV3 -the one used in our application- was introduced in 2015, along with version 2 in the same paper[54], with the main objectives of introducing some optimization over the previous model:

- **Smart factorization**: It was observed that convolutions with large spacial filters like 5x5 or 7x7 could be split into multiple layers of 3x3 convolutions (figure 1.17a), maintaining expressiveness and reducing computational cost. These can be further split into an asymmetric convolution using a 1x3 filter followed by a 3x1. Factorization can reduce computational costs, which can be instead be used to increase the network's filter bank sizes. Splitting so many filters can induce drastic dimensionality reduction which leads to information loss. These are referred to as **computational bottlenecks**,

[3]During **backpropagation** in deep neural networks, the gradient tends to become increasingly small, which can results in the weights being barely updated, if at all. It was first discussed by Josef Hochreiter in 1991[26]

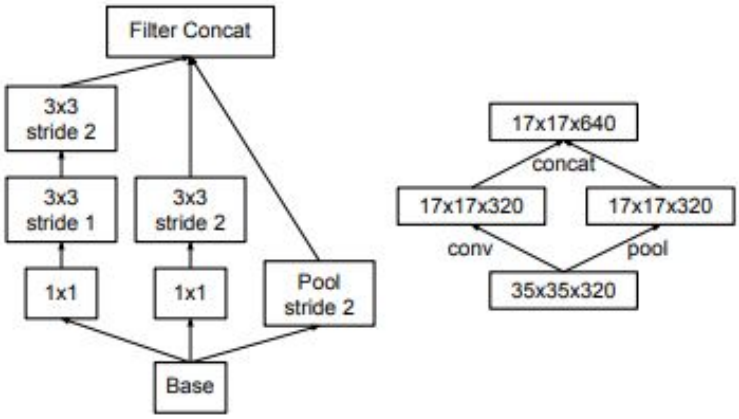


Figure 1.18: Inception module that reduces grid size and expands filter banks. The left size show operations, while the right one is focused on grid sizes.

and are avoided by expanding filter banks in width rather than size, as seen in figure 1.17b.

- **Auxiliary classifiers**: Further test with the previous auxiliary classifiers denoted that they did not improve convergence early in the training, only giving better results near the end of the training. Furthermore, removing the lower classifier did not affect the accuracy of the network. Overall performance of the main classifier was increased by adding batch normalization to the auxiliary one, suggesting that it acts more as a *regularizer*.

- **Efficient grid size reduction**: When trying to reduce grid size, the module employs two parallel stride 2 blocks: one for pooling and one for convolution (Figure 1.18). This operation is cheap and expands the filter bank while decreasing grid size, without introducing representational bottlenecks.

The keras implementation of Inception v3 is 48 layers deep, uses an input size of (299x299) and expects images with pixel values $\in [0, 1]$. Inception v3 achieves 21.2% top-1 and 5.6% top-5 error on the ILSVRC2012 dataset, showing overall better results than VGG without requiring as much computation, at the cost of a rather complex architecture, better shown in Figure 1.19.

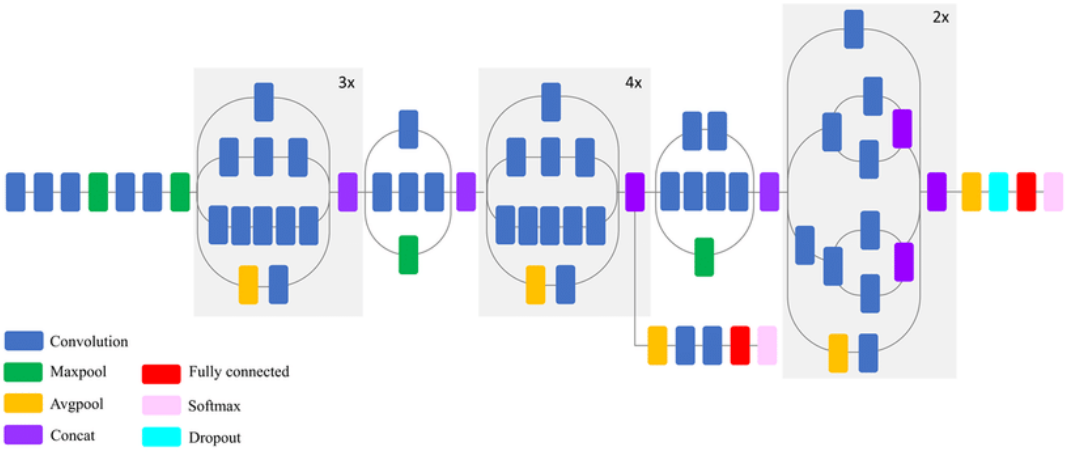
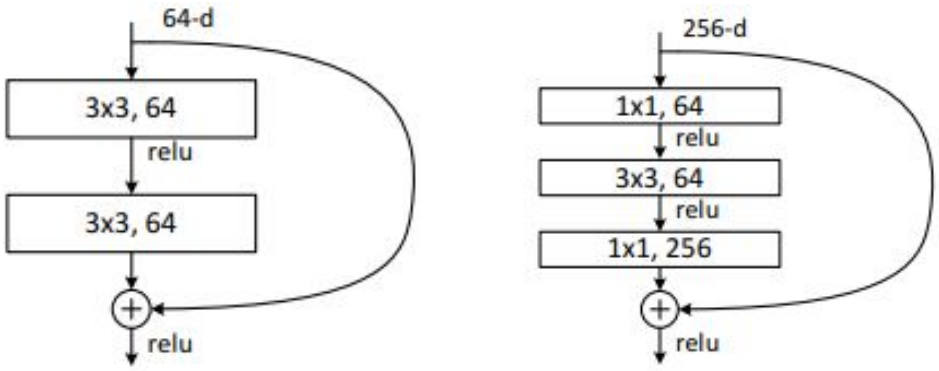


Figure 1.19: Schematized architecture of inception v3.

1.4.3 ResNet

ResNet, presented for the first time in 2015 [25], is based on the **residual network** architecture and aims to ease the training process of deeper neural networks.
Residual networks contain *shortcuts*, where the output of a layer skips one or more layers and is added directly to that of a deeper layer(Figure 1.20.
The shortcuts employed by ResNet use **identity mappings**: functions that return the same value as their inputs. As such, they simply propagate a gradient onto a deeper layer, conserving information. These shortcuts also add no parameters or computational cost, they are used early in the training phase, while the skipped layers are gradually enabled as the model learns the feature space. This not only speeds-up the training process by effectively using fewer layers at the start, but also passes unfiltered information to the deeper layers, which mitigates the *vanishing gradient* problem. Applying this technique allows ResNet to greatly benefit from added depth to the network.
In its publication paper, ResNet highlights the problem of *accuracy degradation*: after the model converges, accuracy gets saturated and degrades rapidly, increasing



(a) **(b)**

Figure 1.20: The residual blocks used in ResNet34 (a) and Resnet50(b).

training error. Tests showed that this behaviour was surprisingly not caused by overfitting, as even adding identity layers ended up increasing the training error (Figure 1.21).
The original ResNet architecture consist of 34 3x3 convolution layers, followed by a pooling layer and a fully connected layer, with shortcuts skipping over 2 layers. The version used in this project is ResNet50, made of 50 layers with 3-layer skips, dividing the networks in residual blocks composed of:

- a 1x1 ReLU convolution of 64 channels.
- a 3x3 ReLU convolution of 64 channels.
- a 1x1 ReLU convolution of 264 channels.

The final architecture looks like this:

1. a 7x7 convolution layer of 64 channels, with stride 2.
2. a 3x3 max pooling layer with stride 2
3. stacks of 3, 4, 6 and 3 residual blocks, with the number of channels doubling after each stack.

4. an average pooling layer, followed by a 1000 node fully connected layer and a softmax layer.

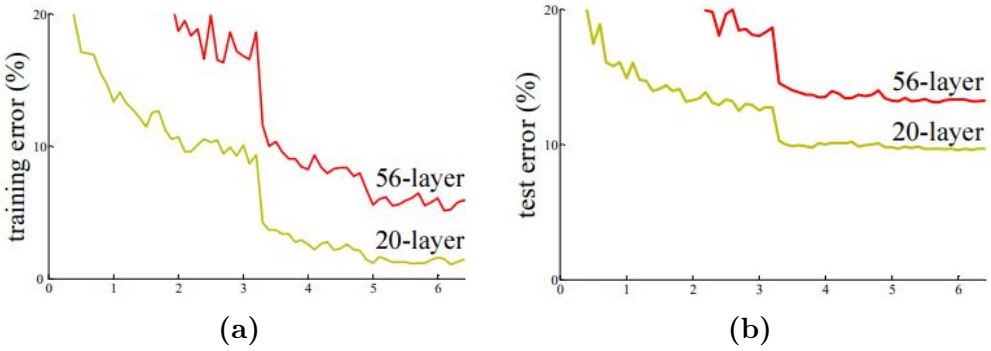


Figure 1.21: training error (a) and test error (b) on CIFAR-10 with networks that do not use residual connections. The deeper Layer presents higher training error(and test error).

