University of Novi Sad
Faculty of Sciences
Department of Mathematics
and Informatics

Application of Genetic Algorithms to Optimization of Convolutional Neural Network Architecture

Master Thesis

Milan Ignjić

Supervisor: dr Oskar Marko

Novi Sad, 2024

Abstract

This thesis explores neural network architecture design, a fundamental aspect of deep learning. A common practice in this domain involves the adaptation of existing architectures to address specific problems. However, this approach, while convenient, comes with an inherent risk of architectural bias, limiting the exploration of alternative, potentially superior designs.

The central objective of this thesis is to propose an innovative methodology for the automated generation of neural network architectures. This approach leverages evolutionary algorithms, inspired by natural selection, to create novel neural network structures. By iteratively evolving architectures using a predefined set of objectives, this method systematically introduces and explores variations, ultimately unveiling diverse, unexplored network structures.

The main advantage of this approach is in its ability to explore unexplored areas of architecture design. While human intuition may gravitate toward familiar patterns, the evolutionary algorithm explores unconventional configurations, possibly uncovering innovative solutions. This work encourages the exploration of novel architectural possibilities, reducing the risk of missing optimal solutions.

Acknowledgments

I extend my heartfelt gratitude to my thesis supervisor, Dr. Oskar Marko, for his guidance and valuable assistance with evolutionary algorithms. Additionally, I want to express my appreciation to Dr. Sanja Brdar for her unwavering motivation and support throughout the research on this topic.

My sincere thanks go to my family for their enduring support and constant encouragement throughout my years of study.

My deepest gratitude is reserved for my fiancée, without whom I would not have had the strength to see this research through to its conclusion. Her care and support were indispensable throughout this journey.

Finally, I offer special thanks to my late father, whose inspiration and courage continue to drive me to pursue my dreams.

Contents

List of Figures

List of Abbreviations

ANN	Artificial Neural Network
CNN	Convolutional Neural Network
ReLU	Rectified Linear Units
MLP	Multilayer Perceptron
RGB	Red, Green, Blue
SBX	Simulated Binary Crossover
EA	Evolutionary Algorithms
GA	Genetic Algorithms
NSGA	Non-dominated Sorting Genetic Algorithm
GPU	Graphical Processing Unit

1. Introduction

When constructing neural network models by manually specifying network architectures, a prevalent method involves the initial exploration of architectures implemented to tackle similar problems [1]. This preliminary step involves exploring existing solutions that have proven effective in addressing similar challenges, and subsequently tailoring these architectures to address the unique prerequisites specific to the particular use case [2] [3]. This approach aims to speed up the network's convergence towards a viable solution by leveraging established patterns that have already demonstrated efficacy in related domains.

However, there is a shortcoming in this approach. While adapting and fine-tuning existing architectures can certainly yield promising outcomes, there remains a notable drawback in the form of architectural bias. By predominantly focusing on pre-existing solutions, researchers might miss an extensive array of alternative architectures that could potentially outperform the modified designs [4].

In essence, the approach of borrowing and modifying architectures, while convenient, carries the potential to miss out on the optimal solutions. The domains of neural network design and architecture hold a multitude of uncharted possibilities that demand thorough exploration. By going beyond the familiar constructs, researchers can increase their chances of crafting novel and superior architectures that align more accurately with the problem they are trying to solve [5].

The central objective of this thesis revolves around the proposition of an innovative methodology for the automated generation of neural network architectures. The proposed solution involves the integration of evolutionary algorithms, a powerful computational paradigm inspired by the principles of natural selection, to facilitate the creation of novel neural network structures. This method is based on utilizing neural network architectures that have proven to be top-performing solutions, guided by a predefined set of objectives

In essence, this approach uses evolutionary algorithms to iteratively evolve and refine neural network architectures. The initial step involves identifying neural network configurations that have demonstrated exceptional performance according to the specified objectives. These high-performing architectures serve as the seed population for the evolutionary process. Through a series of generational iterations, the algorithm systematically introduces and explores variations within the architecture space, resulting in the emergence of diverse and unexplored neural network structures.

This approach stands out because it can explore new areas in architecture design. Human intuition often sticks to what is known, but the evolutionary algorithm goes beyond, investigating unusual designs that people might not consider. This ability to venture into unconventional architectural territory can reveal innovative solutions that might otherwise stay hidden [5].

1.1. Artificial Neural Networks

The history of Artificial Neural Networks (ANN) [1] [6] [7] [8] traces back to the mid-20th century, with roots firmly established in the field of neuroscience and early attempts to replicate the functioning of the human brain. The development of ANNs has been marked by several significant milestones.

In 1943 Warren McCulloch and Walter Pitts made a significant contribution to the field of neural networks by introducing the McCulloch-Pitts neuron model [6]. This model provided a fundamental understanding of how artificial neurons could mimic the behavior of biological neurons.

In their groundbreaking work, McCulloch and Pitts proposed a computational abstraction of a neuron, which they termed the McCulloch-Pitts neuron. This model aimed to capture the basic functionality of a biological neuron by simplifying its behavior into mathematical operations.

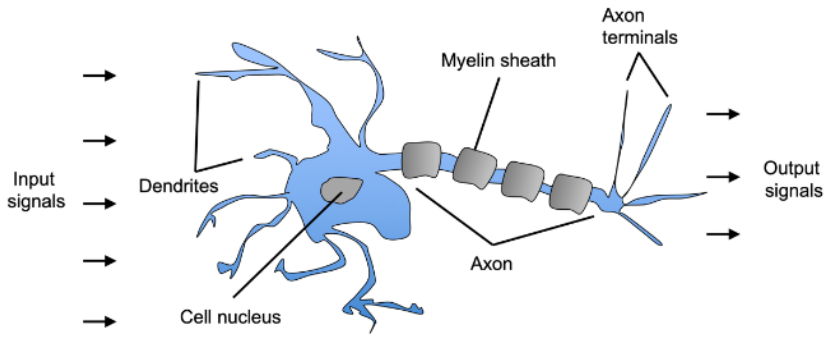

Figure 1.1: Biological neuron [6]

At its core, the McCulloch-Pitts neuron operates on the concept of binary thresholds [9]. It takes input signals, each with an associated weight, and processes them. If the weighted sum of inputs surpasses a certain threshold value, the neuron "fires," emitting an output signal. Otherwise, it remains inactive. This binary firing mechanism mirrored the behavior of biological neurons, which either transmit an electrical signal (a "spike") when a certain level of excitation is reached or remain dormant.

Expanding further upon the foundation laid by the McCulloch-Pitts neuron model, in 1957 psychologist Frank Rosenblatt played a pivotal role in advancing the field of neural networks with his development of the perceptron. The perceptron represents a significant step forward by introducing a single-layer neural network capable of learning basic patterns [9].

Rosenblatt's perceptron incorporates the principles of the McCulloch-Pitts neuron while also adding a crucial element: the ability to learn and adapt. Unlike the fixed-weight McCulloch-Pitts neuron, the perceptron's algorithm allows it to adjust its internal weights in response to input data. This innovation marks one of the earliest instances of incorporating machine learning into neural networks [6].

The perceptron's learning process is guided by the concept of supervised learning [10]. During training, the perceptron is presented with input-output pairs, and it iteratively adjusts its weights to minimize the difference between its predicted outputs and

the desired outputs. This weight adjustment process is driven by a mathematical algorithm, specifically the perceptron learning rule, which aims to optimize the network's ability to classify and distinguish patterns.

While the perceptron demonstrated promising capabilities in handling simple pattern recognition tasks, it has limitations. It can only effectively learn linearly separable patterns [6], which limits its applicability to more complex problems that require nonlinear decision boundaries.

Following the initial excitement and promising developments in the field of artificial intelligence, the AI winter [11] emerged as a significant setback during the late 20th century. The AI winter refers to a period of diminished funding, dwindling interest, and reduced progress in artificial intelligence research and applications. This downturn was influenced by a combination of factors, including overhyped expectations, unmet promises, and technological limitations.

One of the contributing factors to the AI winter was the "hype cycle" that surrounded artificial intelligence in its early stages. As expectations soared following breakthroughs like the perceptron and other early AI achievements, there was a growing sense that AI systems could perform tasks at a level comparable to human intelligence across a wide range of domains. However, as researchers encountered challenges related to the complexity of cognition, language understanding, and reasoning, it became evident that the initial optimism had been inflated.

Moreover, during the 1970s and 1980s, the limitations of existing computing hardware became a significant hurdle. The computational power required for advanced AI tasks far exceeded the capabilities of the available hardware at the time. As a result, the practical realization of the ambitious goals set for AI systems remained elusive, leading to disillusionment within the research community and funding bodies.

These factors converged to create a general skepticism about the feasibility of AI applications, leading to a decline in funding for AI research projects and a decrease in public interest. Despite the challenges and setbacks, however, the AI winter also served as a valuable lesson for the field. It highlighted the need for more realistic expectations, a focus on foundational research, and the development of practical applications that could gradually demonstrate the value of artificial intelligence in specific domains. This recalibration ultimately paved the way for the renaissance of AI that followed, driven by advances in computing technology, the rise of big data, and the emergence of more sophisticated machine learning algorithms.

The introduction of the backpropagation algorithm [12] in 1986 by David Rumelhart, Geoffrey Hinton, and Ronald Williams breathed new life into the study of neural networks. This innovative algorithm revitalized interest in the field, sparking a renewed enthusiasm for exploring the potential of neural networks. Backpropagation played a pivotal role in enabling the training of multi-layer neural networks, allowing them to learn complex patterns and features from data.

The significance of backpropagation was evident in its ability to address a long-standing challenge in neural network research—training networks with multiple layers. Prior to this breakthrough, the training of deep neural networks was fraught with difficulties, making it challenging for these networks to effectively learn and generalize

from complex data. Backpropagation changed this landscape by providing a systematic way to adjust the network's internal parameters based on the discrepancies between predicted and actual outputs. This mechanism allowed the network to iteratively fine-tune its weights and biases, gradually improving its performance on the given task.

In the 21st century, neural networks experienced a renewed interest, owing to various significant factors that breathed new life into the field. Key among these were advancements in computational power, the accessibility of extensive datasets, and the emergence of innovative architectural designs. This resurgence gave rise to the paradigm of Deep Learning [7], marked by the utilization of deep neural networks, which demonstrated remarkable achievements in tasks like image recognition [1], natural language processing [13], and strategic gaming (AlphaGo [14]).

1.2. Convolutional Neural Network

A Convolutional Neural Network (CNN) [1] [6] [7] [15] [16] [17] is a specialized type of artificial neural network designed for processing and analyzing visual data, particularly images and videos. CNNs are inspired by the visual processing mechanisms of the human brain and are highly effective in tasks like image recognition, object detection, image segmentation, and more.

Usually, CNNs are structured with a series of convolutional layers, followed by one or more fully connected layers at the end. These fully connected layers essentially form a Multilayer Perceptron.

A standard convolutional layer within a convolutional network consists of three stages. In the first stage, the layer conducts multiple convolutions in parallel, resulting in a collection of linear activations. In the second stage, each of these linear activations is passed through a nonlinear activation function, often referred to as the "detector stage." Finally, in the third stage, a pooling function is employed to further transform the layer's output.

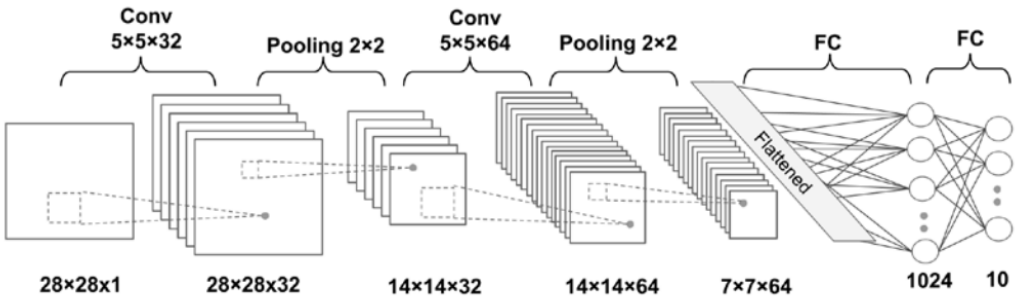

Figure 1.2: Example of Convolutional Neural Network [6]

Early layers are responsible for extracting low-level features directly from raw data, while the later layers utilize these extracted features for making predictions. This characteristic distinguishes CNNs from traditional machine learning models, as they can automatically learn and extract relevant features from the input data, reducing the dependence on manually engineered features provided by domain experts.

1.3. Neuroevolution

The concept of neuroevolution [18] [19] in artificial intelligence draws inspiration from the evolution of biological nervous systems. It applies abstractions of natural evolution, such as evolutionary algorithms, to construct artificial neural networks mimicking biological neural networks. The ultimate and ambitious goal is to evolve complex artificial neural networks capable of intelligent behavior. Therefore, neuroevolution serves both as a means to explore how intelligence evolved in nature and as a practical method for engineering artificial neural networks to perform specific tasks.

Much like natural selection in the domain of biology, which is influenced by feedback from reproductive success, neuroevolution relies on a measure of overall performance as its guiding principle.

What sets neuroevolution apart is its adaptability to various network architectures and neural models. To implement neuroevolution, it is necessary to evaluate network performance over time and to be able to modify network behavior through evolutionary processes. While many neural learning methods [20] primarily focus on adjusting neural connection strengths (connection weights), neuroevolution goes a step further by optimizing other parameters, such as network topology (e.g., adding neurons or connections) and the computational functions performed by individual neurons.

In the typical neuroevolution process, a population of genetic encodings representing neural networks is evolved with the objective of finding a network capable of solving a specific task. Most neuroevolution methods adhere to the conventional generate-and-test loop [21] commonly seen in evolutionary algorithms.

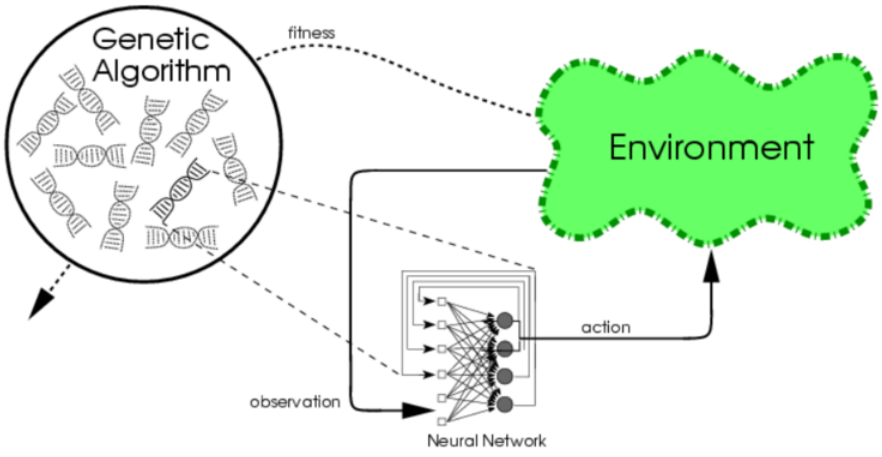

Figure 1.3: Neural network evolution [22]

1.4. Digit Recognition

Digit recognition [1] [15], a fundamental challenge in computer vision, involves deciphering and categorizing handwritten or printed digits into their corresponding numerical values. This task has wide-ranging applications, from check processing in banking to

digitizing postal codes. The aim is to develop a system that can automatically distinguish between different digits, regardless of variations in handwriting styles, sizes, and orientations.

