AI-Assisted Creative Expression: a Case for Automatic Lineart Colorization

Yliess Hati

February 12, 2023

Contents

List of Abbreviations

Acronyms

ACT-R Adaptive Control of Thought—Rational

AE Autoencoder

AI Artificial Intelligence

ANN Artificial Neural Network

AST Abstract Syntax Tree

CNN Convolutional Neural Network

CV Computer Vision

DDM Denoising Diffusion Model

DL Deep Learning

GAN Generative Adversarial Network

GD Gradient Descent

GPU Graphical Processing Unit

KL-Divergence Kullback-Leibler Divergence

LLM Large Language Model

LSTM Long Short-Term Memory

ML Machine Learning

MLP Multi-Layer Perceptron

MNIST Modified National Institute of Standards and Technology

MSE Mean Squared Error

NN Neural Network

NPU Neural Processing Unit

ReLU Rectified Linear Unit

RLHF Reinforcement Learning from Human Feedback

RNN Recurrent Neural Network

SGD Stochastic Gradient Descent

SVM Support Vector Machine

TPU Tensor Processing Unit

UMAP Uniform Manifold Approximation and Projection

VAE Variational Autoencoder

VI Variational Inference

Abstract

Aknowledgements

Part I

Context

Chapter 1

Introduction

Humans possess the ability to perceive and understand the world allowing us to accomplish a wide range of complex tasks through the combination of visual recognition, scene understanding, and communication. The ability to quickly and accurately extract information from a single image is a testament to the complexity and sophistication of the human brain and is often taken for granted. One of the Artificial Intelligence (AI) field's ultimate goals is to empower computers with such human-like abilities, one of them being creativity, being able to produce something original and worthwhile [43].

Computational creativity is the field at the intersection of AI, cognitive psychology, philosophy, and art, which aims at understanding, simulating, replicating, or in some cases enhancing human creativity. One definition of computational creativity [44] is the ability to produce something that is novel and useful, demands that we reject common beliefs, results from intense motivation and persistence, or comes from clarifying a vague problem. Top-down approaches to this definition use a mix of explicit formulations of recipes and randomness such as procedural generation. On the opposite, bottom-up approaches use Artificial Neural Networks (ANNs) to learn patterns and heuristics from large datasets to enable non-linear generation.

We, as a species, are currently witnessing the beginning of a new era where the gap between machines and humans is starting to blur. Current breakthroughs in the field of AI, more specifically in Deep Learning (DL), are giving computers the ability to perceive and understand our world, but also to interact with our environment using natural interactions such as speech and natural language. ANNs, once mocked by the AI community [33], are now trainable using Gradient Descent (GD) [52] thanks to the massive availability of data and the processing power of modern hardware accelerators such as Graphical Processing Units (GPUs), Tensor Processing Units (TPUs), and Neural Processing Units (NPUs).

Neural Networks (NNs), those trainable general function approximators, gave rise to the field of generative NNs. Specialized DL architectures such as Variational Autoencoders (VAEs) [30], Generative Adversarial Networks (GANs) [15], Denoising Diffusion Models (DDMs) [21], and Large Language Models (LLMs) [4, 60] are used to generate artifacts such as text, audio, images, and videos of

unprecedented quality and complexity.

This dissertation aims at exploring how one could train and use generative NN to create AI-powered tools capable of enhancing human creative expression. The task of automatic lineart colorization act as the example case used to illustrate this process throughout the entire thesis.

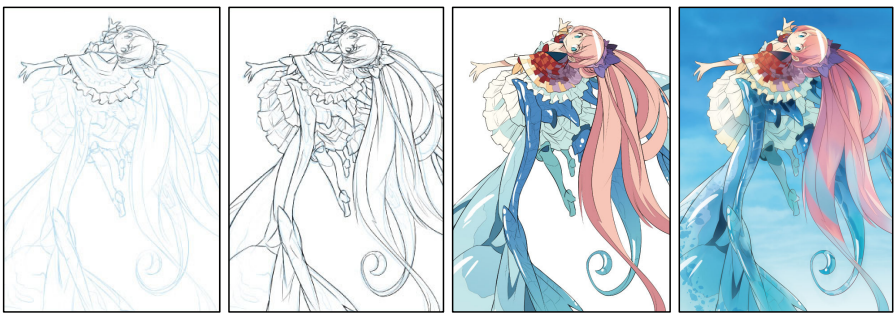

Figure 1.1: Common illustration process. From left to right: sketching, inking, coloring, and pros-processing. Credits: Taira Akitsu

1.1 Motivations

Lineart colorization is an essential aspect of the work of artists, illustrators, and animators. The task of manually coloring lineart can be time-consuming, repetitive, and exhausting, particularly in the animation industry, where every frame of an animated product must be colored and shaded. This process is typically done using image editing software such as Photoshop [48], Clip Studio PAINT [7], and PaintMan [45]. Automating the colorization process can greatly improve the workflow of these creative professionals and has the potential to lower the barrier for newcomers and amateurs. Such a system was integrated into Clip Studio PAINT [7], demonstrating the growing significance of automatic colorization in the field.

The most common digital illustration process can be broken down into four distinct stages: sketching, inking, coloring, and post-processing (see Fig 1.1). As demonstrated by the work of Kandinsky [26], the colorization process can greatly impact the overall meaning of a piece of art through the introduction of various color schemes, shading, and textures. These elements of the coloring process present significant challenges for the Computer Vision (CV) task of automatic lineart colorization, particularly in comparison to its grayscale counterpart [13, 19, 65]. Without the added semantic information provided by textures and shadows, inferring materials and 3D shapes from black and white linearts is difficult. They can only be deduced from silhouettes.