The keras implementation of ResNet50 is 50 layers deep, uses an input size of (224x224) and expects images with pixel values $\in [0, 1]$.

1.4.4 MobileNet

Introduced in 2017 by Google[27], aims to make accurate convolutional neural networks relative to size and speed, to be usable in mobile systems and applications where recognition tasks have to be performed in a timely fashion, like robotics, self driving cars and augmented reality.

The model is based on **depthwise separable convolutions**, which require two separate layers, one for filtering inputs and one for combining them. This drastically reduces computation time and model size. Consider an input feature map $L_i = h_i \times w_i \times d_i$ and an output feature map $L_j = h_j \times w_J \times d_j$, where h_i and h_j are the heights, w_i and w_j are the widths, and d_i and d_j are the channels of, respectively, the input and output feature maps. A standard convolutional layer uses a

kernel K of size $k \times k \times d_i \times d_j$, and introduces a computational cost of:

$$C_{conv} = h_i \cdot w_i \cdot d_i \cdot d_j \cdot k^2 \tag{1.9}$$

Where k is an integer. By separating the convolution in two operations, we can break the multiplication between the output size D_f and the kernel size D_k. The two operations are:

- **depthwise convolution**: applies a single filter per input channel by a 3x3 kernel, with a computational cost of:

  $$C_{depth} = h_i \cdot w_i \cdot d_i \cdot k^2 \tag{1.10}$$

  making it much more efficient than a standard convolution. However, it does not combine the outputs of the filters, which means it cannot generate new features.

- **pointwise convolution**: computes a linear combination of the depthwise output using a 1x1 convolution, allowing it to generate new features, with a final cost of:

  $$C_{point} = h_i \cdot w_i \cdot d_i \cdot d_j \tag{1.11}$$

The total cost of a depthwise separable convolution is the *sum* of (1.10) and (1.11):

$$C_{tot} = h_i \cdot w_i \cdot d_i(k^2 + d_j) \tag{1.12}$$

Compared to a standard convolution, it is **faster by a factor of k^2**, 9 times using 3x3 kernels [46], with only a marginal accuracy reduction.
The described layers are used in the entire MobileNet architecture, except for the first layer, which is a full convolution. Mobilenet also provides 2 *hyperparameters* to further reduce model size:

- **Width multiplier α**: Directly multiplied to the number of input and output channels in every layer. α ranges between 0 and 1, with $\alpha = 1$ being the default value. It can reduce the computational cost and the number of parameters by roughly α^2.

- **Resolution multiplier ρ**: Used to change the size of the input image, it is implicitly set by the input resolution. It ranges between 0 and 1 with the default value being one. Like the width multiplier, it can reduce computational cost by ρ^2.

Our project uses MobileNetV2 [46], released 2 years later, which introduces a new block called the **Inverted residual with linear bottleneck**[4]**(MBConv Block)**, shown in Figure 1.22, based on the *manifold hypothesis*, which states that real-world high-dimensional data lies on low-dimensional manifolds embedded within the high-dimensional space[17].
Assuming that the manifold of interest lies in a low-dimensional subspace of the input space and that enough channels are used, non-linear transformations (like ReLU) will not cause the loss of that information, and the manifold can be captured by a linear bottleneck.
MBConv blocks are implemented by adding a 1x1 ReLU convolution, which expands the channels, then a depthwise 3x3 convolution, followed by a 1x1 linear convolution, that acts as a bottleneck. Furthermore, every bottleneck layer propagates its output directly to the next one, similarly to residual networks, but as the name suggests, they *invert* the channel narrowing and expanding operations. This design proved to be more efficient and improve overall performance.
A new hyperparameter is also added: the **expansion factor** t, referred to as the ratio between the size of the inputs in the bottleneck and expansion layers, with a default value of 6. The final architecture of MobileNetV2 is the following:

[4] "bottleneck" refers to a layer with a low number of channels

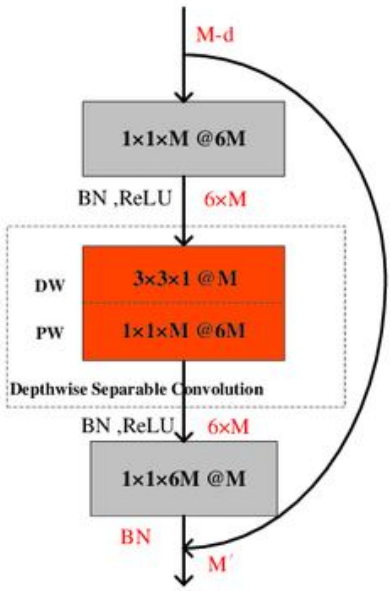


Figure 1.22: An *inverted* residual block.

1. A full convolution layer with 32 filters.

2. 19 MBConv blocks, with activation function ReLU6.

3. A 7x7 average pooling layer

The keras implementation of MobileNetV2 is 48 layers deep, uses an input size of (224x224) and expects images with pixel values $\in [0, 1]$.
MobileNet is one of the best networks for mobile applications, being both very light and accurate.

1.4.5 DenseNet

Densenet is a recent (2018) model inspired by ResNet's identity mappings, based on the observation that shorter connections between layers close to the input and to the output greatly benefit accuracy and efficiency[28]. **Dense connections** link each layer to every other deeper layer with matching feature map size, in a feed-forward fashon, resulting in $\frac{N(N+1)}{2}$ connections given N layers.

As layers receive feature maps from their predecessors, the layers can be more narrow, i.e. consisting of fewer channels.
Consider $\mathbf{x}_i$ as the output feature map of the i^{th} layer, were $i \in [0, ..., N]$, the i^{th} layer receives input feature maps from all the preceding layers:

$$\mathbf{x}_i = \mathbf{H}_i([\mathbf{x}_0, \mathbf{x}_1, ..., \mathbf{x}_{i-1}]) \tag{1.13}$$

Where $[\mathbf{x}_0, \mathbf{x}_1, ..., \mathbf{x}_{i-1}]$ is the concatenation of feature maps produced in layers $[0, ..., i-1]$, and where $\mathbf{H}_i(\cdot)$ is a composite function of three consecutive operations: batch normalization, ReLU and a 3x3 convolution, applied in layer i.
For the concatenation defined in (1.13), all the feature maps need to share the same size. Because of this, the architecture was divided in **dense blocks** with dense connections, separated by **transition layers** which apply batch normalization, 1x1 convolution and 2x2 average pooling. Like in ResNet, 1x1 convolutions are introduced before 3x3 convolutions as *bottleneck* layers, to reduce the number input features, improving efficiency. DenseNet configurations that use bottleneck layers are called *DenseNet-B*.
DenseNet uses two main hyperparameters:

- **growth rate k**: defined as the number of feature maps produced by each function $\mathbf{H}_i$, determines the number of channels added by each layer. We can calculate the number of input channels k_i for the layer i as:

$$k_i = k_0 + k \times (i - 1) \tag{1.14}$$

  where k_0 is the number of channels in the input layer.

- **compression factor $\boldsymbol{\theta}$**: Used to reduce the number of feature maps at transition layers. If a dense block contains m feature maps, the following transition layer will generate $\lfloor \theta m \rfloor$ output feature maps. θ ranges between 0 and 1, with a default value of 0.5. Those with $\theta < 1$ are called *DenseNet-C*.