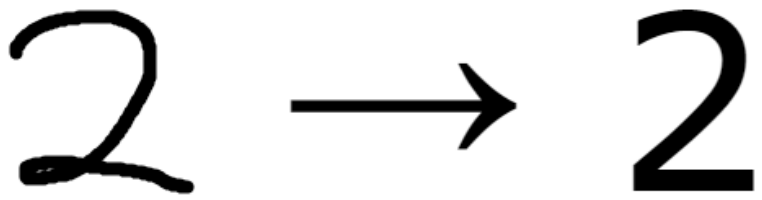

Figure 1.4: Handwritten digit recognition [23]

To tackle this problem, researchers and engineers employ various techniques to extract distinctive features from the input images. These features capture essential characteristics like lines, curves, and intersections that are common across different instances of each digit. The process involves preprocessing the images to enhance contrast and remove noise, followed by feature extraction to pull out potentially meaningful information.

After feature extraction, a classification algorithm comes into play. This algorithm is designed to categorize the input images based on the extracted features. It learns patterns from a labeled dataset containing examples of each digit. During the training phase, the algorithm adjusts its internal parameters to minimize the difference between predicted classifications and the true labels of the training data.

Once the algorithm is trained, it can be evaluated on new, unseen images. The goal is to generalize its learning and accurately classify digits it has never encountered before. Evaluating the model's performance involves metrics such as accuracy, precision, recall, and F1-score, which quantify its ability to correctly classify different digits and handle potential errors [6] [9].

In the history of number recognition, significant strides were made in the late 20th century to develop efficient methods for automatic digit recognition. In the 2000s, the Viola-Jones algorithm [24] emerged as a landmark technique for object detection and facial recognition. While primarily designed for detecting faces, it laid the groundwork for subsequent advancements in recognizing digits and patterns. The Viola-Jones algorithm exploited Haar-like features [25] and utilized a cascading approach to quickly discard non-relevant regions of an image, making it computationally efficient.

During the 1980s and 1990s, researchers began exploring the use of neural networks for pattern recognition tasks like digit recognition. However, these early attempts were hindered by several drawbacks. Neural networks of that era struggled with vanishing gradients [26], limited computational resources, and insufficient training data. The models were often shallow and lacked the depth needed to capture intricate features. As a result, the performance of these networks was not on par with expectations.

The turning point came with the mainstream adoption of the internet. The internet brought about an explosion of digitized data, with numerous handwritten

digit datasets becoming widely accessible. This influx of data proved to be a benefit for training machine learning models, as it allowed for more comprehensive and diverse training sets. Furthermore, the increased availability of computing power, coupled with advancements in graphics processing units (GPUs), revolutionized the training of neural networks. GPUs offered parallel processing capabilities that significantly accelerated training times, enabling the training of deeper and more complex models.

The combination of abundant data and improved computational infrastructure set the stage for a neural network resurgence. Researchers revisited the concept of deep neural networks, giving rise to deep learning architectures. CNNs brought about a breakthrough in digit recognition and computer vision tasks in general. CNNs demonstrated exceptional performance in recognizing digits due to their ability to automatically learn hierarchical features from raw pixel values.

AlexNet [27] marked a pivotal moment in the history of neural networks, specifically in the field of computer vision. Developed by Alex Krizhevsky, Ilya Sutskever, and Geoffrey Hinton, AlexNet was a deep convolutional neural network that achieved groundbreaking results in the ImageNet Large Scale Visual Recognition Challenge (ILSVRC) in 2012.

The ImageNet [28] competition, part of the ILSVRC, focused on classifying images into a vast number of categories, with a massive dataset containing millions of labeled images. AlexNet was a game-changer in this competition due to its remarkable performance. It consisted of multiple convolutional and fully connected layers, introducing key architectural innovations that laid the foundation for modern deep neural networks.

What set AlexNet apart were several factors:

1. Depth: AlexNet was considerably deeper than previous neural networks, allowing it to learn complex features and hierarchical representations of the input images. This depth was made possible by the combination of convolutional and pooling layers.

2. Convolutional Layers: The architecture utilized convolutional layers that automatically learned features like edges, textures, and basic shapes directly from the raw pixel data. This reduced the need for manual feature engineering, which was a common practice before deep learning.

3. ReLU Activation [29]: AlexNet employed ReLU as the activation function. ReLU improved training by addressing the vanishing gradient problem, enabling faster and more stable convergence during training.

4. Data Augmentation [30]: The team employed aggressive data augmentation techniques during training, such as cropping, flipping, and adjusting brightness. This helped the network generalize better to different variations of the same image.

5. Dropout [31]: AlexNet incorporated dropout, a regularization technique that randomly deactivated some neurons during each training iteration. This prevented overfitting and improved generalization performance.

AlexNet's innovative architecture and techniques allowed it to achieve a top-5 error rate of around 15.3% on the challenging ImageNet dataset, significantly outperforming the competition. This demonstrated the power of deep neural networks for

large-scale image recognition tasks and signaled a resurgence of interest in neural network research, inspiring the development of even more sophisticated architectures.

2. Background theory

2.1. Artificial Neural Networks

In this segment, we present the concept of the ANN. We then delve into the explanation of the perceptron, which is the most elementary form of a neural network. We subsequently explore the convergence of the perceptron algorithm and the multilayer perceptron.

2.1.1 Perceptron

A perceptron is one of the fundamental building blocks of artificial neural networks and machine learning. It is a type of artificial neuron that models a simplified version of a biological neuron. The perceptron takes multiple input values, multiplies each input by a corresponding weight, sums up the weighted inputs, and then passes the result through an activation function to produce an output.

Mathematically, the output of a perceptron can be represented as:

$$y(x) = f(\mathbf{w}^T x + w_0) \tag{2.1}$$

where $x \in \mathbb{R}^n$, $\mathbf{w} \in \mathbb{R}^n$, $w_0 \in \mathbb{R}$, and f is a step function.

$$f(a) = \begin{cases} 1, & a \leq 0 \\ -1, & a > 0 \end{cases} \quad a \in \mathbb{R} \tag{2.2}$$

It was designed for binary classification tasks [9], where it could learn to separate data points into two classes based on the learned weights and biases. These classes are denoted as $\mathcal{C}_1$ and $\mathcal{C}_2$.

Define $g(x) = \mathbf{w}^T x + w_0$. The function g can be rewritten as:

$$g(x) = \mathbf{w}^T x + w_0 = \begin{bmatrix} \mathbf{w}^T & w_0 \end{bmatrix} \begin{bmatrix} x \\ 1 \end{bmatrix} = \mathbf{w}'^T x' \tag{2.3}$$

In further notation we can simply use $g(x) = \mathbf{w}^T x$. The aim is to find a vector w such that:

$$\mathbf{w}^T x = 0 \quad \text{and} \quad \begin{cases} \mathbf{w}^T x > 0, & \forall x \in \mathcal{C}_1 \\ \mathbf{w}^T x < 0, & \forall x \in \mathcal{C}_2 \end{cases} \tag{2.4}$$

To simplify the process, we change the sign of every sample in class $\mathcal{C}_2$:

$$x \to -x, \quad \forall x \in \mathcal{C}_2 \tag{2.5}$$

after which optimization problems becomes:

$$\mathbf{w}^T x > 0, \forall x \tag{2.6}$$

To find a solution, it is mandatory to establish the objective function $J(\mathbf{w})$ [9]. A suitable choice for this function is what is commonly referred to as the perceptron criterion function:

$$J_p(\mathbf{w}) = -\sum \mathbf{w}^T x \tag{2.7}$$

where $\mathcal{M}$ is a set of all misclassified samples. Given that $\mathbf{w}^T x < 0$ holds true for all $x \in \mathcal{M}$, it follows that $J_p(\mathbf{w})$ is always nonnegative. The minimum value of this function is 0, and it is attained when there are no incorrectly classified samples. Due to the continuous and piecewise linear nature of the perceptron criterion function, we can utilize the stochastic gradient descent algorithm to search for its minimum.

Gradient is defined as

$$\nabla_{\mathbf{w}} J_p(\mathbf{w}) = -\sum_{x \in \mathcal{M}} x \tag{2.8}$$

therefore:

$$\mathbf{w}_{k+1} = \mathbf{w}_k - \eta_k \nabla_w J_p(\mathbf{w}) = \mathbf{w}_k + \eta_k \sum x \tag{2.9}$$

where η_k is the learning rate parameter.

Perceptron learning algorithm

Perceptron algorithm [6] can be summarized by the following steps:

1. Initialize the weights and bias unit to 0 or small random numbers.
2. For each training example, $\mathbf{x}$:
   (a) Compute the output value, $\hat{y}$.
   (b) Update the weights and bias unit.
3. Go to step 2 until there are no misclassified samples.

The output value is the class label predicted by the step function previously defined.

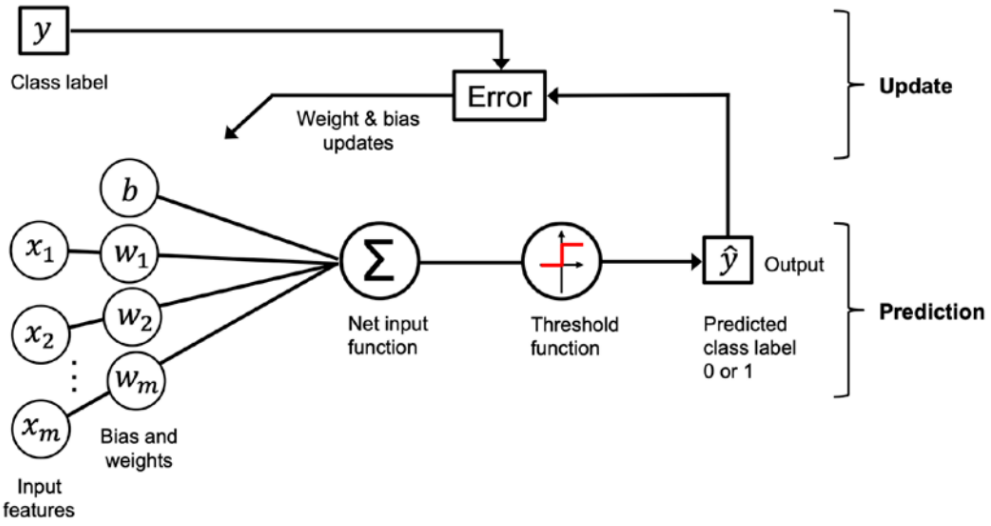

Figure 2.1: Weights and bias of the model are updated based on the error function [6]

Convergence of Perceptron rule

Convergence of the perceptron is only guaranteed if the two classes are linearly separable. However, in cases where two classes are not linearly separable, there is no vector $\mathbf{w}^*$ that can accurately classify every sample, therefore the updates in perceptron algorithms will never end. Figure 2.2 shows a difference between linearly and nonlinearly separable classes.

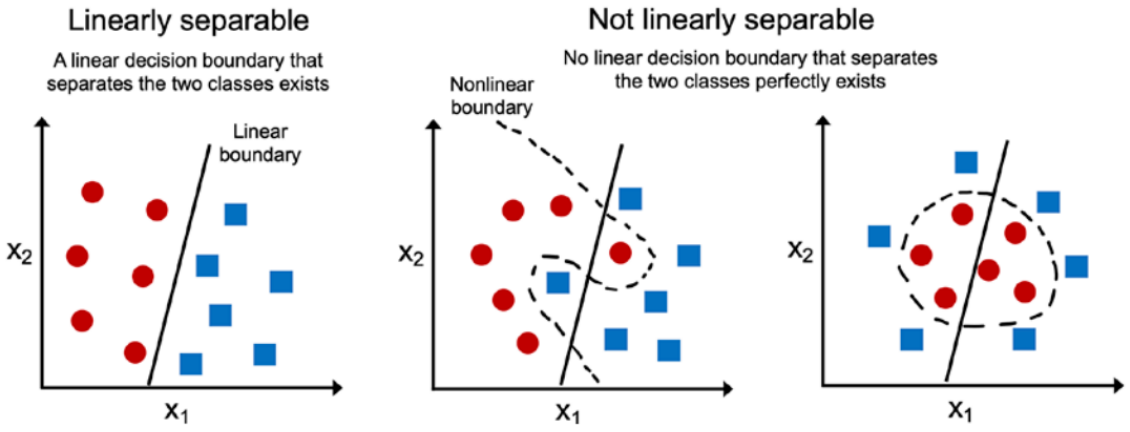

Figure 2.2: Examples of linearly and nonlinearly separable classes [6]

Assume that classes $\mathcal{C}_1$ and $\mathcal{C}_2$ are linearly separable. Let $\mathbf{w}^*$ be the solution of a problem and $\alpha \in \mathbb{R}$. Therefore, based on 2.9 follows:

$$\mathbf{w}_{k+1} - \alpha\mathbf{w}^* = w_k - \alpha\mathbf{w}^* + \eta_k \sum x \tag{2.10}$$

Applying the square of Euclidean norm from both sides gives:

$$\|\mathbf{w}_{k+1} - \alpha\mathbf{w}^*\|^2 = \|\mathbf{w}_k - \alpha\mathbf{w}^*\|^2 + \eta_k^2 \left\| \sum_{x\in\mathcal{M}} x \right\| + 2\eta_k \sum_{x\in\mathcal{M}} (\mathbf{w}_k - \alpha\mathbf{w}^*)^T x \tag{2.11}$$

Since $\sum_{x\in\mathcal{M}} \mathbf{w}^T x < 0$:

$$\|\mathbf{w}_{k+1} - \alpha\mathbf{w}^*\|^2 \leq \|\mathbf{w}_k - \alpha\mathbf{w}^*\|^2 + \eta_k^2 \left\| \sum_{x\in\mathcal{M}} x \right\|^2 - 2\eta_k\alpha \sum_{x\in\mathcal{M}} \mathbf{w}^{*T} x \tag{2.12}$$

We define:

$$\beta^2 = \max_{\tilde{\mathcal{M}}\subseteq\mathcal{C}_1\cup\mathcal{C}_2} \left\| \sum_{x_n\in\tilde{\mathcal{M}}} x \right\|^2 \tag{2.13}$$

$$\gamma = \min_{\tilde{\mathcal{M}}\subseteq\mathcal{C}_1\cup\mathcal{C}_2} \sum_{x_n\in\tilde{\mathcal{M}}} \mathbf{w}^{*T} x \tag{2.14}$$

β^2 represents the maximum value of a vector norm among all possible subsets of the given data and that γ is always negative based on 2.7. Therefore, equation can be rewritten as:

$$\|\mathbf{w}_{k+1} - \alpha\mathbf{w}^*\|^2 \leq \|\mathbf{w}_k - \alpha\mathbf{w}^*\|^2 + \eta_k^2\beta^2 - 2\eta_k\alpha\gamma \tag{2.15}$$

If we set $\alpha = \frac{\beta^2}{2\gamma}$ and subsequently apply the upper inequality for $k, k-1, \ldots, 0$, we can derive the following:

$$\|\mathbf{w}_{k+1} - \alpha\mathbf{w}^*\|^2 \leq \|\mathbf{w}_0 - \alpha\mathbf{w}^*\|^2 + \beta^2(\sum_{i=0}^{k} \eta_i^2 - \sum_{i=0}^{k} \eta_i) \tag{2.16}$$

Selecting the sequence $\{\eta_i\}_{i=0}^k$, $\eta_i \in \mathbb{R}$, such that it meets the following conditions:

$$1. \lim_{k\to\infty} \sum_{i=0}^{k} \eta_i = \infty \tag{2.17}$$

$$2. \lim_{k\to\infty} \sum_{i=0}^{k} \eta_i^2 < \infty \tag{2.18}$$

means that there exists k_0 such that for all $k > k_0$, $k \in \mathbb{N}$, the right side of the inequality is nonpositive. Since the left side of the inequality is nonnegative, it follows:

$$\|\mathbf{w}_{k_0} - \alpha\mathbf{w}^*\| \tag{2.19}$$

In other word:

$$\mathbf{w}_{k_0} = \alpha \mathbf{w}^* \tag{2.20}$$

Thus, we have demonstrated the convergence of the perceptron algorithm when these conditions are met.