1.2 Problem Statement

One major challenge of automatic lineart colorization is the availability of qualitative public datasets. Illustrations do not always come with their corresponding

lineart. The few datasets available for the task are lacking consistency in the quality of the illustrations, gathering images from different types, mediums and styles. For those reasons, online scrapping and synthetic lineart extraction is the method of choice for many of the contributions in the field [6, 65].

Previous works in automatic lineart colorization are based on the GAN [15] architecture. They can generate unperfect but high-quality illustrations in a quasi realtime setting. They achieve user control and guidance via different means, color hints [6, 11, 37, 47, 55], style transfer [64], tagging [28], and more recently natural language [21]. One common pattern in these methods is the use of a feature extractor such as Illustration2Vec [54] allowing to compensate for the lack of semantic descriptors by injecting its feature vector into the models.

1.3 Contributions

This work focuses on the use of color hints in the form of user strokes as it fits the natural digital artist workflow and does not involve learning and mastering a new skill. While previous works offers improving quality compared to classical CV techniques, they are still subject to noisy training data, artifacts, a lack of variety, and a lack of fidelity in the user intent. In this dissertation we explore the importance of a clean, qualitative and consistent dataset. We investigate how to better capture the user intent via natural artistic controls and how to reflect them into the generated model artifact while preserving or improving its quality. We also look at how the creative process can be transformed into a dynamic iterative workflow where the user collaborates with the machine to refine and carry out variations of his artwork.

Here is a brief enumeration of this thesis's contributions:

- We present a recipe for curating datasets for the task of automatic lineart colorization [16, 17]
- We introduce three generative models:
  - PaintsTorch [16], a double GAN generator that improved generation quality compared to previous work while allowing realtime interaction with the user.
  - StencilTorch [17], an upgrade upon PaintsTorch, shifting the colorization problem to in-painting allowing for human collaboration to emerge as a natural workflow where the input of a first pass becomes the potential input for a second.
  - StablePaint, an exploration of DDM for bringing more variety into the generated outputs allowing for variation exploration and conserving the iterative workflow introduced by StencilTorch for the cost of inference speed.
- We offer an advised reflection on current generative AI ethical and societal impact.

1.4 Concerns

Recent advances in generative AI for text, image, audio, and video synthesis are raising important ethical and societal concerns, especially because of its

availability and ease of use. Models such as Stable Diffusion [50] and more recently Chat-GPT [5] are disturbing our common beliefs and relation with copyright, creativity, the distribution of fake information and so on.

One of the main issues with generative AI is the potential for model fabulation. Generative models can create entirely new, synthetic data that is indistinguishable from real data. This can lead to the dissemination of false information and the manipulation of public opinion. Additionally, there are ambiguities surrounding the ownership and copyright of the generated content, as it is unclear who holds the rights to the generated images and videos. Training data is often obtained via online scrapping and thus copyright ownership is not propagated. This is especially true for commercial applications.

Another important concern is the potential for biases and discrimination. These models are trained on large amounts of data, and if the data is not diverse or representative enough, the model may perpetuate or even amplify existing biases. The Microsoft Tay Twitter bot [62] scandal is an outcome of such a phenomenon. This initially innocent chatbot has been easily turned into a racist bot perpetuating hate speech. The task was made easier because of the inherently biased dataset it was trained on.

In this work, we are committed to addressing and raising awareness for these concerns. The illustrations used for training our models and for our experiments are only used for educational and research purposes. We only provide recipes for reproducibility and do not distribute the dataset nor the weights resulting from model training, only the code. We hope this will not ensure that our work is used ethically and responsibly but limit its potential misuse.

1.5 Outline

The first part of this thesis (chapters 1-3) provides context to the recent advances in generative AI and introduces the CV task of user-guided automatic lineart colorization, its challenges, and our contributions to the field. It then provides additional background, from DL first principles to current architectures used in modern generative NN, and introduces the methodology used throughout the entire document. This part should be accessible to the majority, experts and non-experts, and serve as an introduction to the field.

The second part (chapters 4-7) presents our contributions, some of which have previously been presented in [16, 17]. It introduces into detail our recipe for sourcing and curating consistent and qualitative datasets for automatic lineart colorization, PaintsTorch [16] our first double generator GAN conditioned on user strokes, StencilTorch [17] our in-painting reformulation introducing the use of masks to allow the emergence of iterative workflow and collaboration with the machine, and finally StablePaint, an exploration of the use of DDM models for variations qualitative exploration.

The third and final part (chapters 7-8) offers a detailed reflection on this thesis's contributions and more generally about the field of generative AI ethical and societal impact, identifies the remaining challenges and discusses future work.

The code base for the experiments and contributions is publicly available on

GitHub at https://github.com/yliess86.

Chapter 2

Background

This chapter introduces the reader to the field of Deep Learning (DL) from first principles to the current architectures used in modern generative AI. The first section (section 1) presents a brief history of AI to ground this technical dissertation into its historical context. The following sections (sections 2-4) are discussing the first principles of modern DL from the early Perceptron to more modern frameworks such as Large Language Models (LLMs).

```
# This is a code snippet
print("Hello World!")
```

Additional code snippets (see Lst 2) are included to make this chapter more insightful and valuable for newcomers.

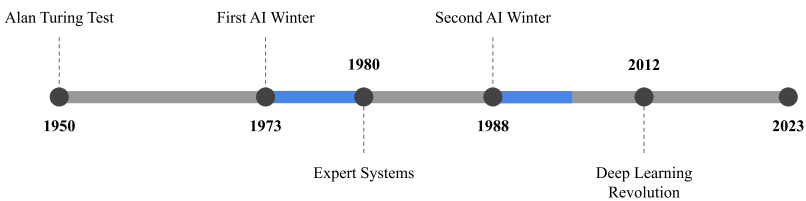

Figure 2.1: A brief timeline of the History of Artificial Intelligence (AI).