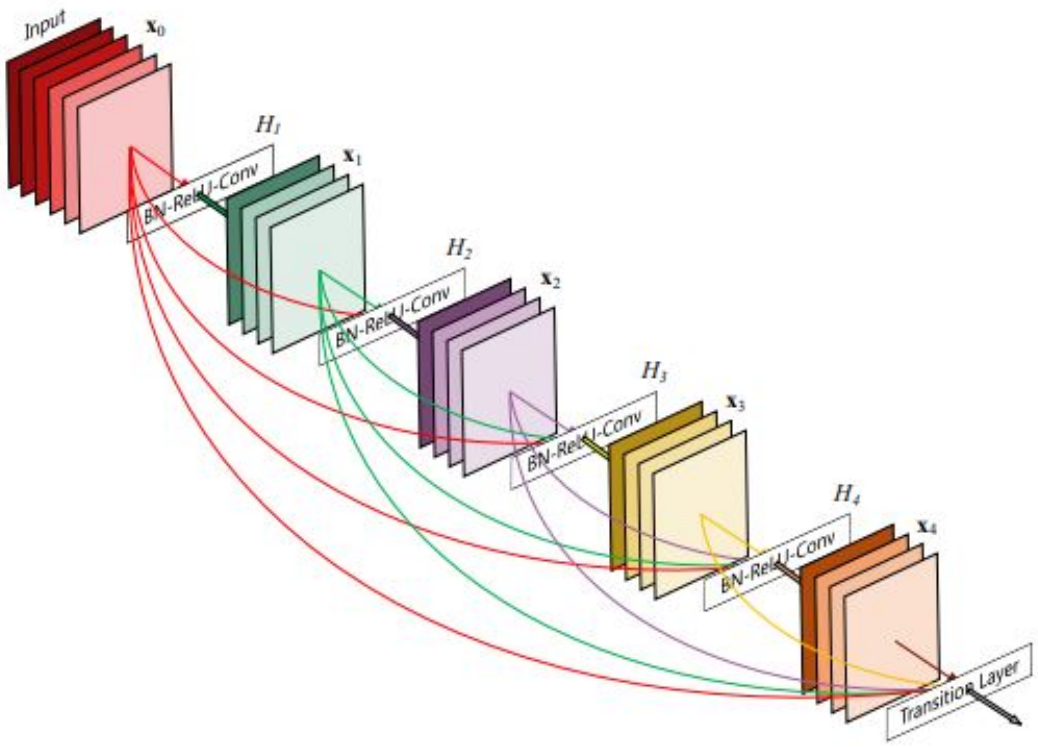


Figure 1.23: A 5-layer dense block with $k = 4$.

DenseNet networks with bottlenecks and $\theta < 1$ are referred to as *DenseNet-BC*. To summarize, the final building block, which we will refer to as a **BC-block**(Figure 1.23), is structured as follows:

1. A **bottleneck layer** with k_i input channels, that applies batch normalization, ReLU and 1x1 convolution.

2. A **composition layer** that applies batch normalization, ReLU and 3x3 convolution, with k output channels.

In our project, we used the DenseNet201 configuration, specifically made for tests on the ILSVRC2012 dataset, using $k = 32$ and $\theta = 0.5$. Its architecture(Figure 1.24) consists of 201 layers:

1. A first layer that applies $2k$ 7x7 convolutions with stride 2, followed by a 3x3 pooling with stride 2.

2. 4 dense blocks made of respectively 6, 12, 48 and 32 BC-blocks.

3. 3 transition layers, each after one of the first 3 dense blocks. They apply 1x1 convolutions followed by 2x2 average pooling with stride 2.

4. A final 7x7 global average pooling layer, followed by a fully connected layer and a softmax layer.

The Keras implementation of this model uses an input size of (224x224) and expects images with pixel values $\in$ [0,255].
DenseNet scales naturally to hundreds of layers while remaining a lightweight model, with no optimization difficulties. It achieves results on par with state of the art ResNets(21.46 top-1, 6.34 top-5), but with significantly lower parameters.

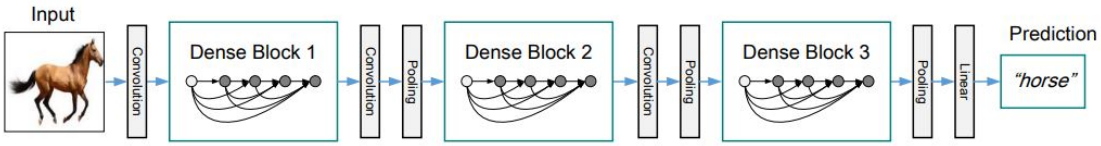


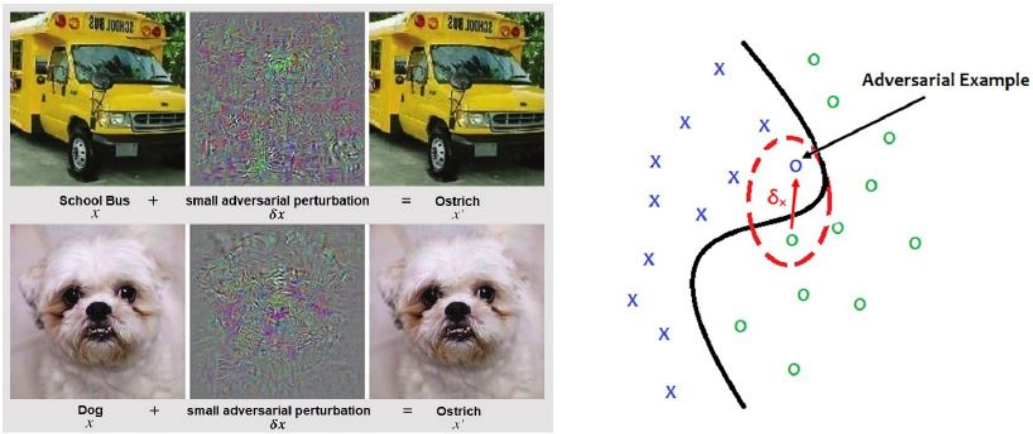
Figure 1.24: Schematized densenet architecture

2 Adversarial Machine Learning

Adversarial machine learning is a field that studies the **security weaknesses** of machine learning models, how to exploit them by providing **deceptive inputs**, and ways to defend against such attacks.

Applications susceptible to adversarial attacks range from email spam filtering, where inserting "good" words may cause a message to not be detected by a filter, to self driving cars, where subtle changes to a photo of a stop sign could cause the car to classify it as something else, and behave dangerously.

The first surveys about security issues in machine learning were published as soon as 2006 [8]. While non-linear classifiers were believed to be more robust to adversarial attacks, another paper by Christian Szegedy [55] noted that a neural network can be made to misclassify an image by applying small, imperceptible perturbations to it. The modified input is called an **adversarial example** (Figure 2.1a). Given a classifier g and a legitimate image x, an adversarial example is an image x' such that $x' = x + \vec{\delta}x$, where $\vec{\delta}$ is a vector that specifies the direction and magnitude of the perturbation needed to translate the image x beyond its class boundary in the decision space (Figure 2.1b), resulting in $g(x) \neq g(x')$ [36].



(a) **(b)**

Figure 2.1: Creation of an adversarial example(a), along with a visualization of the class boundary in the decision space(b)[36]

Another finding from [55] is that the same perturbations could be *transfered* to other models and still damage them successfully. This phenomenon, called **transferability**, can be defined as the property of some adversarial to mislead not only a target model g, but also other models g'. In the worst case, an attack may be applicable to models that solve entirely different tasks.

The reasons for the existence of adversarial attacks and their transferability are still very unclear. In general, learning models are believed to be able to solve complex problems without really learning any features from the data, which is what makes them susceptible to small perturbations. This is know as the *Clever Hans Effect*[1]. One notable theory is the *linearity* hypothesis[23], which suggests that neural networks are actually very linear, and that the direction of the perturbation is the crucial factor that makes transferability possible. We will not discuss the issue further, but interested readers can also refer to [20], [30], [48] and [56].

In the rest of this section we will define a taxonomy to categorize adversarial attacks and look at some of the algorithms used to generate adversarial examples, focusing on the one our application is based on.

2.1 Adversarial threat model

Defining a threat model requires us to know which information about our machine learning model is available to the attacker, and what are his goals. Inspired by [36] and [59], we will describe threat models using the following points:

1. **Attacker's influence**: Determines how the attacks will affect the model. **Poisoning** attacks introduce adversarial examples in the training data to produce a classification model incompatible with the original data distribution. The more common **evasion** attacks target the test phase, creating false

[1] Clever Hans was a horse believed to be able to answer arithmethic questions by tapping his hoof a specific number of times. After several experiments, psychologists deducted that he simply learned to stop tapping when people would cheer at him.

negatives that are undetected by the model, to gather information about it or simply cause misclassification.

2. **Attacker's knowledge**. In **white box** attacks, the attacker has access to the training data, model architecture, hyperparameters, number of layers, activation function and weights. When the attacker has only access to model's output, he will launch a **black box** attack. *White box* attacks can usually be transfered to *black box* scenarios by creating a *surrogate* of the target model and using it to create adversarial examples, by exploiting *transferability*.