2.1.2 Multilayer perceptron

The simple perceptron is effective for binary classification tasks when dealing with two linearly separable classes. However, it faces challenges when confronted with multiple classes, and even by adding more perceptrons to the layer, it struggles with nonlinear cases. This limitation is where the Multilayer Perceptron (MLP) [32] steps in.

The MLP consists of three types of layers: an input layer, hidden layers, and an output layer, as shown in Figure 2.3. Perceptrons within each layer receive inputs from neurons in the preceding layer, multiply these inputs by corresponding weights, and sum them up. The result then undergoes an activation function, introducing non-linearity to capture complex relationships.

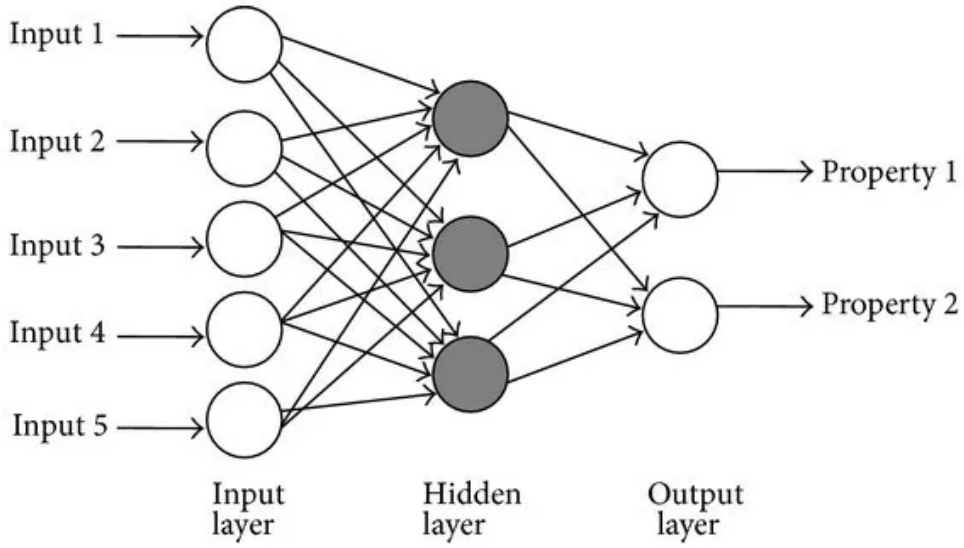

Figure 2.3: Simple Multilayer Perceptron [32]

In the MLP, each activation function is replaced with a differentiable non-linear function to facilitate the calculation of derivatives. Interestingly, if all the activation functions in the hidden layer of a network are linear, then the entire network can be reduced to an equivalent network without hidden units. This is due to the fact that the composition of successive linear transformations remains a linear transformation.

Common functions with this property include sigmoid, tanh, and ReLU [33]. These functions are defined as follows:

Sigmoid:

$$f(x) = \frac{1}{1 + e^{-ax}} \tag{2.21}$$

where a is the slope parameter.

Tanh:

$$f(x) = \frac{\sinh x}{\cosh x} = \frac{e^x - e^{-x}}{e^x + e^{-x}} \tag{2.22}$$

ReLU:

$$f(x) = \max\{0, x\} \tag{2.23}$$

Sigmoid and tanh functions constrain inputs to specific ranges, while ReLU activates for positive inputs.

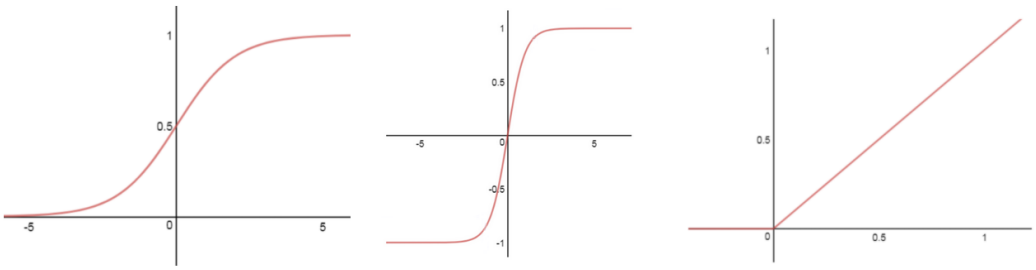

Figure 2.4: Sigmoid, Tanh and ReLU (respectively) [33]

The MLP operates as a feedforward network, which means information flows in only one direction, from one node to another. There are no cycles or loops between nodes, ensuring a clear forward progression of information through the network.

2.2. Convolutional Neural Network

Convolutional neural networks (CNNs) are simply neural networks that use convolution in place of general matrix multiplication in at least one of their layers [6]. The name "convolutional neural network" indicates that the network employs a mathematical operation called convolution.

2.2.1 Convolution

Convolution is an operation on two functions of a real-valued argument [7].

$$s(t) = \int x(a)w(t-a)\,da \tag{2.24}$$

where $t \in \mathbb{R}$. In CNN terminology, the function x is frequently referred to as the input, the function w as the kernel, and the output as the feature map. The convolution operation is typically denoted with an asterisk:

$$s(t) = (x * w)(t) \tag{2.25}$$

In practice, input data is usually not continuous, but discrete. Discrete convolution is defined as:

$$s(t) = (x * w)(t) = \sum_{a=-\infty}^{\infty} x(a)w(t-a) \tag{2.26}$$

In machine learning applications, the input is typically represented as a multidimensional array of data, and the kernel is often a multidimensional array of parameters that are adjusted by the learning algorithm. Since each element of both the input and the kernel must be individually stored, it's generally assumed that these functions are zero everywhere except for the finite set of points for which we store values. In practice, this means that we can compute the infinite summation as a summation over a finite number of array elements.

In digit recognition we use a two-dimensional image I as our input and two-dimensional kernel K:

$$S(i,j) = (K * I)(i,j) = \sum_m \sum_n I(m,n)K(i-m, j-n) \tag{2.27}$$

or

$$S(i,j) = (K * I)(i,j) = \sum_m \sum_n I(i-m, j-n)K(m,n) \tag{2.28}$$

because convolution is a commutative operation. In neural networks, the commutative property of convolution is not a crucial consideration. Instead, cross-correlation is commonly employed. Cross-correlation is essentially the same as convolution, with the key distinction that it doesn't involve flipping the kernel:

$$S(i,j) = (K * I)(i,j) = \sum_m \sum_n I(i+m, j+n)K(m,n) \tag{2.29}$$

This choice simplifies the implementation and interpretation of operations in many neural network frameworks.

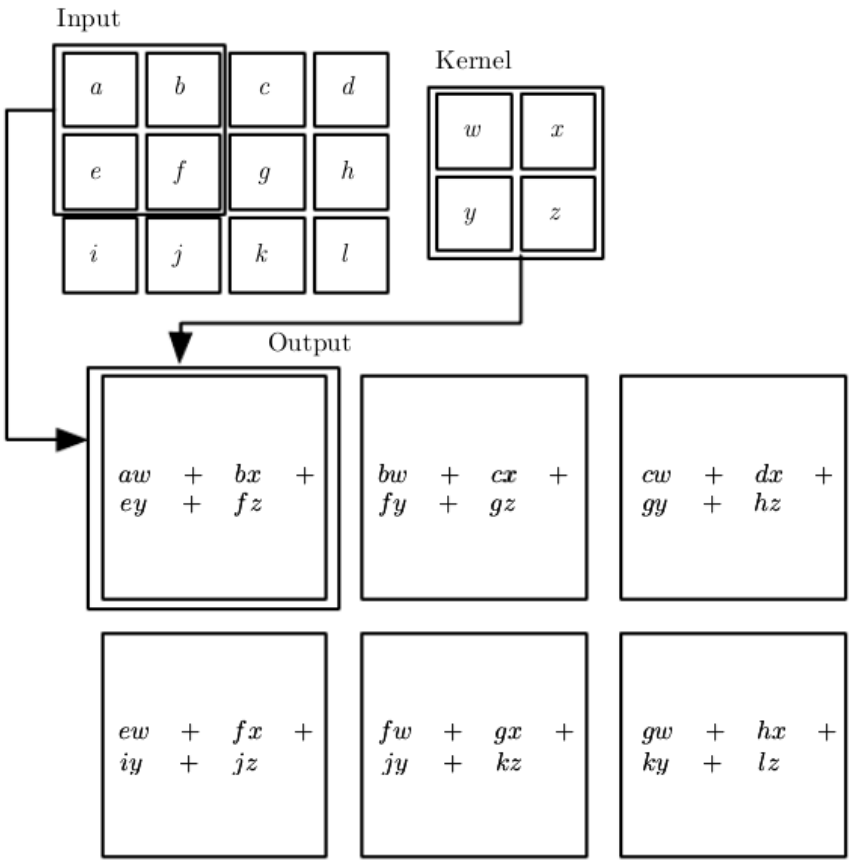

Figure 2.5: An example of 2-D convolution without kernel flipping [7]

Two important properties of convolutional operations in neural networks are sparse interactions and parameter sharing.

In contrast to MLP, where each output feature (neuron) is connected to every input feature, convolutional layers exhibit a different pattern of connectivity. In convolutional layers, each output feature is connected only to a small subset of the input features. This sparse connectivity [34] reduces the number of connections and computations, making it computationally efficient. It also captures local patterns in the data, which is particularly useful for tasks where local relationships are essential, such as image processing.

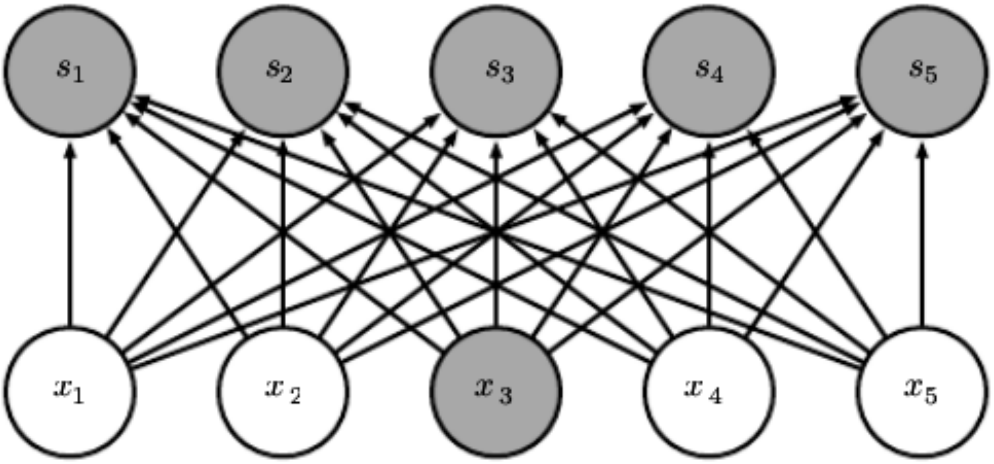

Figure 2.6: Example of interaction between input and output features in an MLP. All outputs are affected by x_3[7]

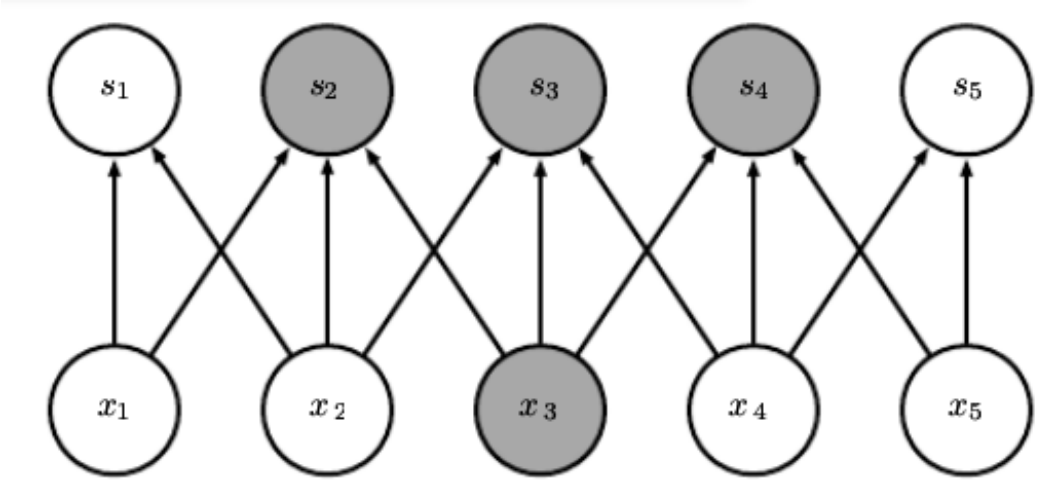

Figure 2.7: Example of interaction between input and output features in a convolution with a kernel of width 3. Only three outputs are affected by x_3[7]

In deep convolutional networks, units in the deeper layers can indirectly interact with a more extensive portion of the input data. This expanded reach of input influence on a unit's behavior is what we refer to as the receptive field [35] of that unit. If the network incorporates architectural features like strided convolution or pooling [36], this receptive field can grow even larger. Therefore, while the direct connections in a convolutional network start with limited local influence, units in deeper layers can establish indirect connections that encompass a significant portion of the input image. This characteristic allows the network to effectively process and recognize complex patterns and relationships in the data.