2.1 A Brief History of Artificial Intelligence

The history of the field of AI is not a simple linear and straightforward story. The field had its success and failures. The term Artificial Intelligence (AI) has first been introduced in 1956 by John Mc Carthy and Marvin Lee Minsky at a workshop sponsored by Dartmouth College [40], gathering about twenty researchers and intellectuals such as the renowned Claude Shannon (see Fig 2.2). The field's main questions were supposed to be solved in a short period.

However, the reality has been far less rosy. Over the years, AI has gone through several "winters", periods of inactivity and disillusion where funding was cut and research interest dropped (see Fig 2.1). But with the advent of Big Data and the rise of Deep Learning (DL), AI is once again in the spotlight. The following sections provide a brief overview of the history of AI, from its early days to the current state of the field. For a more in-depth look at the history of modern AI, DL, we recommend "Quand la machine apprend" from Yann LeCun [33].

Figure 2.2: Photography of seven of the Dartmouth workshop participants. From left to right: John McCarthy, Marvin Lee Minsky, Nathaniel Rochester, Claude Elwood Shannon, Ray Solomonoff, Trenchard More, and Oliver Gordon Selfridge. Credit: Margaret Minksy

2.1.1 The Early Years

The term Artificial Intelligence (AI) was first used at the 1956 Dartmouth Workshop [40], where John McCarthy proposed the idea of creating a machine that could learn from its mistakes and improve its performance over time. The twenty researchers and intellectuals present worked on topics such as the automatic computer, the use of natural language by machines, neuron nets (Neural Network (NN)), randomness and creativity, and many more. This was a revolutionary idea at the time, and the work done at Dartmouth attracted a great deal of attention and funding.

Much of the early research focused on symbolic AI, which uses symbols and logical operations to represent and manipulate data. Logic programming, production rules, semantic nets and frames, knowledge-based systems, symbolic mathematics, automatons, automated provers, ontologies and other paradigms were at the core of symbolic AI [53]. This approach was based on the early work of Alan Turing and the development of functional languages such as the LISP by McCarthy and al. at MIT [39].

One significant contribution of this period was the Perceptron by Frank Rosenblatt [51], a simplified biomimical model of a single neuron. This artificial neuron fires when the weighted sum of its input is above a predefined threshold. The weights, scalars attributed to the connection edges of the neuron's inputs, are tuned iteratively and manually given supervised data, inputs with corresponding labels, until good enough classification accuracy is met.

2.1.2 The First AI Winter

The Perceptron was an early example of a connectionist approach, which uses a network of artificial neurons to process data. The Perceptron was met with much enthusiasm but was eventually criticized by Marvin L. Minsky and Seymour Papert [42], who argued that it could not solve a simple XOR problem. The criticisms, as well as other issues, led to a period of disillusion in the field of AI, known as the "First AI Winter". It was a time when AI research lost its momentum and funding was not abundant anymore. This period lasted from 1973 to 1980.

2.1.3 Expert Systems and Symbolic AI

The eighties saw a resurgence of interest in AI. Expert systems [24] were the new hot AI topic. They are made of hierarchical and specialized ensembles of symbolic reasoning models and are used to solve complex problems. Symbolic AI continued to prosper as the dominant approach until the mid-nineties.

During this period, AI was developed as logic-based systems, search-based systems using depth-first-search, and genetic algorithms, requiring complex engineering and domain-specific knowledge from experts to work. It was also the time of the first cognitive architectures [36] inspired by advances in the field of neuroscience such as SOAR [32] and Adaptive Control of Thought—Rational (ACT-R) [2] attempting at simulating the human cognitive process for solving and task automation.

Although the connectionist approaches were not well received by the community at the time, some individuals are known for significant contributions that later would form the basis for modern NN architectures. It was the case for Kunihiko Fukushima and his NeoCognitron [12], or David E. Rumelhart et al. who introduced the most used learning procedure for training Multi-Layer Perceptrons (MLPs), the backpropagation [52].

2.1.4 The Second AI Winter

Unfortunately, this period was also marked by a lack of progress because of the resource limitations of the time. Those algorithms required too much power, data, and investments to work. They were not sufficient to make AI truly successful. The lack of progress in the eighties led to the "Second AI Winter". AI research was largely abandoned during this period. Funding and enthusiasm dwindled. This winter lasted from 1988 to early 2000.

2.1.4.1 The Indomitable Researchers

The second AI winter limited research for NN. However, some indomitable individuals continued their work. During this period, Vladimir Vapnik et al. developed the Support Vector Machine (SVM) [8], a robust non-probabilistic binary linear classifier. The method has the advantage to generalize well even with small datasets. Sepp Hochreiter et al. introduced the Long Short-Term Memory (LSTM) for Recurrent Neural Networks (RNNs) [22], a complex recurrent cell using gates to route the information flow and simulate long and short-term memory buffers. In 1989, Yann LeCun provided the first practical and industrial demonstration of backpropagation at Bell Labs with a Convolutional Neural Network (CNN) to read handwritten digits [34, 35] later used by the American postal services to sort letters.

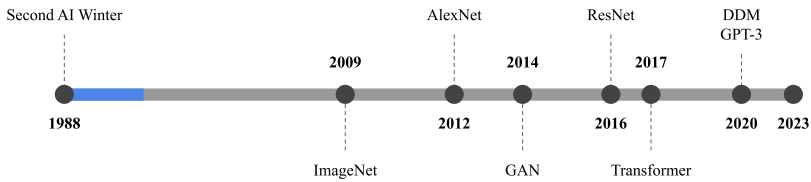

Figure 2.3: A brief timeline of the Deep Learning (DL) Revolution.