3. **Attack specificity**: Given a clean input x, **Targeted** attacks craft examples x' such that $g(x') = y'$, where y' is a target label. **Untargeted** attacks accept any classification that ensures that $g(x') \neq g(x)$. Untargeted attacks are easier to implement and less computationally expensive.

4. **Attack frequency**: the computational logic of the attack algorithms. **Sequential** attacks generate the perturbed input in one iteration, while **iterative** attacks use more than one. The latter are more expensive than the former, but also more effective.

5. **Perturbation scope**: in **individual** attacks, each example x'_i is created using a different perturbation δ_i. **Universal** attacks apply a single δ to all examples x'_i, and are usually more applicable to the real world.

As adversarial examples are designed to be imperceptible by humans, the perturbation needs to be as small as possible while still being able to disrupt a machine learning model. To control the magnitude of a perturbation $\vec{\delta}$, we can define several measures, the most common being the ***p*-norm distance** L_p, which computes the distance between a clean example x and and an adversarial example x' in the solution space:

$$L_p = \left(\sum_{i=1}^{n} |x - x'| \right)^{\frac{1}{p}} \tag{2.1}$$

where n is the dimensionality of the examples, and $p \in \{0, 1, 2, \infty\}$. For $p = 0$, L_p counts the number of modified pixels in x'. $p = \infty$ denotes the maximum difference between all corresponding pixels in x and x' over the n dimensions:

$$L_\infty = max\Big\{|x_1 - x'_1|, |x_2 - x'_2|, ..., |x_n - x'_n|\Big\} \tag{2.2}$$

for $p = 1$ and $p = 2$ the expression corresponds respectively to the *Manhattan* and *Euclidean* distances. Being able to control the amount of perturbation allows us to define the adversarial attack as an **optimization** problem, where we find the lowest L_p that allows us to fool the model without being detected by humans, or simply find a **threshold** value and have our perturbation respect it.

2.2 Generating adversarial examples

Most adversarial examples are generated by exploring generalization flaws in pre-trained models, usually through optimization algorithms that make use of perturbation measures like L_p. We will briefly present 4 algorithms used to attack neural networks:

2.2.1 Fast Gradient Descent Method(FGSM)

(FGSM)[23] is a *sequential* algorithm inspired by the linearity hypothesis that executes a single perturbation in the direction of the gradient of the loss function.

$$x' = x - \varepsilon \cdot sign(\vec{\nabla}_x J(\theta, x, y)) \tag{2.3}$$

Where $\vec{\nabla}_x$ is the gradient vector, ε is an hyperparameter that expresses the maximum perturbation magnitude, and J is the loss function of the neural network. $sign(x)$ is the *sign* function, that returns 1 for positive inputs and -1 for negatives. FGSM has low computational cost, but its perturbations are easy to detect.

2.2.2 Basic Iterative Method(BIM)

BIM[34] is an *iterative* version of FGSM that restricts the amount of perturbation by an upper bound ε. It can be recursively defined as:

$$x' = \begin{cases} x'_0 = x \\ x'_{i+1} = Clip_{x,\varepsilon}(x'_i + \alpha \cdot sign(\vec{\nabla}_x J(\theta, x'_i, y))) \end{cases} \tag{2.4}$$

where α is the step size, i $\in 0, ..., \alpha - 1$ and $Clip_{x,\varepsilon}$ is the function that restricts the perturbation to an ε-neighborhood of the source image, defined in [34].

2.2.3 DeepFool(DF)

DF[40], also based on the linearity hypothesis, is an *untargeted, iterative* attack optimized for the L_2 distance. It finds the closest point in the decision boundary, then subtly perturbs the image in that direction at each step. DF was formulated by finding the best perturbation in a linear approximation of the target model, then applying the solution to higher levels of non-linearity until the right adversarial example is produced. We will not get into the formalities of the algorithm, as it is rather articulated. DeepFool is able to compute more subtle perturbations compared to FGSM.

2.2.4 Carlini & Wagner(CW)

CW[12] is a *targeted, iterative* attack that relies on gradient descent to find a correct adversarial inpt. The algorithm tries to generate a perturbation δ such that x' will be misclassified with the label y'. Given a neural network g with an *inverse sigmoid*(logit) layer z, it uses a loss function J defined as:

$$J(x') = max(max(Z(x'))_y : y \neq y') - Z(x')_{y'}, -k), \tag{2.5}$$

where

- $max(Z(x'))_y$ is the highest probability among non y' predicted classes.
- $Z(x')$ is the probability of y'.
- k is an hyperparameter that expresses with how much confidence we want x' to be misclassified.

Given a distance measure L_d, CW tries to minimize $L_d(x, x')$ given the constraint of $g(x') = y$. This problem is difficult to solve for many algorithms, as the constraint is very non linear, but rewriting it using the cost function J makes it much more feasable:

$$\text{minimize } \{L_d(x - x') + c \cdot J(x')\} \tag{2.6}$$

Where c is an hyperparameter used to compute the minimum required perturbation magnitude. CW is considered the state-of-the-art algorithm for generating adversarial images. Compared to FGSM, the attack will always succeed without the need of a perturbation threshold. It is also much stronger, albeit more costly.

2.3 Defenses

Designing countermeasures to adversarial attacks has proven to be a difficult task, as there are no theoretical tools to describe solutions to the optimization problem presented by the attacks. Most of the currently experimented defenses are only effective for specific models or against certain attacks. They are generally classified according to their *objective*:

- **proactive** defenses aim to make the model more robust, so that it classifies adversarial examples like their clean counterparts
- **reactive** defenses try do detect the adversarial examples in advance, acting as filters.

Looking in detail at single defense mechanisms goes beyond the scope of this thesis, as the models we will be attacking do not employ any.

2.4 Universal Unrestricted attacks with AGV

AGV is the algorithm interfaced by our application, presented in [7] as a *universal*, *unrestricted*, *untargeted* attack for *black box scenarios*. AGV uses a multi-objective evolutionary algorithm to select 5 instances from a pool of *image-agnostic* photographic filters and uses them to generate adversarial examples. The term **unrestricted** refers to the perturbation applied by these filters: most of the previously cited attack algorithms operate by adding noise and/or altering specific pixels, creating unnatural looking images which force attackers to impose an L_p bound to the perturbation amount δ. The filters used by AGV affect the entire image equally, preserving its semantic features, allowing for much stronger perturbations without raising suspicion in a human observer. In order to be more applicable to real-world scenarios, the algorithm does not use any information about the targeted model, and a single filter sequence can be used for multiple images.

2.4.1 Three-step process

AGV works by employing two nested evolutionary algorithms, then by evaluating their attacks against a target model:

1. The first algorithm is tasked with **choosing the correct sequence of filters**. Given a set $S = \{f_1, f_2, \ldots, f_m\}$ of m image filters, it finds a candidate sequence $\hat{S}$ of 5 filters and encodes them as a list of integers.

2. The second algorithm **finds the optimal parameters** for each filter in $\hat{S}$. These parameters are called intensity α and strength s, and are used to determine the magnitude of the perturbation applied by each filter. They will be defined in section 3.3, when looking at how the single filters were

created manually. The parameters are encoded into a list of decimal values.

3. Every new candidate sequence $\hat{S}$ is used to generate multiple adversarial images from a dataset of size n, to attack the target model. Given a target network g, the original dataset X and the adversarial dataset $\hat{X}$, the attack is modeled as a multi-objective problem that uses two objective functions: **attack success rate(ASR)**, defined as

$$ASR(X, \hat{X}) = \frac{1}{n} \sum_{i=0}^{n} g(x_i) \neq g(\hat{x}) \tag{2.7}$$

and the **Detection Rate(DR)**, defined as the average

$$DR(\hat{X}) = \frac{1}{n} \sum_{i=0}^{n} D(\hat{x}) \tag{2.8}$$

where D is any reactive defense mechanism, referred to as *detector*, that returns 1 if the attack is detected, otherwise 0. Thus, the multi-objective problem can be defined as:

$$minimize\ \mathcal{F}(X, \hat{X}) = \{1.0 - ASR(X, \hat{X}),\ DR(X, \hat{X})\} \tag{2.9}$$

Where $\mathcal{F}$ is the solution space. The equation 2.9 tries to maximize the attack success rate while minimizing the detection rate, to find the best sequence $\hat{S}$ to attack g.

The next chapter will be dedicated to building an interface to visualize the adversarial examples created by AGV and using them to manually attack popular CNNs.