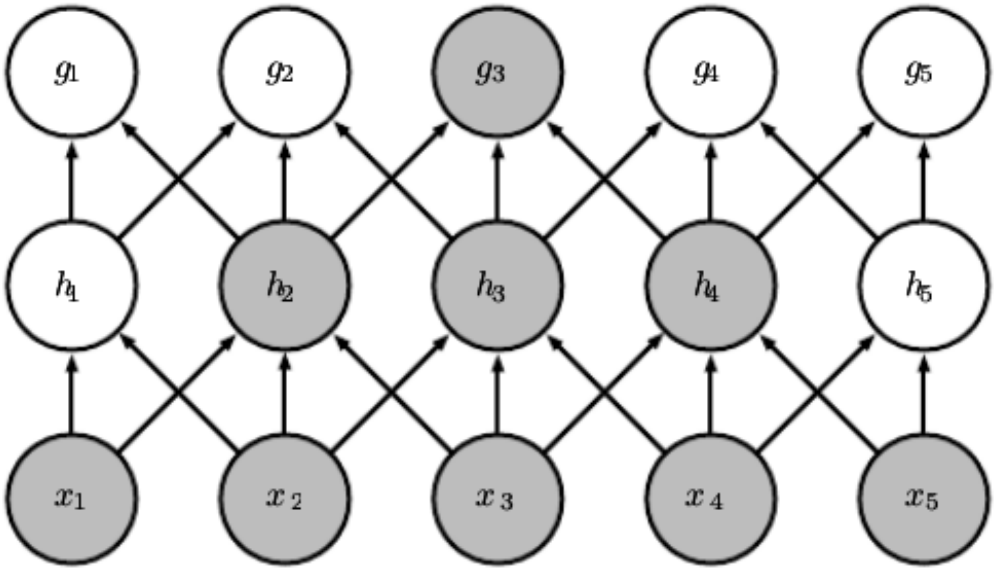

Figure 2.8: Receptive field of the units in the deeper layers [7]

Parameter sharing [34] means that the same set of weights (kernel) is used for multiple locations in the input data. This sharing of weights enables the network to learn and recognize the same features or patterns at different positions in the input. It not only reduces the number of parameters in the model but also encourages the network to learn translation-invariant features. This is a critical property for tasks like image recognition, where the same object or pattern can appear at different positions in an image.

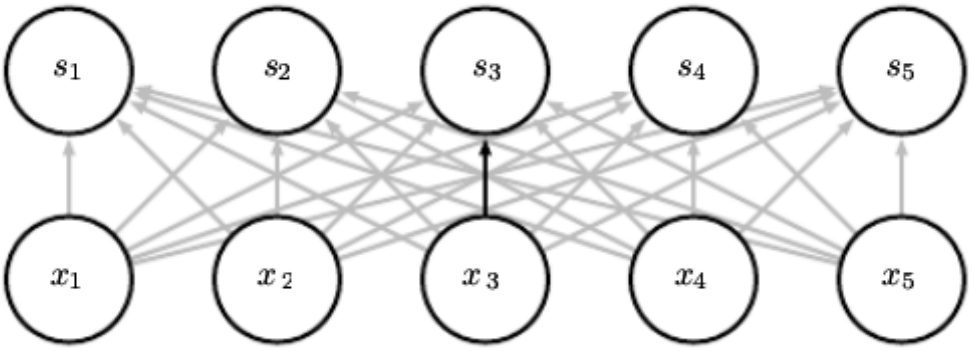

Figure 2.9: Parameter sharing in an MLP. The black arrow indicates the use of the central element of the weight matrix [7]

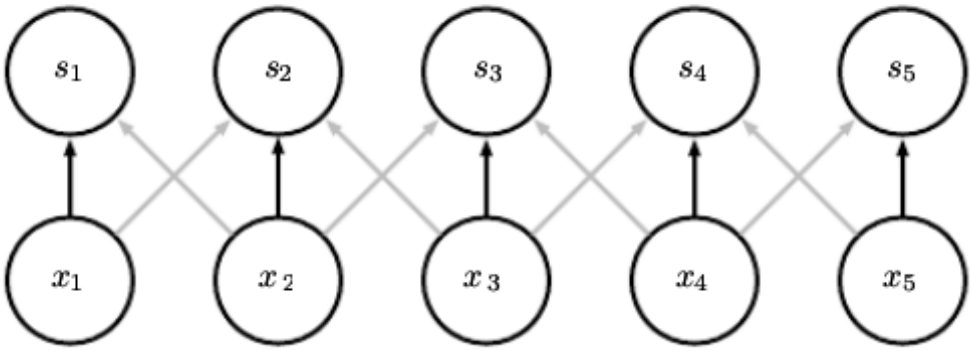

Figure 2.10: Parameter sharing in a convolution. The black arrows indicate uses of the central element of a 3-element kernel [7]

In certain cases, we might choose to skip over some positions of the convolutional kernel to reduce computational expenses, although at the cost of obtaining less detailed features. This can be conceptualized as a form of downsampling applied to the output produced by the full convolution operation. This can be achieved by moving a kernel more than one pixel at a time. We refer to the number of rows and columns traversed per slide as the stride. Additionally, it is feasible to define distinct strides for each direction of motion as needed.

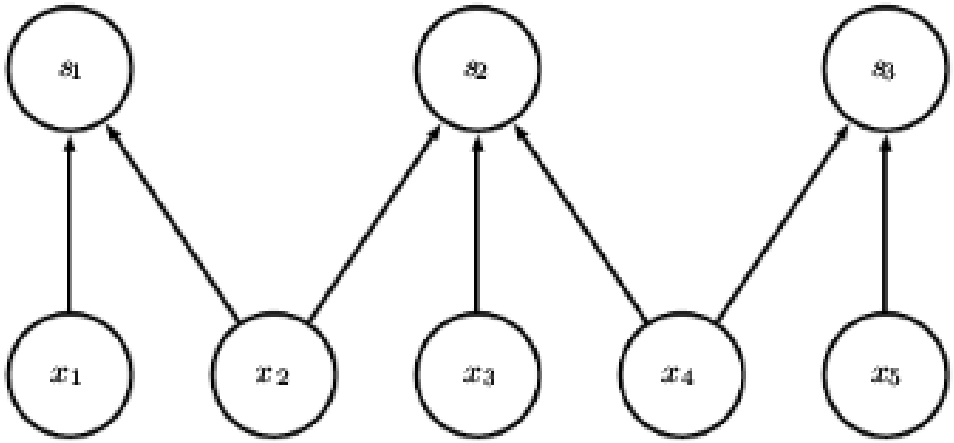

Figure 2.11: Example of a convolution with a stride of two [7]

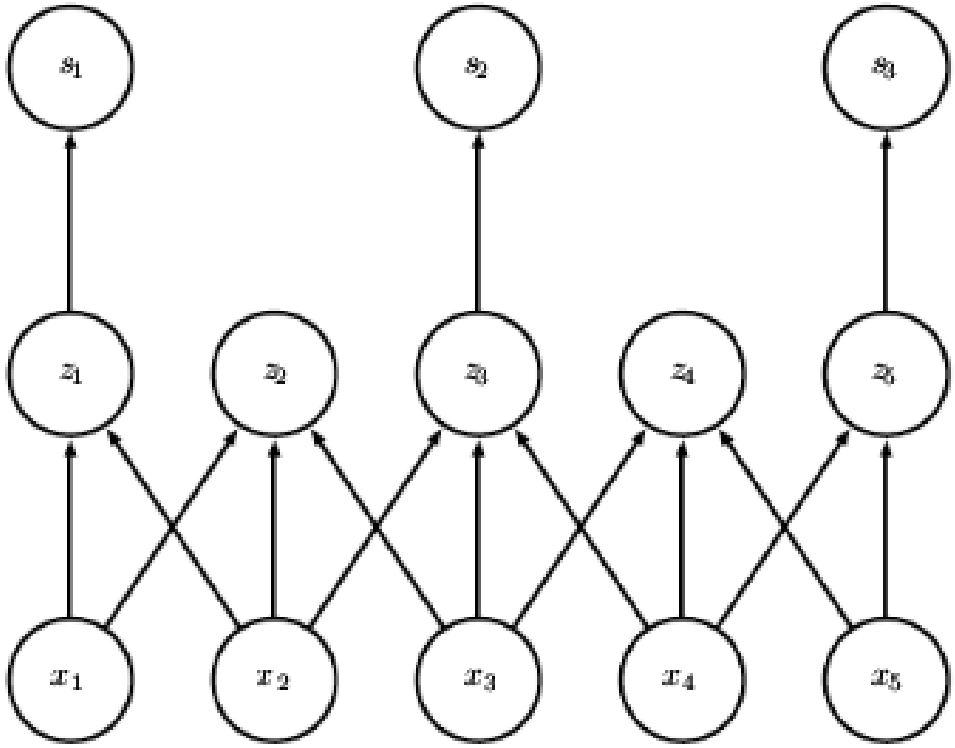

Figure 2.12: Example of a convolution with a unit stride followed by downsampling [7]

With each application of a convolution operation, it's common for the dimension of the representation to decrease. This reduction in dimensionality is a consequence of how convolution operates and the size of the convolutional kernels. That's why one of the critical features in any convolutional network implementation is the ability to implicitly zero-pad the input, effectively expanding its width. Zero-padding [37] grants us the flexibility to independently control the kernel width and the size of the output. Without zero-padding, we would be forced to choose between either rapidly shrinking the spatial extent of the network or resorting to small kernels — both of which would significantly constrain the network's expressive power and capabilities.

Three special cases of the zero-padding setting are worth mentioning:

1. **Valid Convolution:** In this extreme case, no zero-padding is used. Each output pixel is influenced by the same number of pixels in the input, resulting in relatively regular behavior. However, the size of the output shrinks with each layer. If the input image has a width of m and the kernel has a width of k, the output will have a width of $m-k+1$. This shrinkage can be significant, especially when large kernels are used. The limit on the number of convolutional layers that can be added is determined by this shrinkage, and the spatial dimensions can eventually be reduced to $1x1$.
2. **Same Convolution:** In this case, just enough zero-padding is added to maintain the size of the output equal to the size of the input. This allows for as many convolutional layers as the hardware can support, as convolution operations do

not alter the architectural possibilities for the next layer. However, input pixels near the border influence fewer output pixels compared to those near the center.

3. **Full Convolution:** In this other extreme case, enough zeroes are added to ensure that every pixel is visited k times in each direction. This results in an output image with a width of $m + k - 1$. However, the output pixels near the border are influenced by fewer input pixels compared to those near the center. This can make it challenging to learn a single kernel that performs well at all positions in the convolutional feature map.

The optimal amount of zero-padding often lies between ”valid” and ”same” convolution, striking a balance between retaining spatial information and avoiding excessive shrinkage of the feature maps.

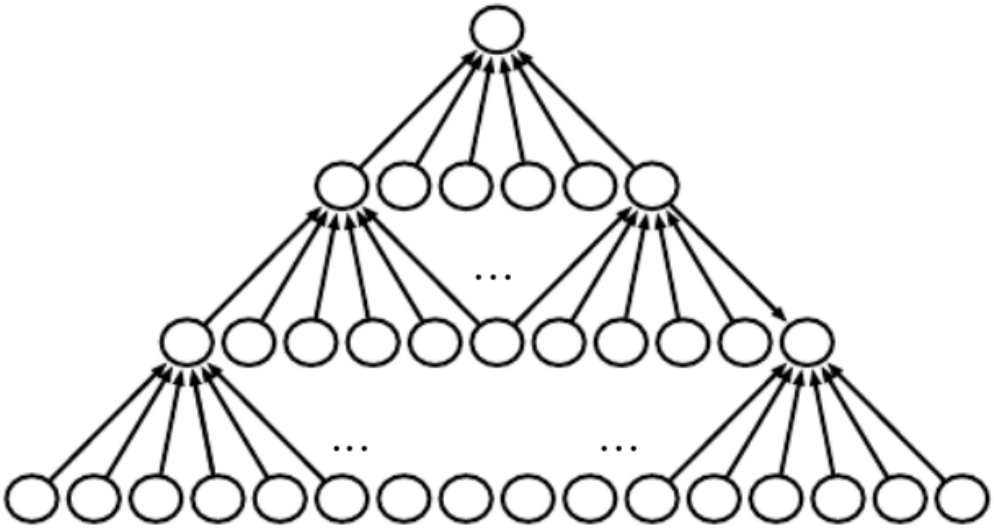

Figure 2.13: The effect of convolution without padding on output size [7]

Figure 2.14: The effect of convolution with same padding on output size [7]

2.2.2 Pooling

A pooling function serves to substitute the output of a neural network at a specific location with a summary statistic derived from nearby outputs. For instance, the widely used max pooling operation reports the maximum output within a defined rectangular neighborhood. Other popular pooling functions encompass computing the average within a rectangular neighborhood, evaluating the $L2$ norm within such a region, or deriving a weighted average based on the distances from the central pixel.

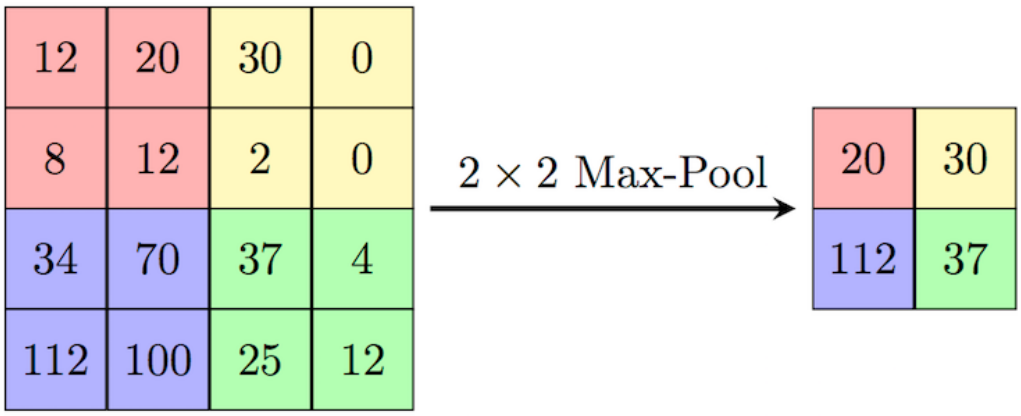

Figure 2.15: Example of a max pooling [36]

Pooling helps to make the representation become approximately invariant to small translations in the input data. Invariance to translation implies that if we shift the input data by a small margin, most of the pooled output values remain unaltered. This property proves highly valuable when the precise location of a feature is less relevant than determining its presence. For instance, when identifying a digit in an image, the exact position of the digit in the image is less significant. However, in scenarios where preserving the spatial location of a feature is more critical, it's advisable to avoid pooling operations.