2.1.5 The Deep Learning Revolution

The next significant evolutionary step Deep Learning (DL), those deep hierarchical NN, descendants of the connectionist movement, occurred in the early twenty-first century (see Fig 2.3). Computers were now faster and GPUs were developed for high compute parallelization. Data was starting to be abundant thanks to the internet and the rapid rise of search engines and social networks. It is the era of Big Data. NN were competing with SVM. In 2009 Fei-Fei Li and her group launched ImageNet [9], a dataset assembling billions of labeled images.

By 2011, the speed of GPUs had increased significantly, making it possible to train CNNs without layer-by-layer pre-training. The rest of the story includes a succession of deep NN architectures including, AlexNet [31], one of the first award-winning deep CNN, ResNet [18], introducing residual connections, the Generative Adversarial Networks (GANs) [15], a high fidelity and high-resolution generative framework, attention mechanisms with the rise of the Transformer "Attention is all you Need" architecture [60] present in almost all modern DL contributions, and more recently the Denoising Diffusion Model (DDM) [21], the spiritual autoregressive successor of the GAN.

2.1.6 Deep Learning Milestones

DL is responsible for many AI milestones in the past decade (see Fig 2.4). These milestones have been essential in advancing the field and enabling its applications within various sectors. One of the first notable milestones was AlphaGo from

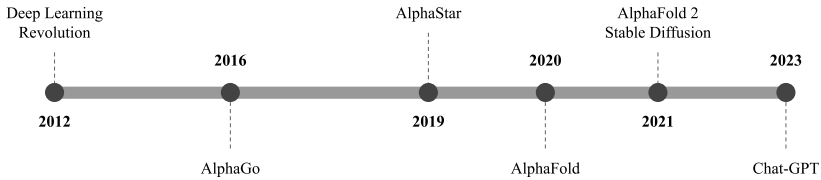

Figure 2.4: A brief timeline of the Deep Learning (DL) Milestones.

DeepMind in 2016 [57], where an AI system was able to beat the Korean world champion Lee Se Dol in the game of Go. AlphaGo is an illustration of the compression and pattern recognition capabilities of deep NN in combination with efficient search algorithms.

In 2019, AlphStar [61] from DeepMind also was able to compete and defeat grandmasters in StarCraft the real-time strategy game of Blizzard. This demonstrated the capability of Deep Learning algorithms to achieve beyond human-level performance in real-time and long-term plannification. In 2020, AlphaFold [56] improved the Protein Folding competition by quite a margin, showing that DL could be used to help solve complex problems that have implications for medical research and drug discovery. In 2021 a follow-up model, AlphaFold 2 [25], was presented as an impressive successor of AlphaFold, showcasing further advances in this field.

In 2021, Stable Diffusion [50] from Stability AI was released. This Latent DDM conditioned on text prompts allows to generate images of unprecedented quality and met unprecedented public reach. Finally, Chat-GPT [5] was released in 2023 as a chatbot based on GPT3 [4] and fine-tuned using Reinforcement Learning from Human Feedback (RLHF) for natural question-answering interaction publicly available as a web demo. However, these last two milestones are also responsible for ethical and societal concerns about copyright, creativity, and more. This highlights both the potential of Deep Learning algorithms but also the need for further research around their implications.

2.2 Core Principles

This section introduces the technical background necessary to understand this thesis dissertation. It introduces Neural Networks (NNs) from first principles. A more detailed and complete introduction to the field can be found in “the Deep Learning book” by Ian Goodfellow et al [14] or in “Dive into Deep Learning” by Aston Zhang et al. [63].

2.2.1 Supervised Learning

In Machine Learning (ML), problems are often formulated as data-driven learning tasks, where a computer is used to find a mapping $f : X \to Y$ from input space X to output space Y. For example, X could represent data about an e-mail

and Y the probability of this e-mail being spam. In practice, manually defining all the characteristics of a function f that would satisfy this task is considered unpractical. It would require one to manually describe all potential rules defining spam. In ML, the supervised framework offers a practical solution consisting of acquiring label data pairs, $(x, y) \in X \times Y$ for the current problem (see Fig 2.5). In our case, this would require gathering a dataset of e-mails and asking humans to label those as spam or not.

Objective Function: Let us consider such a training dataset containing n independent pairs $\{(x_1, y_1), \ldots, (x_n, y_n)\}$ sampled from the data distribution D, $(x_i, y_i) \sim D$. In ML, we seek for learning a mapping $f : X \to Y$ by searching the space of the candidates function class $\mathcal{F}$. Defining a scalar objective function $L(\hat{y}, y)$ measuring the distance from true label y and our prediction $f(x_i) = \hat{y}_i$ given $f \in \mathcal{F}$, the ultimate objective is to find the function $f^* \in F$ that best satisfy the following minimization problem (see Eq 2.1):

$$f^* = arg \min_{f \in \mathcal{F}} E_{(x,y)\sim D} L(\hat{y}, y) \tag{2.1}$$

The function f^* must minimize the expected loss L over the entire data distribution D. Once such a function is learned one can use it to perform inference and map any element from the input space X to the output space Y.

However, this minimization problem is intractable as it is impossible to represent the entire distribution D. Fortunately, as every pair (x_i, y_i) is independently sampled and identically distributed, the objective can be approximated by sampling and minimizing the loss over the training dataset (see Eq 2.2):

$$f^* \approx arg \min_{f \in \mathcal{F}} \frac{1}{n} \sum_{i=1}^{n} L(\hat{y}_i, y_i) \tag{2.2}$$

Regularization: While simplifying the problem allows us to perform loss minimization, this approximation comes at a cost. This optimization problem can have multiple solutions, a set of functions $\{f_1, \ldots, f_m\} \in F$ performing well on the given training set, but would behave differently outside of the training data and outside of the data distribution. Those functions would not necessarily be able to generalize. To mitigate those concerns, we can introduce a regularization term R into the objective function (see Eq 2.3), a scalar function that is independent of the data distribution and represent a preference on certain function class.

$$f^* \approx arg \min_{f \in \mathcal{F}} \frac{1}{n} \sum_{i=1}^{n} L(\hat{y}_i, y_i) + R(f) \tag{2.3}$$

In the following, we investigate two examples where supervised learning is first applied to a Neural Network (NN) regression problem, and then a NN classification problem. The examples highlight the objective functions composed by the loss and the regularization term for regression and classification respectively.