3 AGV Web Application

This chapter describes the ideas and process behind the development of the web interface for the AGV algorithm. Our objective is to create a **scalable** software with a **user-friendly** interface to easily create and test adversarial examples like the ones used by AGV , with the ability to see the generated images before using them to attack a network. Users should also be able to **create** their own attack models by using combinations of one or more filters, and to tune parameters for each one. We will start by talking about the tools and frameworks used for developing the software. Afterwards, we will introduce the *ImageNet* dataset, which is the source of our images. Lastly, we will discuss how we implemented the image filters used by AGV , before presenting the graphical user interface(GUI) for our application.

3.1 Tools

The language we chose for our application is **Python 3.8**, as it offers several libraries suited for our task:

- The web application back-end was developed using **Flask 1.1**, a lightweight web framework that uses the *Werkzeug* WSGI and the *Jinja2* template engine. We found flask's simplicity to be suited for the application.
- **Keras 2.3**, an open source python library used as an interface to lower-level machine learning frameworks, like **TensorFlow 2.4**. Both libraries are extensively used in the industry, as well as research. Keras contains implementations for the neural networks we will be targeting with our attacks.
- The image filters for creating adversarial examples were written using **OpenCV 4.4**, a multiplatform, open source library for computer vision, along with **Pillow 8.0**, for simple image manipulation.

The web interface was built in *html5*, using the *css* library *Bootstrap*, along with *JavaScript* and *jQuery* for front-end scripting.

3.2 The ImageNet dataset

ImageNet is a free image database organized using the **WordNet** hierarchy: WordNet is organized in a tree structure, were entries are grouped by similarity of meaning in units called *synets*[19]. Imagenet aims to provide an average of 500-1000 images to illustrate each synet, by labeling and organizing them into different tree structures. The goal of imagenet is to provide the most comprehensive and diverse coverage of images on the net, while keeping accuracy high even on the lower levels of the hierarchy.

The dataset is constructed by collecting thousands of candidate images from several search engines, querying sets of synonyms for each synet, in multiple languages including Chinese, Spanish, Dutch and Italian. ImageNet does not own copyright to any of its images, but simply provides thumbnails and URLs[14].

Every candidate image is then verified and labeled by humans, using *Amazon Mechanical Turk*. The same image is labeled by multiple users, to minimize accuracy errors, with more complex categories requiring higher levels of consensus.

According to the latest reportings on the Imagenet official website, the database hosts more than 14 million images, spanning over 20,000 synets of 27 high-level categories. Over 1 million images have been annotated with bounding boxes, and 1000 synets with SITF features, for object localization and detection tasks [1].

Prior to ImageNet, many algorythms were only trained to recognize specific objects, with comparatively limited datasets. ImageNet was the first image dataset of its scale and allowed the training of models capable of recognizing many different objects, while greatly improving their overall accuracy. [37]. ImageNet has become a staple in the field of computer vision and proved that good accuracy depends not only on the model, but also on the quality of the training data.

In our application, we use a subset of the dataset used in the *ImageNet Large Scale Visual Recognition Challenge 2012(ILSVRC2012)*[43], that has been used to test different neural networks, including ones used in this project, and can be downloaded for free[1]. The weights of the Keras models used in our application have been pre-trained on the ILSVRC2012 dataset, and use the same class label mappings as the *Caffe framework*[32][2]. This gives us a convenient way to access the ground truth class for each image, and check if they have been predicted correctly by our models.

3.3 Image Filters

The first task was to write the image filters to be used by AGV for creating adversarial images. These filters need to apply an uniform effect over the target image, reminiscent of ones that could be produced by anyone with a simple photo processing application or program. We were inspired by Instagram's photo filtering interface [2], and tried to emulate *five* of its most popular filters.

Due to the exact modifications applied by Instagram's filters not being publicly available, we conducted research on photography blogs [3], and manually tested Instagram's original filters on different images. After that, we deducted what different photographical effects any filter would apply to an image, and in what order. We also stuck to the *classic* image filters, as the story filters in the Instagram app can not be applied on static images, and would have been too hard to test, understand, and reference for further studies.

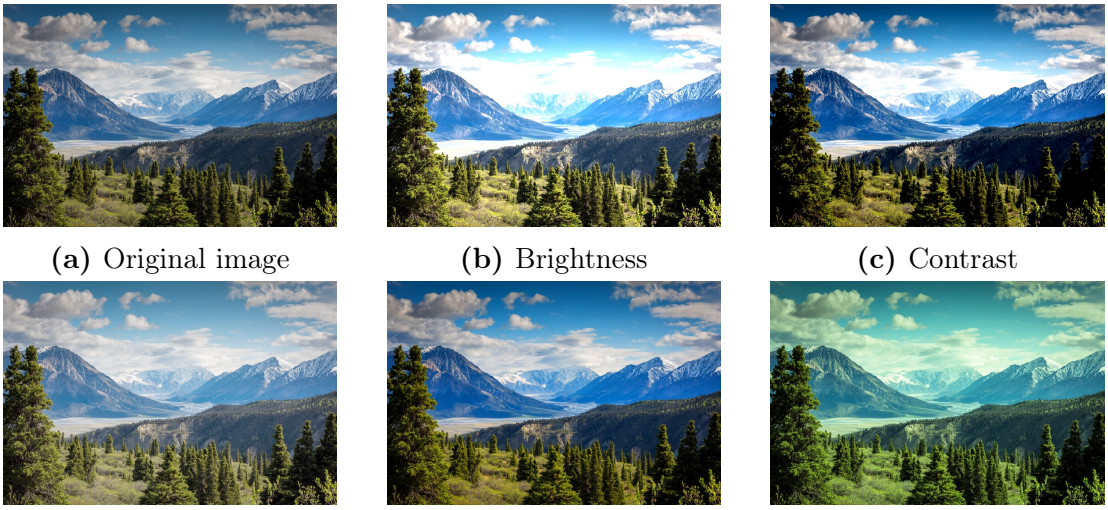
The task was accomplished through the development of the **filters module**, using the *OpenCV* and *Pillow* libraries. We started by creating simple photographic effects, that only change single aspects of an image, like *brightness*, *contrast*, *saturation*, *gamma correction* and *hue*(Shown in Figure 3.1). It is worth noting that

[1]http://www.image-net.org/download-images

[2]Can also be downloaded from http://dl.caffe.berkeleyvision.org/

[3]https://blog.iconosquare.com/top-10-instagram-filters/

the order in which they are applied will influence the final output.



(a) Original image **(b)** Brightness **(c)** Contrast

(d) Gamma correction **(e)** Saturation **(f)** Hue shift(green)

Figure 3.1: Some of the image effects applied by the *filters* module.

These effects were then used as building blocks for writing our interpretations of instagram's filters:

- **Clarendon:** increases contrast and saturation, then cools down the colors by reducing red values and enhancing the blues.
- **Gingham:** raises brightness and lowers contrast, giving a white tint to the image.
- **Reyes:** decreases contrast and saturation, then raises brightness and gamma. Finally, it slightly enhances the red and green tonalities.
- **Juno:** raises contrast, saturation and applies gamma correction.
- **Lark:** lowers the gamma, then applies a mask to selectively enhance the blue tonalities in the image, while reducing reds and greens.

(a) Clarendon(Instagram)

(b) Clarendon(filters)

(c) Gingham(Instagram)

(d) Gingham(filters)

(e) Reyes(Instagram)

(f) Reyes(filters)

(g) Juno(Instagram)

(h) Juno(filters)

(i) Lark(Instagram)

(j) Lark(filters)

Figure 3.2: Comparison between Instagram's filters and our implementations. The left column shows the original filters, while the right column shows the ones in the *filters* module. The filters are applied with $\alpha = 1$ and $s = 1$.

The filters can be applied in sequence, and can be customized using the parameters α and s we already mentioned when we described the AGV algorithm in section 2.4:

- **intensity $\boldsymbol{\alpha}$**: A decimal value directly multiplied to the *alpha* of the single effects in every filter, directly affecting the perturbation magnitude, e.g. raising α in clarendon would further increase the contrast, saturation and cold tonalities. We found that the best results are achieved in the range $0.8 < \alpha < 1.3$, as higher or lower values tend to denaturate images.

- **strenght $\boldsymbol{s}$**: After every effect has been applied, the filtered image x' is interpolated with the original one x using OpenCV's *cubic interpolation*, based on a ratio given by s. Every pixel in the final image x'' is computed as follows:

$$x''_{i,j} = (1.0 - s) \cdot x_{i,j} + s \cdot x'_{i,j} \tag{3.1}$$

  where (i, j) are the pixel indexes. s can vary between 0 and 1, where 1 yields a fully filtered image, while 0 returns an unmodified image.