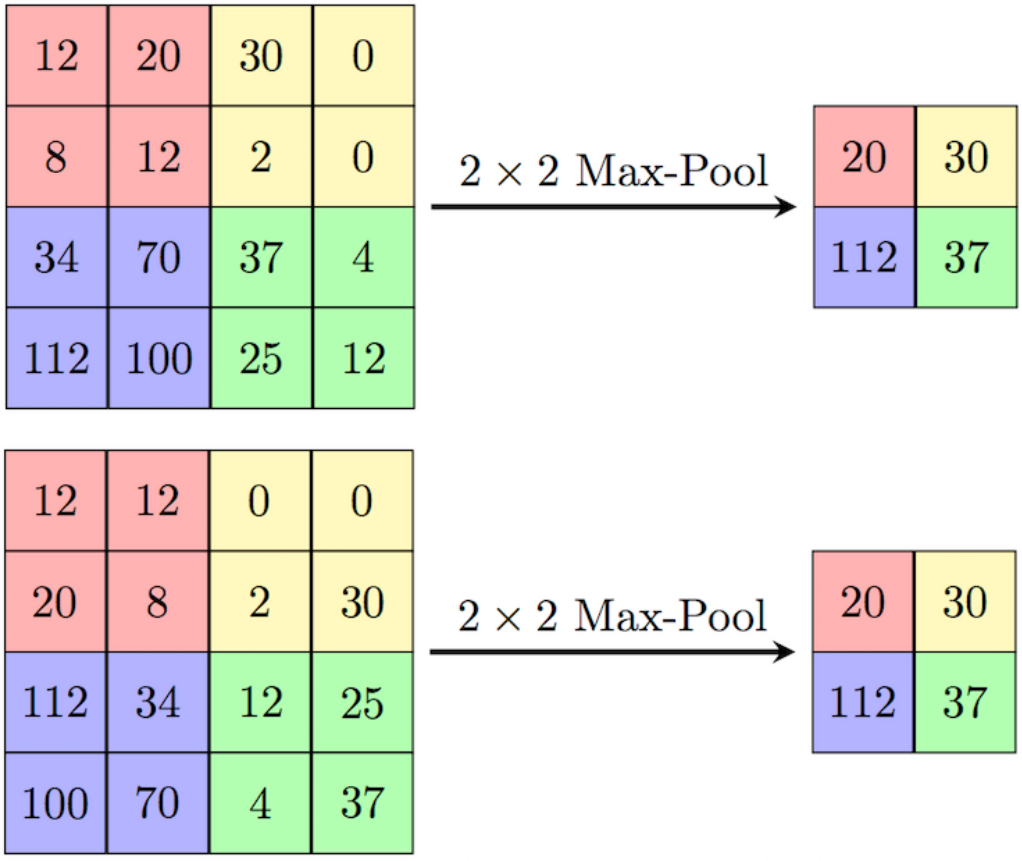

Figure 2.16: Example how pooling preserves invariance to translation [36]

2.2.3 Handling Input Channels in CNNs

In a convolutional layer, the input can consist of one or more $2D$ arrays or matrices, each with dimensions N_1xN_2. These individual N_1xN_2 matrices are referred to as channels. We denote the input as a three-dimensional array, X, with dimensions N_1xN_2xC_{in}, where C_{in} represents the number of input channels.

For example, when images are used as input in the first layer of a CNN, the value of C_{in} depends on the nature of the input image:

- For colored images in the RGB color mode, $C_{in} = 3$, corresponding to the three color channels (red, green, and blue).
- However, for grayscale images, $C_{in} = 1$, because there is only one channel containing grayscale pixel intensity values.

When processing such input, the convolution operation is performed separately for each channel. In other words, each channel c has its own kernel matrix represented as $W[:, , c]$. After the convolution operation is applied to each channel, the results are combined using matrix summation to produce the final output of the convolutional layer. This approach allows the network to capture different features or aspects present in each channel of the input data.

In most cases, a convolutional layer of a CNN has more than one feature map.

If we use multiple feature maps, the kernel tensor becomes four-dimensional: $width \times height \times C_{in} \times C_{out}$. In this context, $width \times height$ represents the kernel size, C_{in} denotes the number of input channels, and C_{out} indicates the number of output feature maps.

Figure 2.18 illustrates a convolutional layer followed by a pooling layer. In this instance, there are three input channels and a four-dimensional kernel tensor. Each kernel matrix is denoted as $m_1 x m_2$, and there are three of them, one for each input channel. Additionally, there are a total of five such kernels, corresponding to five output feature maps. Lastly, a pooling layer is included to perform subsampling on the feature maps.

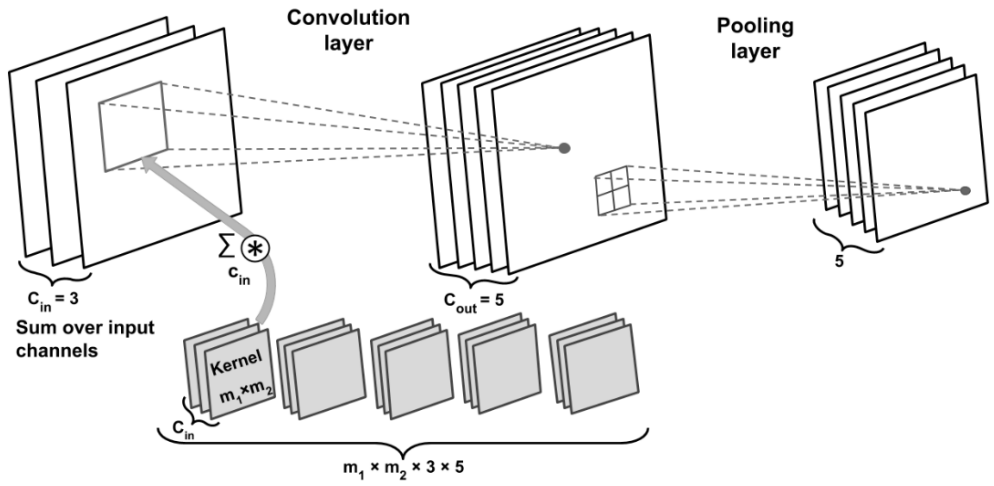

Figure 2.17: Convolutional layer followed by a pooling layer [6]

2.3. Evolutionary algorithms

Evolutionary algorithms (EA) [38] [39] [40] play a crucial role in global optimization. They are grounded in several principles that are inherent to most of them. First and foremost, it's essential to define two concepts - the search space and the objective function space.

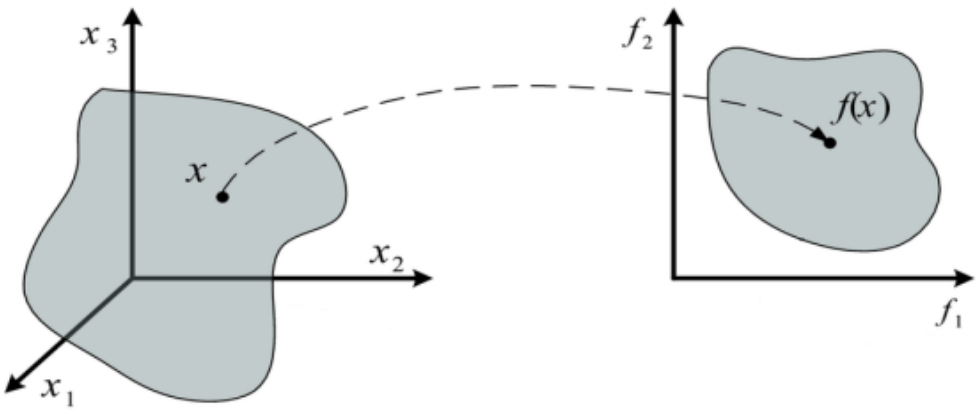

Figure 2.18: A simple representation of the relationship between the search space and the objective function space [22]

The search space encompasses a set of variables, often defined by users, typically representing some physical entities. On the other hand, the objective function space, also known as the criterion function space, consists of variables that correspond to the quality of a particular solution.

The process unfolds by constructing a series of potential solutions, subsequently subjecting them to evaluation. Following Darwinian principles, the fittest solutions survive, and their genetic makeup (or parameters in a non-biological context) are passed onto the next generation. When generating a new population, techniques such as mutation [41], recombination (crossover) [42], and selection [43] are employed to guide the algorithm towards global optimization rather than getting stuck in local optima.

Crossover aims to spread the genetic sequences of well-performing solutions within the population. In binary problems, two parents are chosen, and their sequences are divided at specific positions. These segments are then shuffled to produce offspring. In the generalized case of crossover, known as simulated binary crossover (SBX) [44], "hard" cuts are replaced with a probabilistic approach. Each offspring inherits specific probability density functions, with the maxima of these distributions corresponding to the values of the parents. Consequently, offspring tend to resemble one of the parents, but there's still a chance of divergence.

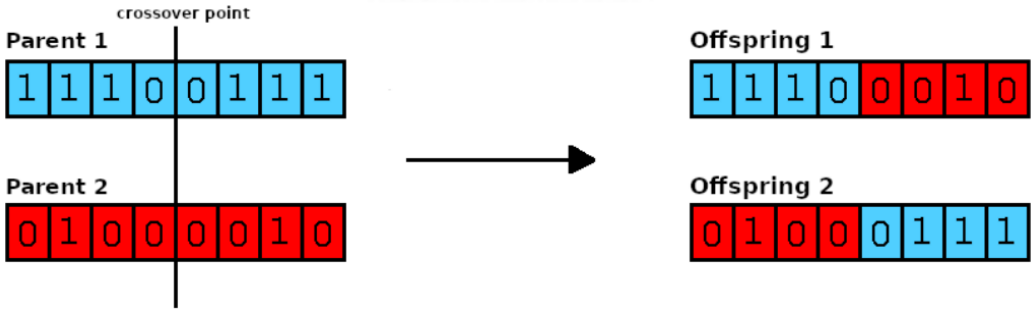

Figure 2.19: Example of binary crossover

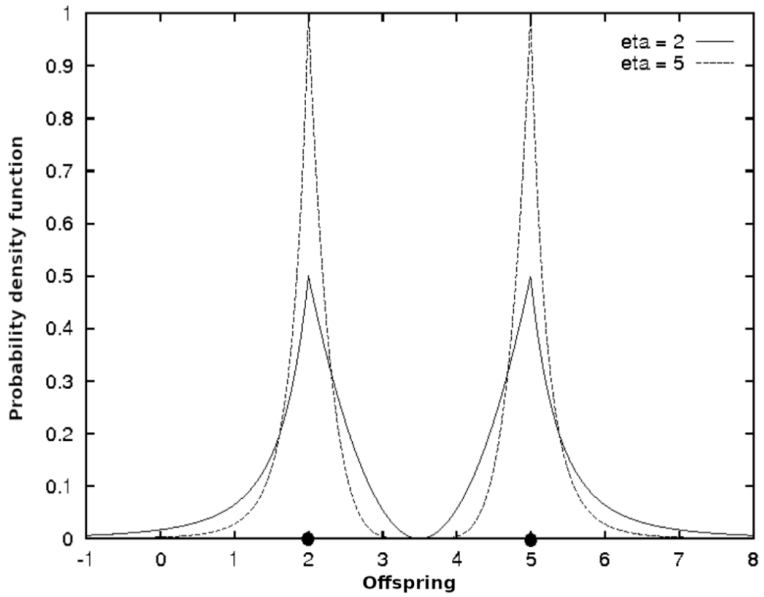

Figure 2.20: Example of SBX [6]

Mutation involves randomly altering variables. It's typically defined with 1 a mutation probability P_m , where $P_m = 1/n$, with n representing the population size. On average, only one variable is changed. Mutation primarily serves the purpose of preventing the optimization algorithm from converging to local optima and allows it to explore different areas of the solution space.

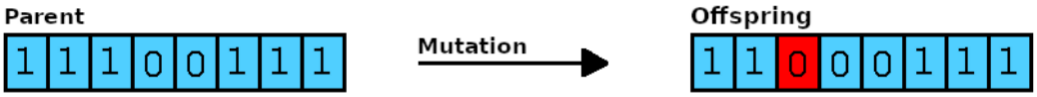

Figure 2.21: Example of mutation

As mentioned earlier, evolutionary algorithms are optimization algorithms based on creating new populations. This characteristic allows for the parallel exploration of a broader search space, providing diversity in solutions. Diverse solutions are vital because they help the algorithm escape local optima and continue the optimization process in areas where better solutions may be found. The concept of a population initially faced criticism for its potential time consuming nature and allowing inferior individuals to persist. However, it has become evident that these inferior solutions are pivotal. Their genetic material can prevent the algorithm from converging prematurely to a local optimum, thereby enabling the pursuit of superior solutions in different regions of the solution space.

2.3.1 Genetic algorithms

In the evolutionary process described thus far, individuals perish only when they are substituted by younger individuals with superior fitness. This leads to a convenient

characteristic where local population statistics, such as maximum, minimum, and average fitness, progressively improve over time, resulting in favorable mathematical convergence properties. However, allowing individuals to persist and reproduce indefinitely can result in a substantial reduction in population diversity and an increased risk of getting stuck in local optima.

To tackle this issue, two common approaches are employed. One approach is to occasionally introduce new individuals to replace existing ones with higher fitness levels, injecting fresh genetic diversity into the population. The other method involves allowing parents to survive for only one generation and then replacing them with their offspring, a strategy typically implemented in genetic algorithms (GAs). Furthermore, GAs rely on objective fitness values to determine which parents are eligible for reproduction.

GAs are some of the earliest, most renowned and widely utilized EAs. The key steps of a GA can be summarized as follows:

1. Generate a population of m parents randomly
2. Establish selection probabilities ($p(i)$) for each parent i, usually based on their fitness
3. Create m offspring by probabilistically selecting parents for reproduction
4. Retain only the offspring to continue, replacing the previous generation

These steps enable GAs to effectively explore the search space, ultimately uncovering optimal or near-optimal solutions across a range of problem domains.

2.3.2 NSGA-II

NSGA-II (Non-dominated Sorting Genetic Algorithm) [45] follows the general outline of a genetic algorithm with a modified mating and survival selection. NSGA,the precursor to NSGA-II, faced criticism due to its high computational complexity in non-dominated sorting and its lack of elitism. NSGA-II addresses these challenges.