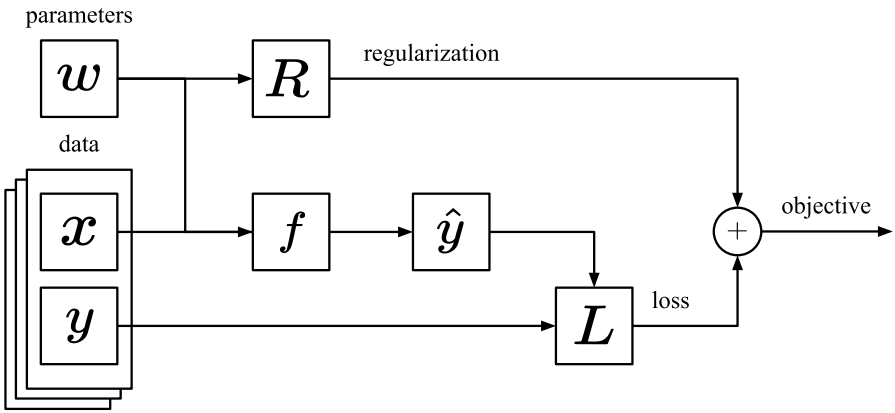

Figure 2.5: Supervised learning data flow. The dataset $(x_i, y_i) \in D$ is used to train the model $f \in \mathcal{F}$ to minimize an objective function with two terms, a data dependant loss L, and a regularization R measuring the system complexity.

Regression Problem: Let us consider the distribution D represented by the *sin* function in the $[-3\pi; 3\pi]$ range (see Fig 2.6). We sample 50 pairs (x_i, y_i) with $X \in [-3\pi; 3\pi]$ and $Y \in [-1; 1]$. Our objective is to learn a regressor f_θ, a three layers NN parametrized by its weights $\{w_0, W_1, w_2\} = \theta$. w_0 contains $(1 \times 16) + 1$ weights, W_1, $(16 \times 16) + 1$, and w_2, $(16 \times 1) + 1$. In this case, the function space is limited to the three layers NN family with 291 parameters $\mathcal{F}$.

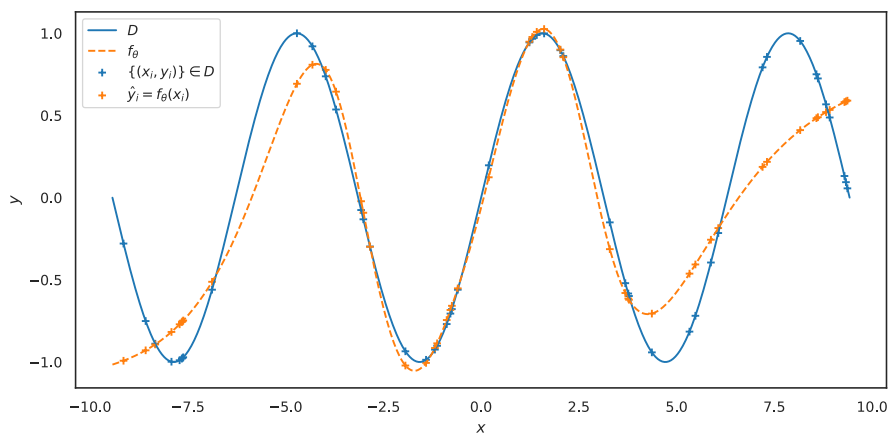

Figure 2.6: Neural Network (NN) regression example. The model f_θ is fit on the training set $(X, Y) \in D$ representing the *sin* function in the range $[-3\pi; 3\pi]$.

To achieve this goal using supervised learning, we can optimize the following objective function (see Eq 2.4):

$$f^* = arg\ \min_\theta \frac{1}{n} \sum_{i=1}^{n} (f_\theta(x_i) - y_i)^2 + \lambda ||\theta||_2^2 \tag{2.4}$$

where the loss is the Mean Squared Error (MSE) $||.||_2^2$ between the ground-truth y_i and the prediction $\hat{y}_i = f_\theta(x_i)$, and the weighted regularization term $\lambda||\theta||_2^2$ to penalize the model for having large weights and converge to a simpler solution. A python code snippet for the objective function and the model is provided below (see Lst 2.2.1):

```
from torch.nn import (Linear, Sequential, Tanh)

# Loss and Regularization
L = lambda y_, y = (y_ - y).pow(2)
R = lambda f: sum(w.pow(2).sum() for w in f.parameters())

# Neural Network model
f = Sequential(
    Linear(1, 16), Tanh(),
    Linear(16, 16), Tanh(),
    Linear(16, 1),
)

# Objective function
C = (1 / n) * L(f(X), Y).sum() + lam *  R(f)
```

Classification Problem: Let us consider the distribution D representing the 2d positions of two clusters $0, 1 \in K$ of moons (see Fig 2.7). We sample 250 moon (x_i, y_i) with $X \in [-1; 1]$ and $Y \in [-1; 1]$. Our objective is to learn a classifier f_θ, a three layers NN parametrized by its weights $\{w_0, W_1, w_2\} = \theta$. w_0 contains $(1 \times 32) + 1$ weights, W_1, $(32 \times 32) + 1$, and w_2, $(32 \times 1) + 1$. In this case, the function space is limited to the three layers NN family with $1,091$ parameters $\mathcal{F}$.