AGV makes direct calls to the *filters* module to generate the new images. After running the algorithm, we can save the decoded filters and parameter values in a *json object*, making it possible to reuse the same sequence of filters to modify images manually. The filters are encoded under the key "filters", as a list of integer values {0: *Clarendon*, 1: *Gingham*, 2: *Reyes*, 3: *Juno*, 4: *Lark*}, while the parameters are encoded under "params", a list containing (s, α) pairs, each one mapped to their appropriate filter in "filters", appearing in the same order.

3.4 Graphical User Interface

The GUI is meant to be an interface to the *AGV* algorithm, to allow users to directly *visualize* and *evaluate* its outputs. At the same time, we want to be able to generate adversarial examples *independently* from the algorithm, choosing our own sequence of filters. It is divided in two main forms: the first one (Figure 3.3) is used to choose a **neural network model** from the ones presented in this thesis, along with an **input image**, either selected from a list of images from the ILSVRC2012 dataset, or **uploaded** by the user himself. The second form (Figure 3.4) will show the top-3 class predictions returned by the model for the previously selected image. It is also used to choose the **attack model**, either by selecting one from the two optimal models presented in section 2.4, or by creating a custom attack with the use of an **editor**. Using these two forms, the user is able to **classify** an image using a NN model and turn it into an adversarial example to **attack** said network. The second form will only be shown after running the class prediction. While it is possible to run multiple attacks on the same network/image pair subsequently, *changing any of the two will hide the second form and revert the procedure to the beginning.* The NN models are lazily loaded by the server and saved in RAM, so only the first prediction for every network will be slow.

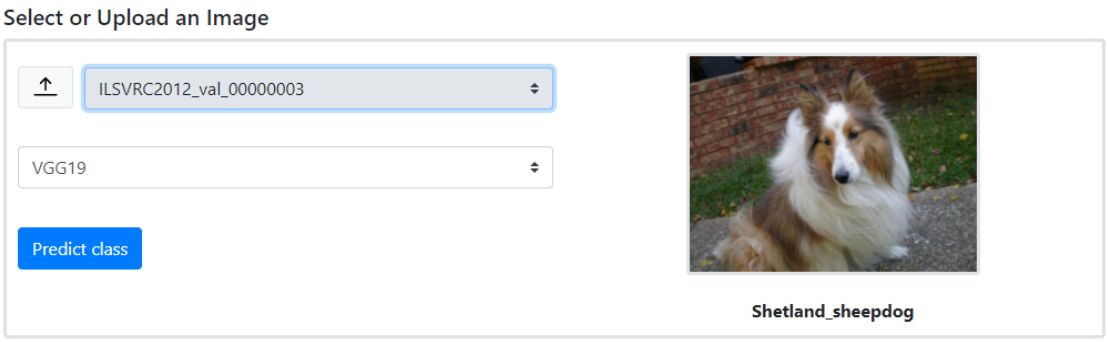


Figure 3.3: The first form.

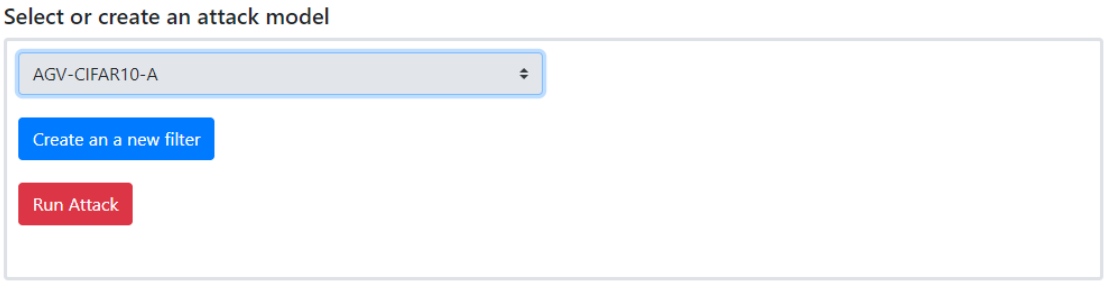
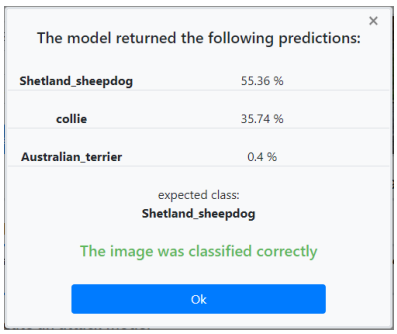


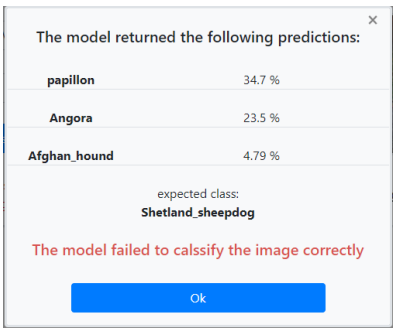
Figure 3.4: The second form.

3.4.1 Prediction using preset images

When the user selects a preset ILSVRC2012 image, it will be shown alongside its **ground truth class** name from the Caffe framework, which contains the same labels and weight mappings used by Keras for the ImageNet dataset. After running the prediction, a modal window will show the user the *top-3* class predictions returned by the selected network for the given image, along with their probability distribution (Figure 3.5). If the highest-confidence class matches the ground truth, the prediction is considered **successful**, and the second form will be shown. Otherwise, the prediction fails and the second form will not be shown, as running attacks on an already misclassified image would be pointless. The user will have to try another image/network combination.



(a) Correct prediction



(b) Incorrect prediction

Figure 3.5: Results of a prediction on the same image, using InceptionV3 **(a)** and VGG19. **(b)**

3.4.2 Prediction using custom images

By clicking on the **upload icon**, the user can choose any image from his computer to be classified. In this case, there is no prior way to determine the ground truth of the image, so it will be assumed to be the highest-confidence prediction returned by the model. Before running attacks, we allow the user to freely pick a different ground truth class among the top-3 by clicking their respective buttons (Figure 3.6).

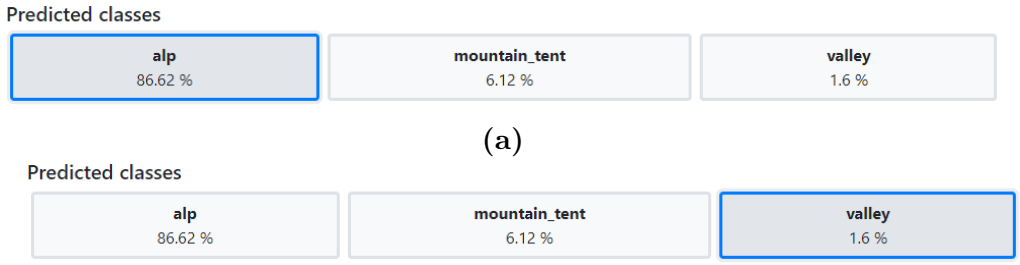


Figure 3.6: When uploading a custom image, the user can change ground truth at any time.

3.4.3 Attacking with preset models

The second form allows to choose between two pre-made attack models from a dropdown list. These models, which we called AGV-A and AGV-B, are implemented by *json* objects returned by AGV . As an example, AGV-A applies:

1. $Juno(s = 0.44,\ \alpha = 0.9)$.
2. $Lark(s = 0.5,\ \alpha = 0.9)$.
3. $Gingham(s = 0.35,\ \alpha = 0.93)$.
4. $Reyes(s = 0.18,\ \alpha = 0.63)$.

5. *Gingham*($s = 0.49$, $\alpha = 1.08$).

The selected model will be use to create a filtered image, which will be classified by the previously selected network. Since the attack is *untargeted*, it will be considered **successful** as long as the *top-1* prediction **differs** from the ground truth class. If the prediction does not differ from the original input, the attack fails.