To clarify the concept of domination, it means that one solution outperforms another if it achieves superior values for all objective functions. If this isn't the case, we refer to the solutions as non-dominated.

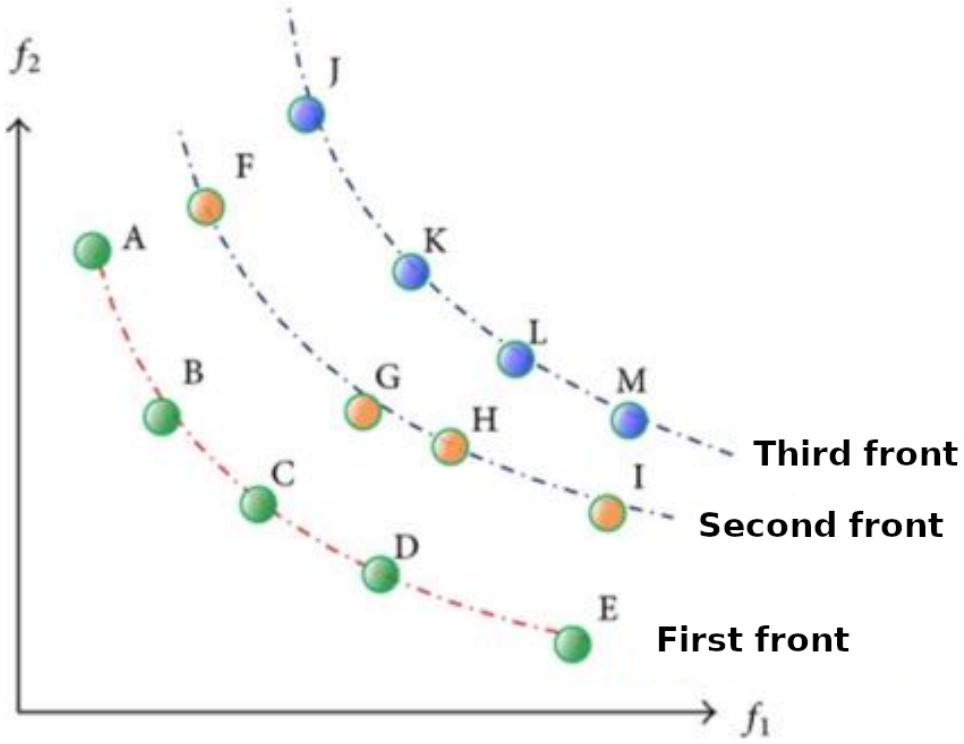

Figure 2.22: Solutions divided into Pareto fronts [46]

For instance, in Figure 2.22, we can assert that solution A dominates solutions F and J because it exhibits better values across all objective functions. However, solutions A and I are considered non-dominated because while solution A excels in one objective function, I dominates in the direction of the other function.

The criticism faced by the initial version of NSGA, as mentioned earlier, primarily stemmed from the method used to sort non-dominated solutions. In this algorithm, each solution within the population is individually compared to all other solutions, leading to the sorting of solutions into different Pareto fronts. This process becomes exceptionally complex when dealing with a large number of individuals within the population and a higher number of objective functions. NSGA-II significantly addressed this complexity by introducing a modification that streamlined and expedited the algorithm.

In this modified approach, every solution within the population is compared to a partially populated set of solutions, denoted as P'. For instance, as new solutions are added to this set, each solution, such as p, is evaluated for dominance concerning the previously recorded solutions within P'. To illustrate, when adding a new solution p, it is compared only to the first solution recorded in P'. Subsequently, the third solution is compared to the preceding two, and so forth. If solution p is found to dominate another solution, let's say q, which is one of the previously recorded solutions, then q is removed from the set P'. Conversely, if certain solutions in the set dominate p, it is not included in P'. If neither of these conditions is met, p is included in P'. This approach enables the set to expand and become populated with non-dominated solutions.

Following the assessment of all solutions in the population, the solutions present in P' collectively form the non-dominated front. This process is then reiterated with the remaining solutions, excluding those previously placed in P', until all remaining fronts are established.

Solution	S_p	n_p
A	F, J	0
B	F, G, J, K, L, M	0
C	G, H, I, J, K	0
D	H, I, K, L, M	0
E	I, M	0
F	J	2
G	K, L	2
H	L, M	2
I	M	3
J		4
K		4
L		5
M		6

Solution	S_p	n_p
F	J	0
G	K, L	0
H	L, M	0
I	M	0
J		1
K		1
L		2
M		2

Solution	S_p	n_p
J		0
K		0
L		0
M		0

Table 2.1: Partitioning Solutions into Pareto Fronts: The first column represents solutions within the population, the second column depicts solutions dominated by the solutions in the first column, and the third column indicates the number of dominant solutions [46]

The sorting of non-dominated solutions is the main step in this algorithm. Other steps are identical to those in the standard EA. The first step of NSGA-II involves randomly creating a population P_0 , of size n, which is then used to create a new population, i.e., offspring, Q_0 , also of size n, through mutation and crossover. These two populations are combined into $R_0 = P_0 \cup Q_0$, a population that now contains $2n$

individuals. From these $2n$ individuals, it is necessary to select the top n, thus creating a new population P_1 . This is achieved by applying the aforementioned sorting algorithm. All Pareto fronts that can fit entirely into the new population of n individuals propagate and thus populate the new population. This property of propagating optimal solutions to a new generation is called elitism. The last Pareto front that cannot entirely fit into the new population is analyzed from the perspective of preserving diversity of solutions. Solutions that maximize diversity are propagated to the extent necessary to fill the new population. In NSGA-II, a parameter called Crowding distance, illustrated in Figure 2.23, is used to determine population density, which in turn helps in evaluating diversity. The crowding distance is the Manhattan Distance in the objective space. However, the extreme points are desired to be kept every generation and, therefore, get assigned a crowding distance of infinity. In this way, solutions with a larger Crowding distance are propagated, while those that are densely "populated" are discarded, thus increasing the diversity of solutions in the new population.

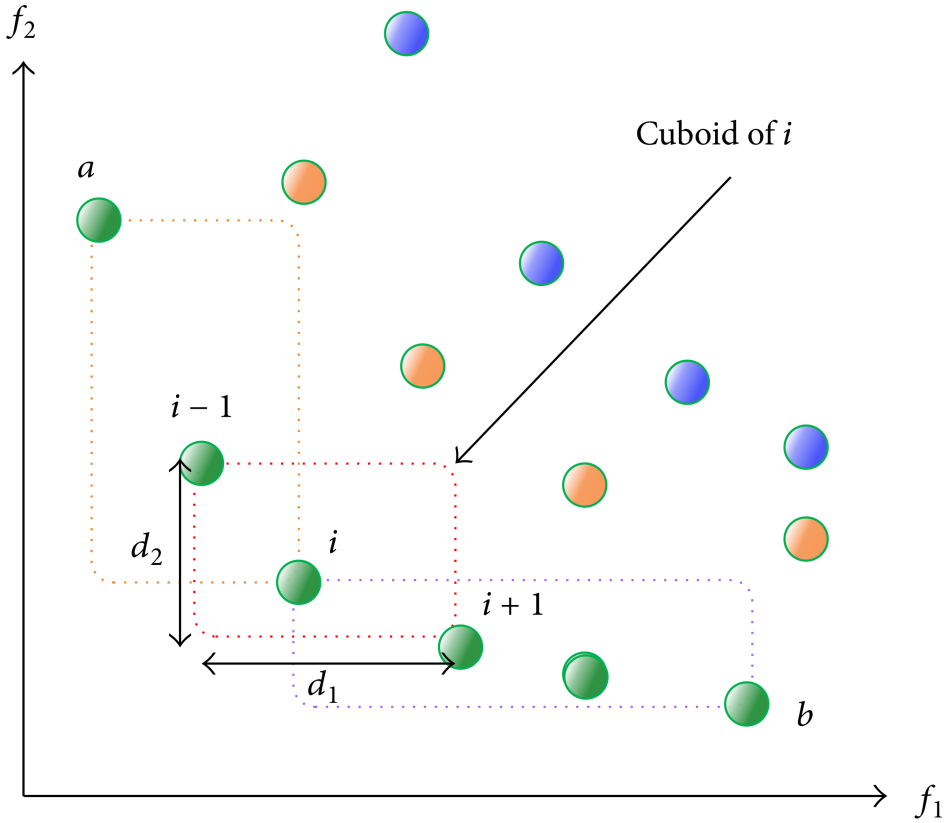

Figure 2.23: Manhattan Crowding distance [46]

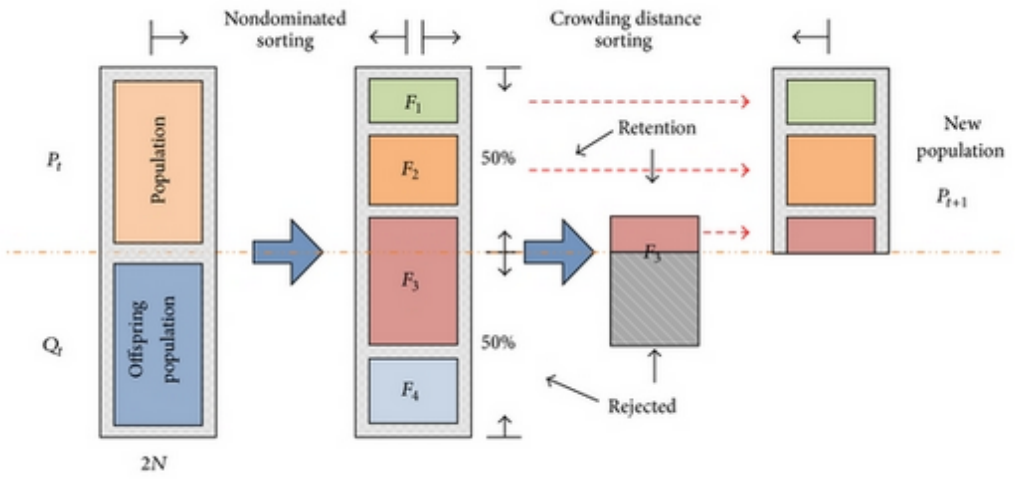

Figure 2.24: NSGA-II [46]

3. Materials and methods

3.1. Hardware

The entire implementation was carried out using the Python programming language within the Anaconda environment, utilizing the PyCharm integrated development environment (IDE). The machine configuration used for performing the experiment is as follows:

- CPU: Ryzen 5 3600, 3.6 GHz, 6 cores/12 thread
- RAM: 24 GB, 2400 MHz DDR4
- GPU: GTX 1070, 8 GB GDDR5, with 1920 CUDA cores

3.2. Dataset

The dataset used for this experiment was MNIST [47]. Comprising 28x28 pixel grayscale images representing handwritten digits from 0 to 9, MNIST is a benchmark dataset for machine learning tasks. Each image was treated as an array of pixel intensities, ranging from 0 (black) to 255 (white). The dataset features 10 classes, one for each digit, and was divided into a training set of 60,000 images and a test set of 10,000 images. Importantly, the dataset splits remained consistent across all model training sessions.

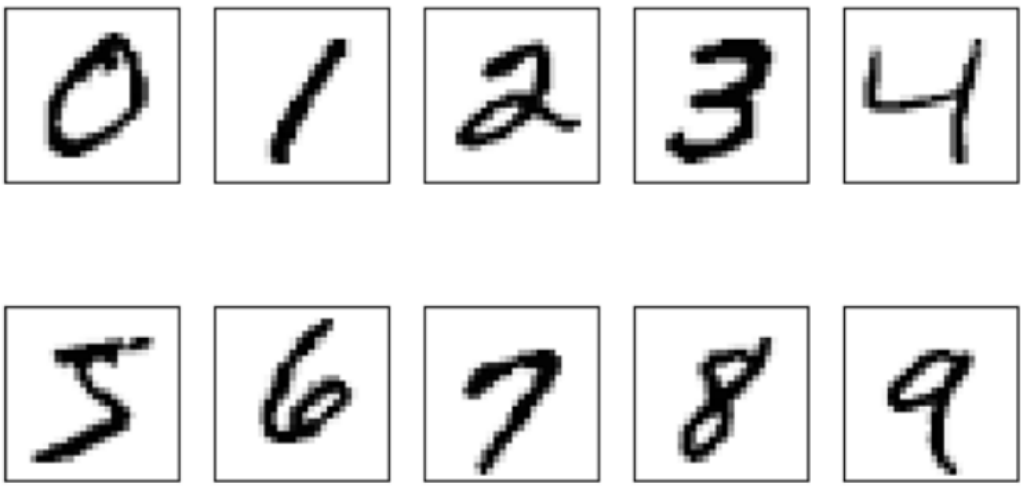

Figure 3.1: Sample digits from MNIST Dataset [6]

3.3. Baseline

The models referenced in [48], [49], and [50], referred to as Model 1, Model 2, and Model 3 respectively, were employed to establish a baseline for classification performance on the MNIST dataset. Model 1 achieved an accuracy of 99.124% with a training time of 123.6 seconds. Model 2 attained an accuracy of 98.904% with a training time of 156 seconds. Model 3 reached an accuracy of 99.27% with a training time of 207.8 seconds. To ensure robustness and account for any potential variations due to the train/test split, each model was trained five times.

	Accuracy [%]	**Training time [s]**
Model 1	99.124	123.6
Model 2	98.904	156
Model 3	99.270	207.8

Table 3.1: Performance of Baseline Models

3.4. Key implementation components

This experiment involved the training and evaluation of over 1000 models, requiring a significant time investment of over 6 days. Although even better results could potentially be achieved with more extended execution time. The primary metrics for comparison in this study were accuracy and training speed. The experiment relied on two key libraries: NumPy and PyTorch.

Key Components of the Algorithm:

- **Modified NSGA-II**
- **Decoder**

The experiment was divided into two distinct stages:

1. **Depth Search:** During this stage, the objective was to determine the optimal depth for the model. This required defining default convolutional and fully connected layers and assessing how their quantity influenced the objectives.
2. **Width Search:** Following the determination of the optimal depth for the CNN, the focus shifted to exploring the width of these layers.

The decision to split the process was made for practicality and to enable a more targeted search. While this approach may have sacrificed optimality to some extent, it considerably accelerated convergence. The primary distinction between the two stages lay in the decoder and population generation.

3.4.1 Modified NSGA-II

The NSGA-II algorithm used here is a slight modification of the standard NSGA-II. In the standard NSGA-II, the only source of diversification in the population is through mutation. Mutation is controlled by a 'mutation probability parameter', but there's a

tradeoff in choosing the size of this parameter. If the 'mutation probability parameter' is too small, mutations will occur in only a small number of samples, which will not introduce much diversity. On the other hand, a large 'mutation probability parameter' will result in too many mutations, creating large steps between generations.