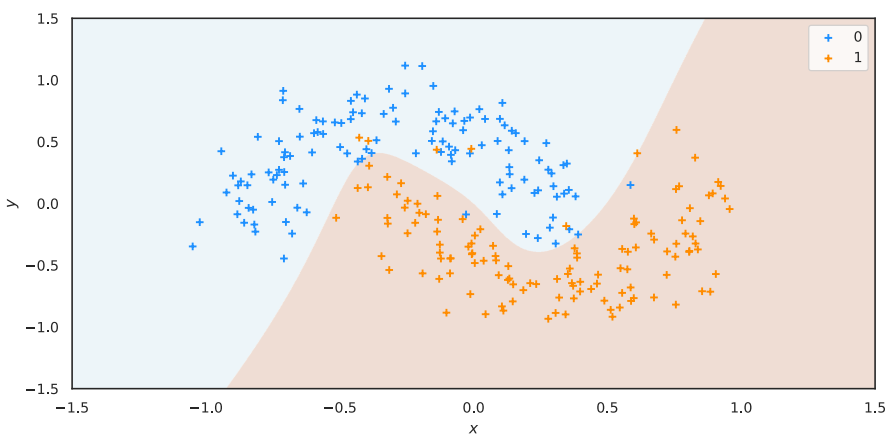

Figure 2.7: Neural Network (NN) classification example. The model f_θ is trained to classify moons based on their positions. The decision boundary is shown.

To achieve this goal using supervised learning, we can optimize an objective function similar to the regression problem (see Eq 2.4) using the cross-entropy as the loss function (see Eq 2.5), measuring the classification discordance.

$$\mathcal{L}(\hat{y}, y) = \sum_{k=1}^{K} y_k \; log \; \hat{y}_k \tag{2.5}$$

A python code snippet for the loss function and the model is provided below (see Lst 2.2.1):

```
from torch.nn import (Linear, Sequential, Tanh)
from torch.nn.functional import cross_entropy

# Loss
L = lambda y_, y = cross_entropy(y_, y, reduce=False)

# Neural Network model
f = Sequential(
    Linear(1, 32), Tanh(),
    Linear(32, 32), Tanh(),
    Linear(32, 1),
)
```

2.2.2 Optimization

In ML, supervised problems can be reduced to an optimization problem where the computer has to find a set of parameters, weights θ, for a given function class $\mathcal{F}$ by optimizing an objective function $\theta^* = arg \; min_\theta \mathcal{C}(\theta)$ made out of two components, a data-dependant loss L and a regularization R.

Random Search: One way to find such a function f_θ that satisfies this objective is to estimate the objective function for a set of random parameter initializations and take the one that minimizes C the most. This θ setting can then be refined by applying random perturbations to the parameters and repeating the operation (see Lst 2.2.2). This is possible due to the fact that we can computer $C(\theta)$ for any value of θ taking the average loss for a given dataset. However, such an approach to optimization is unpractical. NN often comes with millions or billions of parameters θ making random-search intractable.

```
import copy
import numpy as np


for step in range(steps):
    fs, os = [f] + [copy.deepcopy(f) for f in range(n_copy)], []
    for f_ in fs:
        # Apply weight perturbation
        for w in f_.parameters():
            w.normal_(0.0, 1.0 / step)
        # Estimate the objective function
        os.append(C(f_(X), Y))

    # Retrieve the winner
    f = fs[np.argmax(os)]
```

First Order Derivation: A more efficient approach is to make the objective function C and the model f_θ differentiable. This constraint allows us to compute the gradient of the cost C with respect to the model's parameters θ. The value $\nabla_\theta C$ can be obtained using backpropagation (discussed in the next sub-section Sec 2.2.3). This vector of first order derivatives indicates the direction from which we need to move the weights θ away. By taking small iterative steps toward the negative direction of the gradients, we can improve θ. This algorithm is called GD. In practice, due to the very large size of the datasets ($14,197,122$ images for ImageNet [9]), the objective gradient is approximated using a small subset of the training data for each step referred to as a minibatch. This approximation of the GD is called Stochastic Gradient Descent (SGD) (see Lst 2.2.2).

```
for step in range(1_000):
    # Retrieve the next minibatch
    x, y = next_minibatch(X, Y)

    # Compute the objective function and the gradients
    C = L(f(x), y) + lam * R(f)
    C.backward()

    # Update the weights and reset the gradients
    for w in f.parameters():
        w -= eps * w.grad
    f.zero_grad(set_to_none=True)
```

One critical aspect of the SGD algorithm is the hyperparameter ϵ, the learning rate. It controls the size of the step we take toward the negative gradients. If it is too height or too low, the optimization may not converge toward an acceptable local minimum. A toy example is provided in Fig 2.8 where different learning rates are used to find the minimum of the square function $y = x^2$.

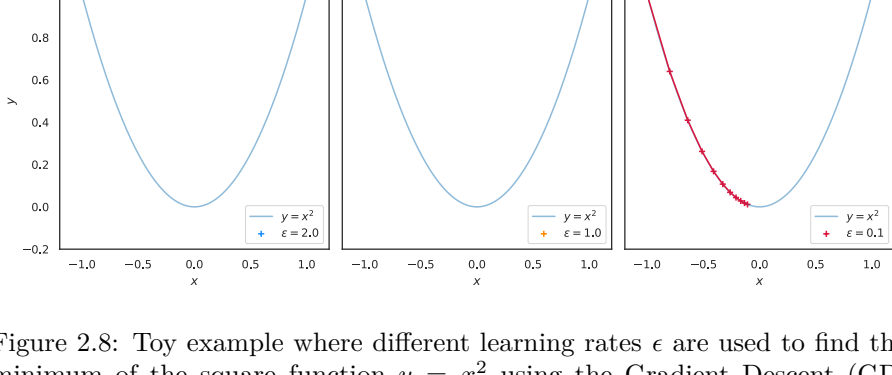

Figure 2.8: Toy example where different learning rates ϵ are used to find the minimum of the square function $y = x^2$ using the Gradient Descent (GD) algorithm starting from $x = -1$. Some learning rate setup result in situations where the optimization does not converge to the solution. A learning rate $\epsilon = 2$ diverges toward infinity, $\epsilon = 1$ is stuck and bounces between two positions -1 and 1. However, a small learning rate $\epsilon = 0.1 < 1$ converges towards the minimum $y = 0$. This example illustrates the impact of the hyperparameter ϵ on GD.