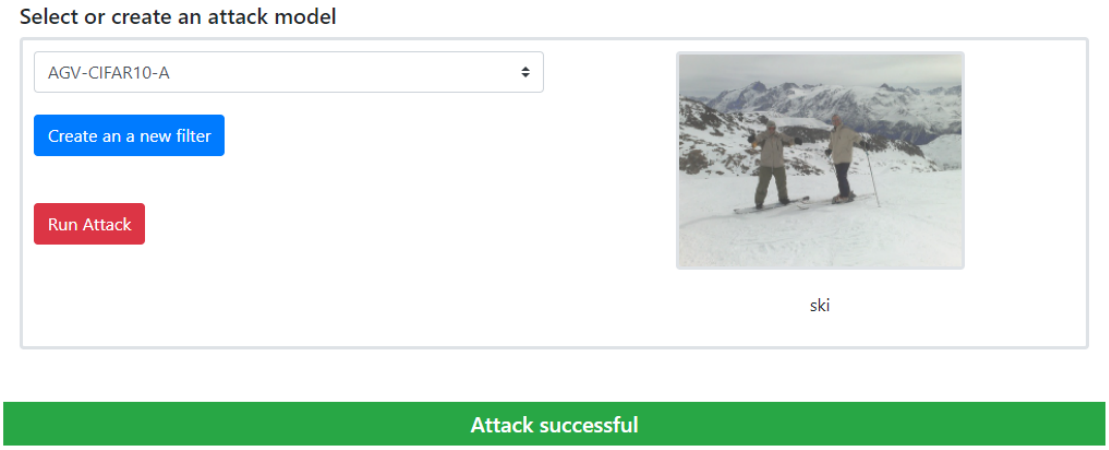


Figure 3.7: Successful attack with *AGV-A*.

3.4.4 Creating custom attacks

Users can also create their own attack models through the use of an editor interface (figure 3.8). The editor creates attacks exactly like *AGV* , allowing the concatenation of **multiple filters** from the ones presented in the previous section, and setting their **parameters**. α can range between 0.8 and 1.3 with a default value of 1, while s ranges between 0 and 1, with a default value of 1. The newly created attacks are encoded as *json objects*, like the ones described in section 3.3 and are added at the bottom of the attack selection list. They are also evaluated in the same way, requiring any form of misclassification to be considered successful. Their purpose is to allow the user to manually test different combinations without relying on the algorithm itself.

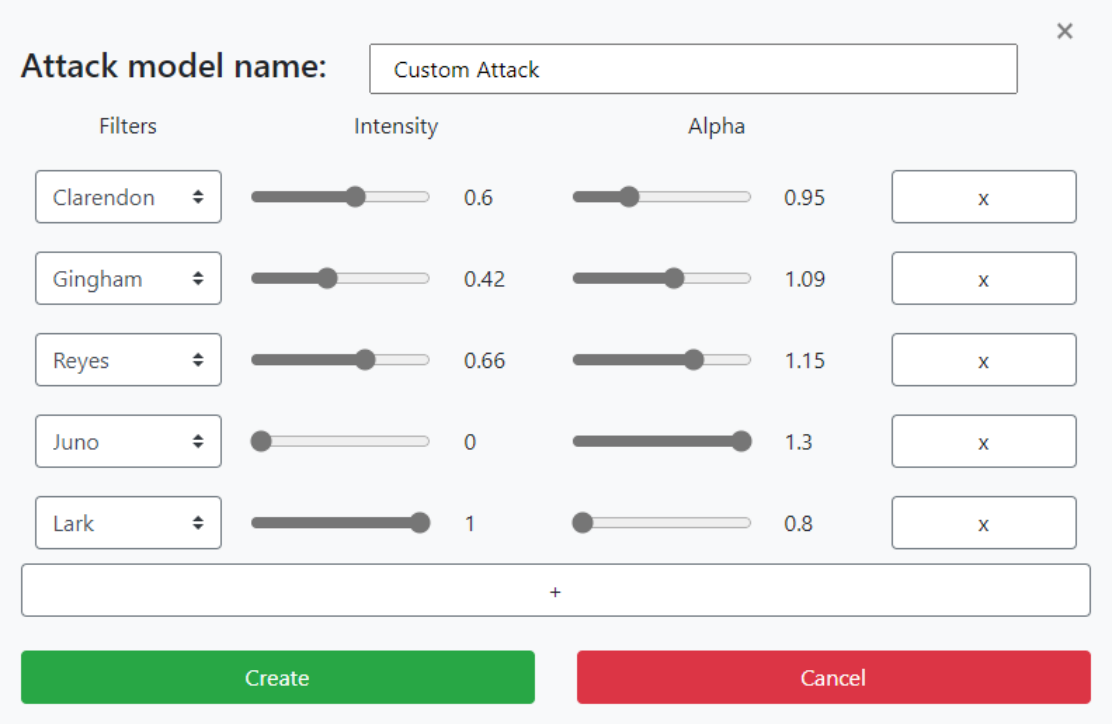


Figure 3.8: The attack model editor.

Conclusions

This thesis presented an overview on the workings of Convolutional Neural Networks and their evolution through the years. We also introduced weaknesses and security issues regarding these models, focusing on the most recent techniques employed to attack them. After analyzing potential limitations of these adversarial attacks, we experimented ways to generate adversarial examples by trying to recreate commonly used Instagram filters. We proposed a solution by using the AGV algorithm to find optimal sequences of filters, and developed a simple web application to manually apply the attack generated by the algorithm, as well as creating cusotm adversarial attacks. These attacks are able to generate successful adversarial examples on the selected neural network models.

Further improvements

The application has much room for improvement and expansion: more filters could be added and the already existing ones could be made more similar to the ones employed by instagram. Furthermore, the pool of network models could also be expanded, allowing a wider range of experiments. Scalability was considered a priority during development, as to allow for further improvements like these down the line. The source code can be found on GitHub, and is freely accessible under a MIT license: **https://github.com/LucaCeccagnoli/flask-attacks**.

Bibliography

[1] Imagenet: summary and statistics. http://image-net.org/about-stats, accessed: 2021-01-17.

[2] instagram.com. https://www.instagram.com/, accessed: 2021-01-07.

[3] Keras api reference. https://keras.io/api/applications/, accessed: 2021-01-20.

[4] Activation function., 2021. https://en.wikipedia.org/wiki/Activation_function,accessed:2021-01-20.

[5] O. I. Abiodun, A. Jantan, A. E. Omolara, K. V. Dada, N. A. Mohamed, and H. Arshad. State-of-the-art in artificial neural network applications: A survey. *sciencedirect.com*, 2018. http://www.sciencedirect.com/science/article/pii/S2405844018332067.

[6] Andy. Stochastic gradient descent – mini-batch and more. *adventuresinmachinelearning.com*, 2018. https://adventuresinmachinelearning.com/stochastic-gradient-descent/, accessed 2021-01-22.

[7] A. E. Baia, G. di Bari, and V. Poggioni. Effective Universal Unrestricted Adversarial Attacks using a MOE Approach. In Proceedings of the 24th International Conference on the Applications of Evolutionary Computation, EvoApplications 2021, April 2021, Seville, Spain, Part of EvoStar2021, to appear.

[8] M. Barreno, B. Nelson, R. Sears, A. D. Joseph, and J. D. Tygar. Can machine learning be secure? In *Proceedings of the 2006 ACM Symposium on Information, Computer and Communications Security*, New York, NY, USA, 2006. Association for Computing Machinery.

[9] M. Basavarajaiah. 6 basic things to know about convolution. *medium.com*, 2019. https://tinyurl.com/qr7dyzq, accessed: 2021-01-20.

[10] J. Brownlee. A gentle introduction to the rectified linear unit(relu). *machinelearningmastery.com*, 2020. https://tinyurl.com/ydxf3xao, accessed: 2021-01-21.

[11] J. Brownlee. How to accelerate learning of deep neural networks with batch normalization, 2020. https://tinyurl.com/ybjbnx5s.

[12] N. Carlini and D. Wagner. Towards evaluating the robustness of neural networks, 2017. https://arxiv.org/abs/1608.04644.

[13] G. Cybenko. *Approximation by superpositions of a sigmoidal function*. Mathematics of Control, Signals and Systems, 1989.

[14] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei. ImageNet: A Large-Scale Hierarchical Image Database. In *CVPR09*, 2009. http://www.image-net.org/papers/imagenet_cvpr09.pdf.

[15] A. Dertat. Applied deep learning-part 4: Convolutional neural networks. *towardsdatascience.com*, 2017. https://towardsdatascience.com/applied-deep-learning-part-4-convolutional-neural-networks-584bc134c1e2, accessed: 2021-01-20.

[16] M. B. et al. End to end learning for self-driving cars, 2016. https://arxiv.org/abs/1604.07316.

[17] C. Fefferman, S. Mitter, and H. Narayanan. Testing the manifold hypothesis, 2013. https://arxiv.org/abs/1310.0425.