To address this dilemma, a small parameter is used, but in every generation, 5% of the population members are replaced with freshly created members. This step is inspired by controlled immigration and aims to introduce new members with completely different characteristics from the current population members.

In the depth search stage, the population comprised 20 members and underwent 50 generations, resulting in the training of over 90 unique models. A primary challenge encountered here was hardware limitations, particularly concerning larger models that could not fit within GPU memory. To circumvent this issue, these larger models were assigned unfavorable objective values, effectively excluding them from the next generation.

In the width search stage, the population size remains the same as in the depth search stage, and it's allowed to evolve also for 50 generations. This process resulted in the creation of 1000 unique models. However, a challenge encountered during this step was the significant amount of time required to train models in the later generations, with some of them requiring up to an hour to complete.

3.4.2 Decoder

Population members were represented as NumPy arrays, necessitating a decoding step to transform these arrays into PyTorch models. A custom decoder was tailored to suit both stages of the experiment.

In the depth search stage, population members are represented by arrays with a size of 2. The first value represented the number of convolutional layers, while the second indicated the number of fully connected layers. Since this stage does not focus on width of the layers, the layers had to be predefined. The convolutional layers are characterized by a 3x3x2^{n_i} kernels, where i is $i-th$ convolutional layer and n_i is $i-th$ value in a array obtained by this algorithm:

1. Create an empty list a
2. Initialize an integer j to 0
3. $\forall i \in [0, \text{num\_of\_convolutional\_layers-1}]$
   if $i \equiv 0$ $(mod\ 3)$ and $i \neq 0$ increment j by 2
4. Append the value $i + min\_order - j$ to the list a

This algorithm is introduced to slow down the exponential growth of the number of filters for deeper networks. Algorithm operates on the pattern of increasing by three orders before decreasing by one order, which can be seen in Figure 3.2

Minimal order is set to 4, as 2^4 is a sufficient number of filters to learn basic shapes, but it's not too large for a starting point. For example, without this algorithm, the number of filters in the $15-th$ layer would be 32768, but with it it's 1024. Since we are hardware limited, this approach allows us to test deeper networks than it would

4	5	6	5	6	7	6	7	8	7	8	9	8	9	10

Figure 3.2: First 15 values obtained by algorithm

16	32	64	32	64	128	64	128	256	128	256	512	256	512	1024

Figure 3.3: Number of filters for first 15 convolutional layers

be possible otherwise.

Number of hidden units in fully connected layer is determined using the same algorithm, but resulting array is reversed, i.e., if k is the number of fully connected layers and l_i is $i-th$ hidden layer, then number of hidden units in l_i is 2^{n_i} . Important to emphasize, the output layer with 10 neurons that gets appended to each model does not contribute to the values in the array.

1024	512	256	512	256	128	256	128	64	128	64	32	64	32	16

Figure 3.4: Number of units in last 15 hidden layers

Population size in the width search stage depends on the model depth selected in the previous stage. Parameters that get explored in this stage are x and y dimensions of the filters, their number and activation function in convolutional layers and number of hidden units and activation function in fully connected layers. This implies that for every convolutional layer we have four parameters, two for each fully connected layer and one extra for the activation function in the output layer. Therefore, population size, if for example model depth is represented by array [2, 5], is 19 ($2 \times 4 + 5 \times 2 + 1 = 19$).

In the depth search stage, we exclusively employed the ReLU activation function. However, in the width search stage, we offer a choice among four distinct activation functions:

1. Relu
2. Softmax
3. Sigmoid
4. Tanh

Due to the modest dimensions of the input, we have applied the same zero padding in both stages. Additionally, we have omitted the utilization of strided convolution and pooling layers in our approach.

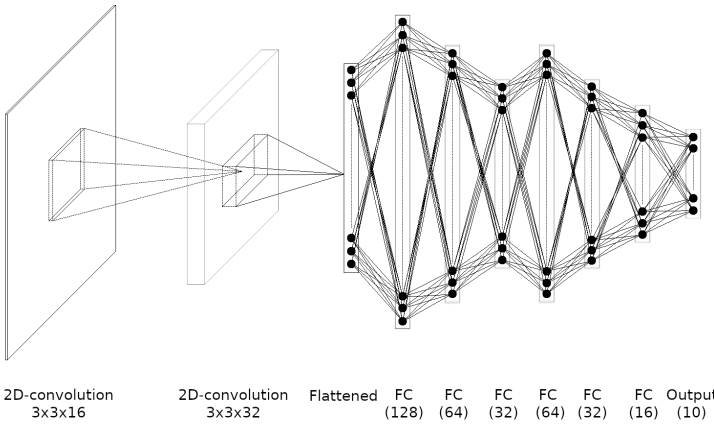

Figure 3.5: Example of CNN represented by [2, 6] array

The simplest model (shown in the Figure 3.6) generated by this experiment was represented by [0, 0], effectively an ANN devoid of convolutional layers.

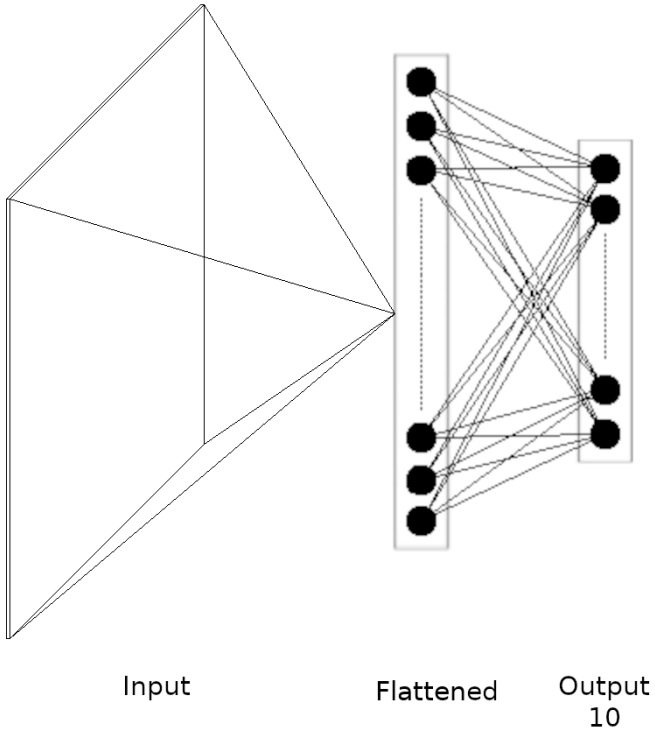

Figure 3.6: Simplest neural network

3.5. Training loop

All the models undergo training for 30 epochs, which is deemed sufficient given the relatively modest complexity of the task. For comparison purposes, we evaluate the models based on two key metrics: accuracy and training time.

The data is divided into batches, with each batch comprising 16 data points. Given that this is a multiclass classification task, we employ the Cross-Entropy Loss as the loss function. In terms of parameter optimization, we utilize the Adam optimizer with a fixed learning rate of 0.001.

4. Results

In this section, we showcase the results we achieved through the utilization of NSGA-II and proceed to make a comparison between the models obtained in depth and width stages.

During the search for an appropriate model depth, it was determined that the most suitable configuration for this task consists of a single convolutional layer followed by a hidden layer, with the output layer positioned at the end. Visual representation of this model architecture is shown in the Figure 4.1. The model achieved an accuracy of 97.78% and underwent training in a mere 110 seconds.

```
(0): LazyConv2d(0, 16, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
(1): ReLU()
(2): Flatten()
(3): LazyLinear(in_features=0, out_features=16, bias=True)
(4): ReLU()
(5): LazyLinear(in_features=0, out_features=10, bias=True)
(6): ReLU()
```

Figure 4.1: Model architecture after depth search

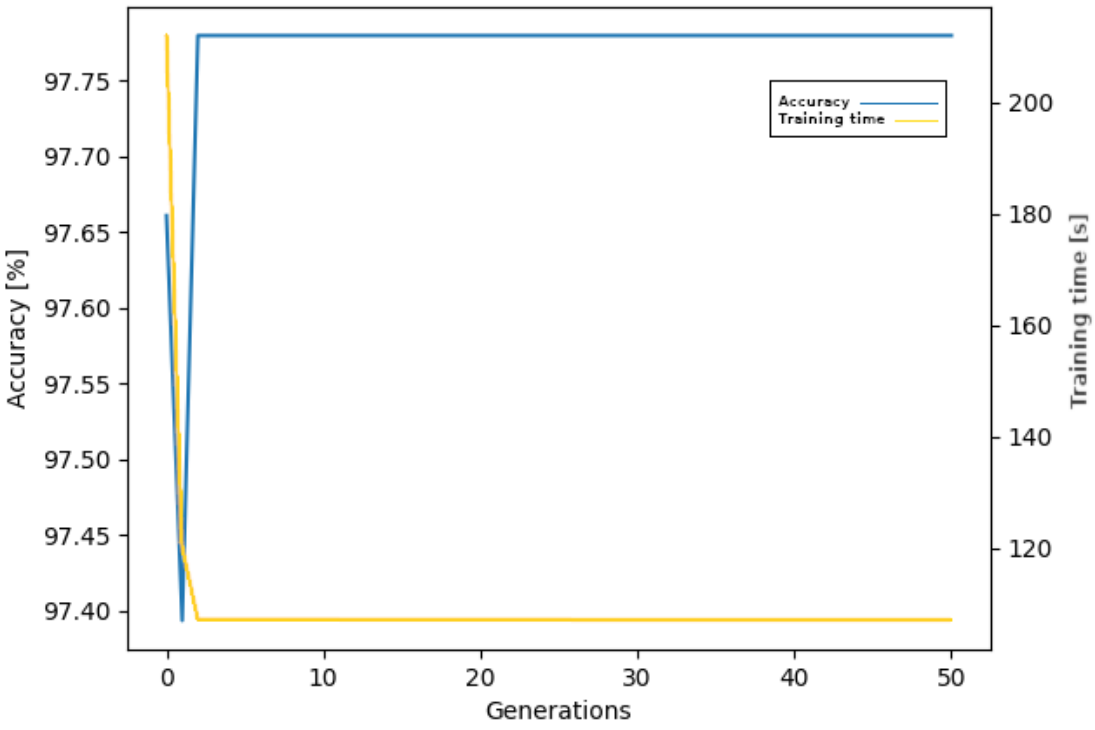

Figure 4.2: Accuracy vs time across generations in depth search

Given that the model derived from the depth search already displayed a high level of accuracy and rapid training times, there was limited scope for further enhancement. Nonetheless, by adjusting the width of the layers and experimenting with different activation functions, were observed improvements of 1% in accuracy. Final model architecture is illustrated in Figure 4.3.

```
(0): LazyConv2d(0, 11, kernel_size=(8, 12), stride=(1, 1), padding=(4, 6))
(1): ReLU()
(2): Flatten()
(3): LazyLinear(in_features=0, out_features=16, bias=True)
(4): ReLU()
(5): LazyLinear(in_features=0, out_features=10, bias=True)
(6): ReLU()
```

Figure 4.3: Model architecture after width search

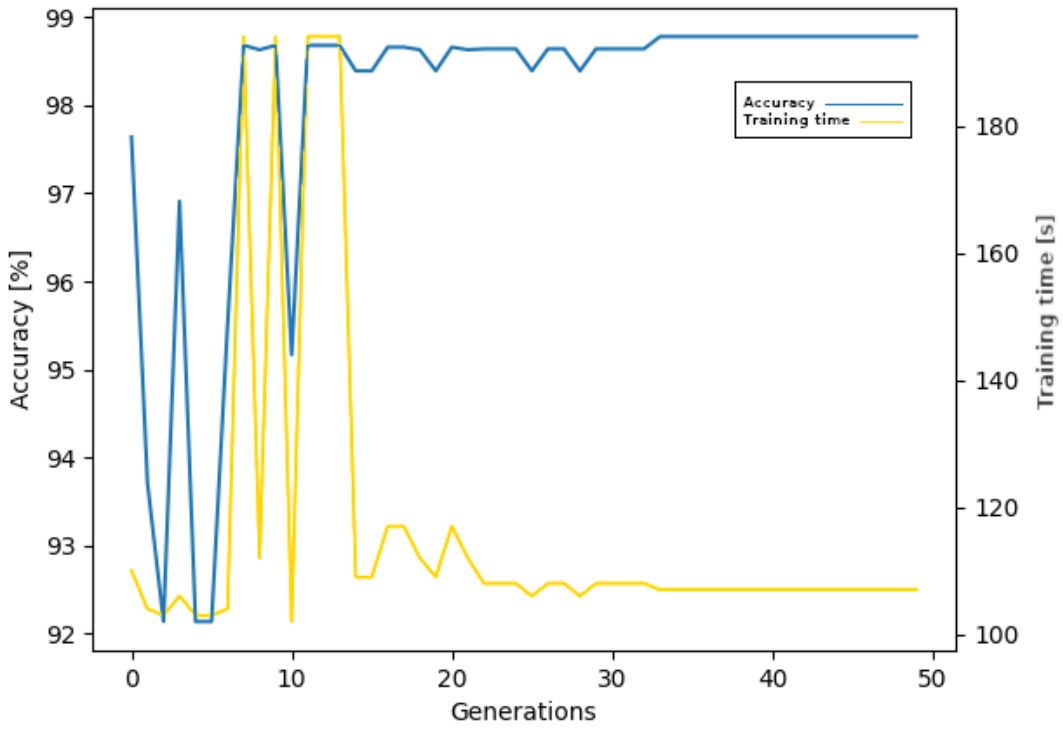

Figure 4.4: Accuracy vs time across generations in width search

Figures 4.2 and 4.4 display changes in accuracy and time of training across various generations. Figure 4.2 reveals that the algorithm quickly identified the optimal network depth for the problem. This is attributed to the problem's relatively low complexity, making a shallow network sufficient for its resolution. Deeper insight is provided in Figure 4.4, where it becomes evident that the algorithm is willing to trade off small accuracy gains to reduce training time and vice versa.

4.1. Benchmarking Results

When compared to the baseline models referenced in [48], [49], and [50], the performance of the models obtained through the utilization of NSGA-II demonstrate a competitive trade-off between accuracy and computational efficiency. Model derived from the depth search achieved an accuracy of 97.78% with a training time of 110 seconds. Although this accuracy is slightly lower than that of the baseline models, it is notable that the training time was significantly reduced compared to Models 2 and 3. Model derived from the width search further improved accuracy to 98.78% while also reducing the training time to 107 seconds. While the baseline Model 3 achieved the highest accuracy at 99.27%, the models obtained through this process offer a more efficient training process. This efficiency could be particularly advantageous in applications where computational resources or time are limited, thereby providing a balanced alternative to the higher accuracy of the baseline models.