First Order Derivation with Momentum: The DL literature contains abundant work on first order optimizer variants aiming for faster convergence such as SGD with Momentum [49], Adagrad [10], RMSProp [20], Adam [29], and its correction AdamW [38]. A toy example is shown Fig 2.9.

The Momentum update [49] introduces the use of a momentum inspired by physics' first principles to favor small and consistent gradient directions. In this particular case, the momentum is represented by a variable v updated to store an exponential decaying sum of the previous gradients $v := \alpha v + \nabla_\theta C(\theta)$. The weights are then updated using negative v as the gradient direction instead of $\nabla_\theta C(\theta)$.

Other optimizers also make use of the second moment of the gradients. Adagrad [10] uses another variable r to store the second moment $r := r + \nabla_\theta C(\theta) \odot \nabla_\theta C(\theta)$ and modulate the update rule toward the negative direction $\frac{1}{\delta + \sqrt{r}} \odot \nabla_\theta C(\theta)$ where δ is a small value to avoid division by zero. Similarly, RMSProp [20] maintains a running mean of the second moment $r := \rho r + (1 - \rho) \nabla_\theta C(\theta) \odot \nabla_\theta C(\theta)$.

Finally Adam [29], and its correction AdamW [38], are applying both Momentum and RMSProp estimating the first and second moment to make parameters with large gradients take small steps and parameters with low gradients take larger ones. This has the advantage to allow for bigger learning rates and faster convergence at the cost of triple the amount of parameters to store during training. A simple implementation of Adam is shown below (see Lst 2.2.2):

```
# Adam state (parameters, gradients first and second moments)
params = list(f.parameters())
d_means = [w.clone().zeros_() for w in params]
d_vars  = [w.clone().zeros_() for w in params]

for step in range(1_000):
    # Retrieve the next minibatch
    x, y = next_minibatch(X, Y)

    # Compute the objective function and the gradients
    C = L(f(x), y) + lam * R(f)
    C.backward()

    data = zip(params, d_means, d_vars)
    for w_idx, (w, d_m, d_v) in enumerate(data):
        # Update the moments (mean and uncentered variance)
        d_m = beta1 * d_m + (1 - beta1) * w.grad
        d_v = beta2 * d_v + (1 - beta2) * (w.grad ** 2)

        # Compute bias correction
        corr_m = d_m / (1.0 - beta1 ** step)
        corr_v = d_v / (1.0 - beta2 ** step)

        # Update weight and reset the gradient
        w -= eps * (corr_m / (corr_v.sqrt() + 1e-8))
        w.grad = None
```