[18] D. X. Fei-Fei Li, Ranjay Krishna. Cs231n: Convolutional neural networks for visual recognition. stanford. accessed 2021-01-20. *Stanford Vision and Learning Lab*, 2019. http://cs231n.stanford.edu/syllabus.html.

[19] C. Fellbaum. *WordNet: An Electronic Lexical Database*. Bradford Books, 1998.

[20] J. Gilmer, L. Metz, F. Faghri, S. S. Schoenholz, M. Raghu, M. Wattenberg, and I. Goodfellow. Adversarial spheres, 2018. https://arxiv.org/abs/1801.02774.

[21] X. Glorot, A. Bordes, and Y. Bengio. Deep sparse rectifier neural networks. In G. Gordon, D. Dunson, and M. Dudík, editors, *Proceedings of the Fourteenth International Conference on Artificial Intelligence and Statistics*. JMLR Workshop and Conference Proceedings, 2011. http://proceedings.mlr.press/v15/glorot11a.html.

[22] I. Goodfellow, Y. Bengio, and A. Courville. *Deep Learning*. MIT Press, 2016. http://www.deeplearningbook.org.

[23] I. J. Goodfellow, J. Shlens, and C. Szegedy. Explaining and harnessing adversarial examples, 2015. https://arxiv.org/abs/1412.6572.

[24] C. W. Hang. Neural network with backpropagation, 2019. https://github.com/Vercaca/NN-Backpropagation, accessed: 2021-01-25.

[25] K. He, X. Zhang, S. Ren, and J. Sun. Deep residual learning for image recognition, 2015. https://arxiv.org/abs/1512.03385.

[26] S. Hochreiter. Untersuchungen zu dynamischen neuronalen netzen. Diploma thesis, Institut f. Informatik, Technische Univ. Munich., 1991. https://tinyurl.com/y63xlj2h.

[27] A. G. Howard, M. Zhu, B. Chen, D. Kalenichenko, W. Wang, T. Weyand, M. Andreetto, and H. Adam. Mobilenets: Efficient convolutional neural networks for mobile vision applications, 2017. https://arxiv.org/abs/1704.04861.

[28] G. Huang, Z. Liu, L. van der Maaten, and K. Q. Weinberger. Densely connected convolutional networks, 2018. https://arxiv.org/abs/1608.06993.

[29] D. H. Hubel and T. N. Wiesel. Receptive fields and functional architecture of monkey striate cortex. *The Journal of Physiology*, 1968.

[30] A. Ilyas, S. Santurkar, D. Tsipras, L. Engstrom, B. Tran, and A. Madry. Adversarial examples are not bugs, they are features, 2019. https://arxiv.org/abs/1905.02175.

[31] S. Ioffe and C. Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift, 2015. https://arxiv.org/abs/1502.03167.

[32] Y. Jia, E. Shelhamer, J. Donahue, S. Karayev, J. Long, R. Girshick, S. Guadarrama, and T. Darrell. Caffe: Convolutional architecture for fast feature embedding, 2014. https://arxiv.org/abs/1408.5093.

[33] A. Krizhevsky, I. Sutskever, and G. E. Hinton. Imagenet classification with deep convolutional neural networks. In F. Pereira, C. J. C. Burges, L. Bottou, and K. Q. Weinberger, editors, *Advances in Neural Information Processing Systems*. Curran Associates, Inc., 2012. https://tinyurl.com/yyzan457.

[34] A. Kurakin, I. Goodfellow, and S. Bengio. Adversarial examples in the physical world, 2017. https://arxiv.org/abs/1607.02533.

[35] C.-C. Lee. Investigation of synaptic plasticity as memory formation mechanism and pathological amyloid fibrillation caused by β-amyloids aggregation. appendix d. https://tinyurl.com/y4g5qujx, 2008.

[36] G. R. Machado, E. Silva, and R. R. Goldschmidt. Adversarial machine learning in image classification: A survey towards the defender's perspective, 2020. https://arxiv.org/abs/2009.03728.

[37] J. Markoff. Seeking a better way to find web images. *New York Times*, 2012. https://tinyurl.com/y4gujrbp.

[38] S. P. Marvin Minsky. *Perceptrons: An Introduction to Computational Geometry*. MIT Press, 1969.

[39] T. Mitchell. *Machine Learning*. McGraw-Hill, 1997.

[40] S.-M. Moosavi-Dezfooli, A. Fawzi, and P. Frossard. Deepfool: a simple and accurate method to fool deep neural networks, 2016. https://arxiv.org/abs/1511.04599.

[41] A. Natekin and A. Knoll. Gradient boosting machines, a tutorial. *Frontiers in neurorobotics*, 7:21, 12 2013. https://tinyurl.com/y4c4ok5v.

[42] V. K. P. Tan, M. Steinbach. *Introduction to Data Mining*. Addison Wesley, 2005.

[43] O. Russakovsky, J. Deng, H. Su, J. Krause, S. Satheesh, S. Ma, Z. Huang, A. Karpathy, A. Khosla, M. Bernstein, A. C. Berg, and L. Fei-Fei. Imagenet large scale visual recognition challenge, 2015. https://arxiv.org/abs/1409.0575.

[44] S. S. Gradient descent: All you need to know. *medium.com*, 2018. https://tinyurl.com/yxgpxj9v, accessed: 2021-01-25.

[45] S. Sabour, N. Frosst, and G. E. Hinton. Dynamic routing between capsules, 2017. https://arxiv.org/abs/1710.09829.

[46] M. Sandler, A. Howard, M. Zhu, A. Zhmoginov, and L.-C. Chen. Mobilenetv2: Inverted residuals and linear bottlenecks, 2019. https://arxiv.org/abs/1801.04381.

[47] S. Santurkar, D. Tsipras, A. Ilyas, and A. Mądry. How does batch normalization help optimization?, 2018. https://dl.acm.org/doi/10.5555/3327144.3327174.

[48] L. Schmidt, S. Santurkar, D. Tsipras, K. Talwar, and A. Mądry. Adversarially robust generalization requires more data, 2018. https://arxiv.org/abs/1804.11285.

[49] S. Sharma. Activation functions in neural networks, 2017. https://tinyurl.com/y9dx5xww, accessed: 2021-01-22.

[50] K. Simonyan and A. Zisserman. Very deep convolutional networks for large-scale image recognition, 2015. https://arxiv.org/abs/1409.1556.

[51] P. S.J.Russel. *Artificial Intelligence: A Modern Approach.* Prentice Hall, 1995.

[52] J. T. Springenberg, A. Dosovitskiy, T. Brox, and M. Riedmiller. Striving for simplicity: The all convolutional net, 2015. https://arxiv.org/abs/1412.6806.

[53] C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, and A. Rabinovich. Going deeper with convolutions, 2014. https://arxiv.org/abs/1409.4842.

[54] C. Szegedy, V. Vanhoucke, S. Ioffe, J. Shlens, and Z. Wojna. Rethinking the inception architecture for computer vision, 2015. https://arxiv.org/abs/1512.00567.

[55] C. Szegedy, W. Zaremba, I. Sutskever, J. Bruna, D. Erhan, I. Goodfellow, and R. Fergus. Intriguing properties of neural networks, 2014. https://arxiv.org/abs/1312.6199.

[56] T. Tanay and L. Griffin. A boundary tilting persepective on the phenomenon of adversarial examples, 2016. https://arxiv.org/abs/1608.07690.

[57] M. Tripathi. Underfitting and overfitting in machine learning. 2020. https://tinyurl.com/yxz6a9gl, accessed: 2020-01-23.

[58] C. Wold. What is translation equivariance, and why do we use convolutions to get it? https://tinyurl.com/y5y8orq9, accessed: 2021-01-26, 2020.

[59] X. Yuan, P. He, Q. Zhu, and X. Li. Adversarial examples: Attacks and defenses for deep learning, 2018. https://arxiv.org/abs/1712.07107.

[60] Y. Zheng, C. Yang, and A. Merkulov. Breast cancer screening using convolutional neural network and follow-up digital mammography. 2018. https://tinyurl.com/y5qr8ry6.

Acknowledgements

I want to thank professor Valentina Poggioni, for allowing me to work on this interesting project and help me deepen my knowledge about machine learning. I also thank the PhD. students Alina Elena Baia and Gabriele di Bari for their invaluable advice, both in the development of the software and the writing of this thesis. I also want to thank the student Laura Vagnetti for helping me write the code for the Instagram filters during our internship, and for being a good friend during the last few years, along with my classmate Federico Galli. Lastly, i want to thank my family for having supported my every decision thus far. I would not have been able to complete my studies in this university without them.