	Accuracy [%]	**Training time [s]**
Model 1	99.124	123.6
Model 2	98.904	156
Model 3	99.270	207.8
Depth Search Model	97.78	110
Width Search Model	98.78	107

Table 4.1: Comparison with Baseline Models

5. Discussion

The results obtained from the application of the NSGA-II algorithm indicate that the derived models provide a viable and efficient alternative to the baseline models referenced in [48], [49], and [50]. Although the models derived through depth search exhibit slightly lower accuracy rates, 97.78% and 98.78%, compared to the highest baseline accuracy of 99.27%, they offer significant improvements in training time, with reductions to 110 seconds and 107 seconds, respectively. This trade-off between accuracy and computational efficiency is particularly noteworthy, as it highlights the potential of these models in environments where computational resources are constrained or where faster model training is essential. The efficiency gains achieved by the NSGA-II derived models suggest that they could be highly suitable for real-world applications where time and resource management are critical considerations. Moreover, the results demonstrate the effectiveness of NSGA-II in optimizing model performance across multiple objectives, reinforcing its value as a tool for developing balanced solutions that meet both accuracy and efficiency requirements. While the baseline models set a high standard in terms of accuracy, the solutions obtained through NSGA-II provide a compelling alternative, particularly when computational efficiency is a priority.

Future enhancements could encompass conducting the experiment on more robust hardware, particularly GPUs with larger memory capacities. This would allow for training of much larger models.

In this scenario, larger models were unnecessary due to the relatively simple task of digit recognition. Such a task does not demand deep networks, and the dimensions of the input images are relatively small. While the MNIST dataset served as a suitable demonstration for this project, future endeavors could involve testing on larger datasets like ImageNet. The ImageNet dataset comprises over a million samples categorized into 1000 classes, making the classification task significantly more challenging. Additionally, ImageNet inputs are typically scaled to a size of 224x224, which is 81 times larger than the MNIST input size. This increase in size would enable the utilization of pooling operations and strided convolutions, introducing much greater variability among population members.

Another avenue for improvement involves the merger of both stages, enabling the simultaneous exploration of depth and width. This integrated approach could lead to the exploration of a more extensive model space and potentially yield superior solutions.

6. Conclusion

In this thesis, an NSGA-II-based algorithm was developed to identify optimal CNN architectures for multiclass classification, demonstrated using the MNIST dataset. The selected CNN achieved both high accuracy and simplicity. This work offers a practical tool for efficient network development, emphasizing the potential of evolutionary algorithms in deep learning.

The primary limitation of the presented algorithm is the extended execution time, which spanned a total of six days. It took one day to identify the optimal depth of the network and an additional five days to fine-tune the parameters of its layers. However, upon closer examination, it becomes apparent that the fine-tuned network improvements are relatively modest when compared to the network without layer parameter tuning.

In practical work settings where expedited time-to-production is a critical factor, it is advisable to consider omitting the tuning phase. This observation underscores the need for a balanced approach, weighing the benefits of parameter tuning against the time and computational resources it demands.

To extend the research on the proposed algorithm, it would be interesting to assess its performance across diverse datasets, particularly those featuring larger image dimensions. This choice would enable the use of pooling layers and convolution without padding, thereby enhancing the flexibility to construct more complex neural networks.

Bibliography

[1] Mohamed Elgendy. *Deep Learning for Vision Systems*. 1st ed. Manning Publications, 2020. ISBN: 1617296198,9781617296192.

[2] Geoffrey F. Miller, Peter M. Todd, and Shailesh U. Hegde. "Designing neural networks using genetic algorithms". In: *Proceedings of the Third International Conference on Genetic Algorithms*. George Mason University, USA: Morgan Kaufmann Publishers Inc., 1989, pp. 379–384. ISBN: 1558600063.

[3] Thomas Elsken, Jan Hendrik Metzen, and Frank Hutter. *Neural Architecture Search: A Survey*. 2019. arXiv: 1808.05377 [stat.ML]. URL: https://arxiv.org/abs/1808.05377.

[4] Geoffrey F. Miller, Peter M. Todd, and Shailesh U. Hegde. "Designing neural networks using genetic algorithms". In: *Proceedings of the Third International Conference on Genetic Algorithms*. George Mason University, USA: Morgan Kaufmann Publishers Inc., 1989, pp. 379–384. ISBN: 1558600063.

[5] Lingxi Xie and Alan L. Yuille. "Genetic CNN". In: *CoRR* abs/1703.01513 (2017). arXiv: 1703.01513. URL: http://arxiv.org/abs/1703.01513.

[6] Sebastian Raschka, Yuxi (Hayden) Liu, and Vahid Mirjalili. *Machine Learning with PyTorch and Scikit-Learn*. Packt Publishing, 2022.

[7] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. http://www.deeplearningbook.org. MIT Press, 2016.

[8] Zsolt Nagy. *Artificial Intelligence and Machine Learning Fundamentals: Develop real-world applications powered by the latest AI advances*. Packt Publishing, 2018.

[9] Vladimir Crnojevic. *Prepoznavanje oblika za inzenjere (Serbian), Pattern Recognition for engineers*. University of Novi Sad, Faculty of Technical Sciences, 2014.

[10] A. Burkov. *The Hundred-page Machine Learning Book*. Andriy Burkov, 2019. ISBN: 9781777005474. URL: https://books.google.rs/books?id=Gc5WzwEACAAJ.

[11] Pamela McCorduck. *Machines who think : a personal inquiry into the history and prospects of artificial intelligence*. 2nd ed. A K Peters/CRC Press, 2004. ISBN: 1-56881-205-1,9781568812052.

[12] David E. Rumelhart, Geoffrey E. Hinton, and Ronald J. Williams. "Learning representations by back-propagating errors". In: *Nature* 323 (1986), pp. 533–536. URL: https://api.semanticscholar.org/CorpusID:205001834.

[13] Jacob Eisenstein. *Natural Language Processing*. 2018. URL: https://github.com/jacobeisenstein/gt-nlp-class/blob/master/notes/eisenstein-nlp-notes.pdf.

[14] Google DeepMind. *AlphaGo*. URL: https://deepmind.google/technologies/alphago/.

[15] Y. Lecun et al. "Gradient-based learning applied to document recognition". In: *Proceedings of the IEEE* 86.11 (1998), pp. 2278–2324. DOI: 10.1109/5.726791.

[16] Yann LeCun and Yoshua Bengio. "Convolutional networks for images, speech, and time series". In: *The Handbook of Brain Theory and Neural Networks*. Cambridge, MA, USA: MIT Press, 1998, pp. 255–258. ISBN: 0262511029.

[17] Keiron O'Shea and Ryan Nash. "An Introduction to Convolutional Neural Networks". In: *CoRR* abs/1511.08458 (2015).

[18] Dario Floreano and Claudio Mattiussi. "Neuroevolution: From architectures to learning". In: *Evol Intell* 1 (Mar. 2008). DOI: 10.1007/s12065-007-0002-4.

[19] Kenneth Stanley et al. "Designing neural networks through neuroevolution". In: *Nature Machine Intelligence* 1 (Jan. 2019). DOI: 10.1038/s42256-018-0006-z.

[20] Xin Yao. "Evolving artificial neural networks". In: *Proceedings of the IEEE* 87.9 (1999), pp. 1423–1447. DOI: 10.1109/5.784219.

[21] Arthur Baars et al. *Search–Based Testing, the Underlying Engine of Future Internet Testing*. Jan. 2011.

[22] Joel Lehman and Risto Miikkulainen. *Neuroevolution*. 2013. URL: http://www.scholarpedia.org/article/Neuroevolution.

[23] Carl Fredriksson. *Digit Recognition*. 2018. URL: https://cfml.se/blog/digit_recognition/.

[24] P. Viola and M. Jones. "Rapid object detection using a boosted cascade of simple features". In: *Proceedings of the 2001 IEEE Computer Society Conference on Computer Vision and Pattern Recognition. CVPR 2001*. Vol. 1. 2001, pp. I–I. DOI: 10.1109/CVPR.2001.990517.

[25] Mahdi Rezaei, Hossein Ziaei Nafchi, and Sandino Morales. "Global Haar-Like Features: A New Extension of Classic Haar Features for Efficient Face Detection in Noisy Images". In: *Image and Video Technology*. Ed. by Reinhard Klette, Mariano Rivera, and Shin'ichi Satoh. Berlin, Heidelberg: Springer Berlin Heidelberg, 2014, pp. 302–313. ISBN: 978-3-642-53842-1.

[26] Antônio H. Ribeiro et al. *Beyond exploding and vanishing gradients: analysing RNN training using attractors and smoothness*. 2020. arXiv: 1906.08482 [cs.LG]. URL: https://arxiv.org/abs/1906.08482.

[27] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. "ImageNet classification with deep convolutional neural networks". In: *Commun. ACM* 60.6 (May 2017), pp. 84–90. ISSN: 0001-0782. DOI: 10.1145/3065386. URL: https://doi.org/10.1145/3065386.

[28] Jia Deng et al. "ImageNet: A large-scale hierarchical image database". In: *2009 IEEE Conference on Computer Vision and Pattern Recognition*. 2009, pp. 248–255. DOI: 10.1109/CVPR.2009.5206848.

[29] Abien Fred Agarap. *Deep Learning using Rectified Linear Units (ReLU)*. 2019. arXiv: 1803.08375 [cs.NE]. URL: https://arxiv.org/abs/1803.08375.

[30] Alhassan Mumuni and Fuseini Mumuni. "Data augmentation: A comprehensive survey of modern approaches". In: *Array* 16 (2022), p. 100258. ISSN: 2590-0056. DOI: https://doi.org/10.1016/j.array.2022.100258. URL: https://www.sciencedirect.com/science/article/pii/S2590005622000911.

[31] Nitish Srivastava et al. “Dropout: A Simple Way to Prevent Neural Networks from Overfitting”. In: *Journal of Machine Learning Research* 15.56 (2014), pp. 1929–1958. URL: http://jmlr.org/papers/v15/srivastava14a.html.

[32] Pankaj Mathur. *A Simple Multilayer Perceptron with TensorFlow*. 2016. URL: https://medium.com/pankajmathur/a-simple-multilayer-perceptron-with-tensorflow-3effe7bf3466.

[33] Artem Oppermann. *Activation Functions in Deep Learning: Sigmoid, tanh, ReLU*. URL: https://artemoppermann.com/activation-functions-in-deep-learning-sigmoid-tanh-relu/.

[34] Laith Alzubaidi et al. “Review of deep learning: concepts, CNN architectures, challenges, applications, future directions”. In: *Journal of Big Data* 8 (2021). URL: https://api.semanticscholar.org/CorpusID:232434552.

[35] Wenjie Luo et al. *Understanding the Effective Receptive Field in Deep Convolutional Neural Networks*. 2017. arXiv: 1701.04128 [cs.CV]. URL: https://arxiv.org/abs/1701.04128.

[36] Computer Science Wiki. *Max-pooling / Pooling*. URL: https://computersciencewiki.org/index.php/Max-pooling_/_Pooling.

[37] Zhi Han et al. *Deep Convolutional Neural Networks with Zero-Padding: Feature Extraction and Learning*. 2023. arXiv: 2307.16203 [cs.LG]. URL: https://arxiv.org/abs/2307.16203.

[38] Zbigniew Michalewicz Thomas Bäck David B. Fogel. *Handbook of Evolutionary Computation (Computational Intelligence Library)*. Lslf. Computational Intelligence Library. Published in cooperation with the Institute of Physics, 1997. ISBN: 9780750303927,0750303921.

[39] Dan Simon. *Evolutionary optimization algorithms. Biologically-Inspired and Population-Based Approaches to Computer Intelligence*. Wiley, 2013. ISBN: 0470937416,9780470937419.

[40] Kenneth A. De Jong. *Evolutionary computation: a unified approach*. 1st. The MIT Press, 2002. ISBN: 9780262041942,0262041944.

[41] Otman Abdoun, Jaafar Abouchabaka, and Chakir Tajani. *Analyzing the Performance of Mutation Operators to Solve the Travelling Salesman Problem*. 2012. arXiv: 1203.3099 [cs.NE]. URL: https://arxiv.org/abs/1203.3099.

[42] Dr. Anantkumar Umbarkar and P. Sheth. “CROSSOVER OPERATORS IN GENETIC ALGORITHMS: A REVIEW”. In: *ICTACT Journal on Soft Computing (Volume: 6 , Issue: 1)* 6 (Oct. 2015). DOI: 10.21917/ijsc.2015.0150.

[43] Khalid Jebari. “Selection Methods for Genetic Algorithms”. In: *International Journal of Emerging Sciences* 3 (Dec. 2013), pp. 333–344.

[44] Kalyanmoy Deb and Ram Bhushan Agrawal. “Simulated Binary Crossover for Continuous Search Space”. In: *Complex Syst.* 9 (1995). URL: https://api.semanticscholar.org/CorpusID:18860538.

[45] K. Deb et al. “A fast and elitist multiobjective genetic algorithm: NSGA-II”. In: *IEEE Transactions on Evolutionary Computation* 6.2 (2002), pp. 182–197. DOI: 10.1109/4235.996017.

[46] K. H. Chen H. S. Wang C. H. Tu. *Supplier Selection and Production Planning by Using Guided Genetic Algorithm and Dynamic Nondominated Sorting Genetic Algorithm II Approaches*. URL: https://www.hindawi.com/journals/mpe/2015/260205/.

[47] Li Deng. "The MNIST Database of Handwritten Digit Images for Machine Learning Research [Best of the Web]". In: *IEEE Signal Processing Magazine* 29.6 (2012), pp. 141–142. DOI: 10.1109/MSP.2012.2211477.

[48] Nutan. *PyTorch Convolutional Neural Network With MNIST Dataset*. URL: https://medium.com/@nutanbhogendrasharma/pytorch-convolutional-neural-network-with-mnist-dataset-4e8a4265e118.

[49] Jason Brownlee. *How to Develop a CNN for MNIST Handwritten Digit Classification*. URL: https://machinelearningmastery.com/how-to-develop-a-convolutional-neural-network-from-scratch-for-mnist-handwritten-digit-classification/.

[50] Data Tech Notes. *MNIST Image Classification with PyTorch*. URL: https://www.datatechnotes.com/2024/04/mnist-image-classification-with-pytorch.html.

Biography

Milan Ignjic was born on the 12th of November 1993 in Novi Sad. In 2012 he started a Bachelor of Teaching in Mathematics at Faculty of Sciences, University of Novi Sad and finished in 2017 with a GPA of 7.83. In the same year he continued with master studies of Data Science at the same faculty and passed all exams in 2022 with a GPA of 9.23.