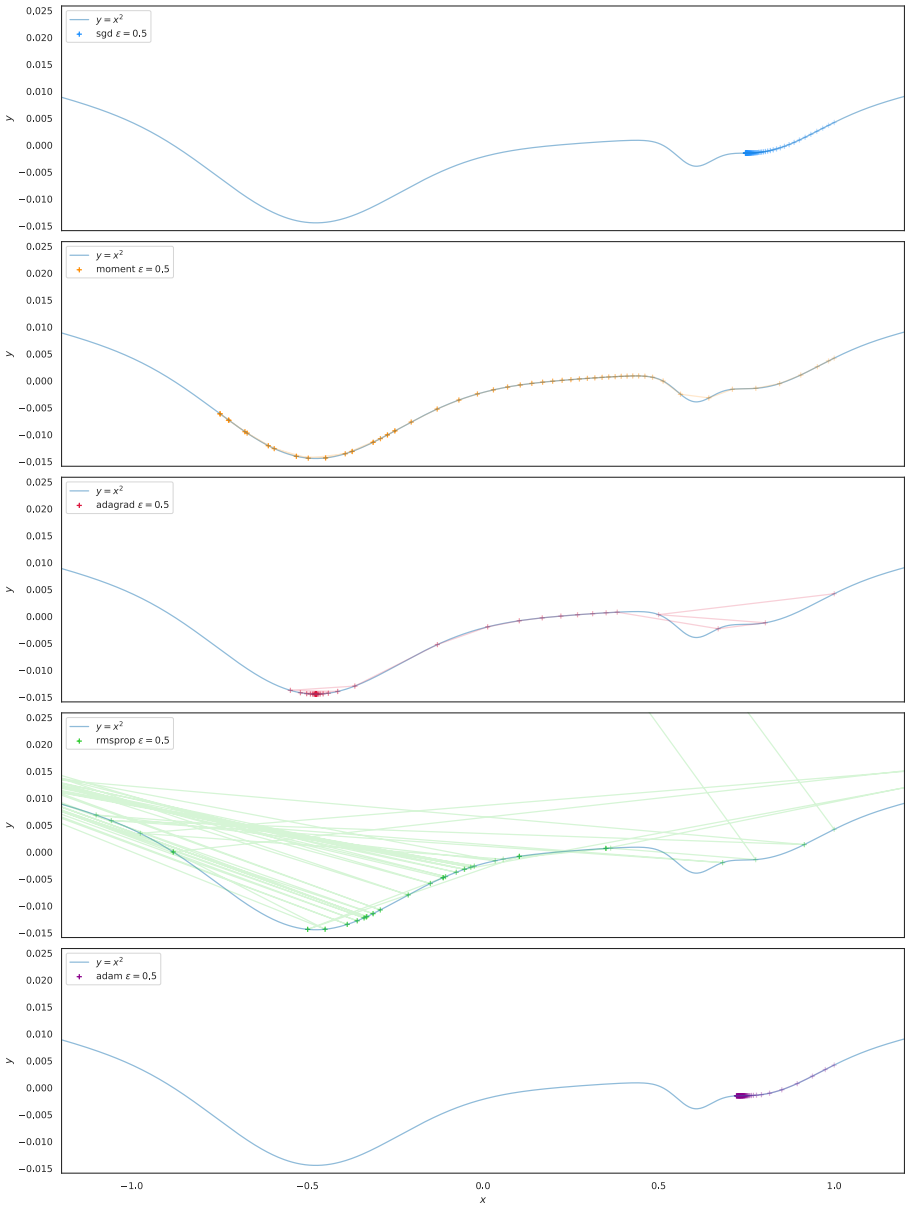

Figure 2.9: This toy example illustrates the impact of the optimizer choice during objective minimization with first order methods. SGD, Momentum, Adagrad, RMSProp and Adam are tasked to find the minimum of a 1-dimensional mixture of Gaussians given the same starting point $x = 1$ and the same learning rate $\epsilon = 0.5$. In this particular setup, Moments and Adagrad find the solution, RMSProp explodes, and SGD and Adam are stuck into a local minimum.

Cross-Validation and HyperParameter Search: As illustrated by the toy examples (see Figs 2.8, 2.9), the training of NN using SGD is highly dependent on the initial setting of hyperparameters. One could ask if there is a rule for choosing such parameters. Unfortunately, this is not the case. The field is highly empirical and driven by exploration using the scientific method.

One common approach is to set up metrics to evaluate the performance of the model during the optimization process. It is a good practice to divide the dataset into validation folds that are different from the training data to evaluate the generalization capabilities of the model. This practice is referred to as k-fold cross-validation and is most of the time in DL, because of the large datasets, reduced to a single fold, called the validation set. By defining such a process, NN can be compared in a controlled manner and the hyperparameter space can be searched. Hyperparameter search is so important that it is a subfield of its own. The broad DL literature however contains many examples of initial parameters and architectures that can be used to bootstrap this search.

2.2.3 Backpropagation

In the previous sub-section (see Sec 2.2.2), we saw how to learn parametrized functions f_θ given a training dataset. By evaluating the gradients of the objective function with respect to the model's parameters, it is possible to obtain a good enough mapping $f_\theta : X \to Y$. In this sub-section, we discuss backpropagation, the recursive algorithm used to efficiently compute those gradients exploiting the chain rule $\frac{\partial z}{\partial x} = \frac{\partial z}{\partial y} \cdot \frac{\partial y}{\partial x}$ with z dependant on y and y on x.

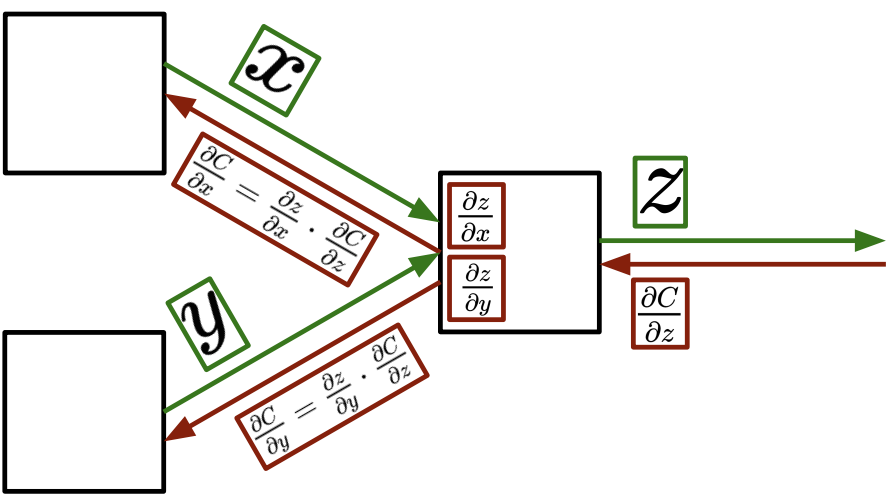

Figure 2.10: Illustration of reverse mode Automatic Differentiation (AD). This Directed Acyclic Graph (DAG) shows the forward pass in green and backward in red. The gradient of an activation is computed by multiplying the local gradient of a node by its output gradient computed in the previous step when following backward differentiation $\frac{\partial C}{\partial x} = \frac{\partial z}{\partial x} \cdot \frac{\partial C}{\partial z}$ where $\frac{\partial z}{\partial x}$ is the location derivative and $\frac{\partial C}{\partial z}$ the output one.

Automatic Differentiation: In mathematics, AD describes the set of techniques used to evaluate the derivative of a function and exploits the fact that any complex computation can be transformed into a sequence of elementary operations and functions with known symbolic derivatives. By applying the chain rule recursively to this sequence of operations, one can automatically compute the derivatives with precision at the cost of storage.

We distinguish two modes of operation for AD, forward mode differentiation, and reverse mode differentiation. In forward mode, the derivatives are computed after applying each elementary operation and function in order using the chain rule. It requires storing the gradients along the way and carrying them until the last computation. This mode is preferred when the size of the outputs exceeds the size of the inputs. This is generally not the case for NN where the input, an image for example, is larger than the output, a scalar for the objective function. On the opposite, reverse mode differentiation traverses the sequence of operations from end to start using the chain rule and requires storing the output of the operations instead. This method is preferred when the size of the inputs exceeds the outputs. This mode thus has to happen in two passes, a forward pass where one computes the output of every operation in the order, and a backward pass, where the sequence of operations is traversed in backward order to compute the derivatives.

Computation Graph: A NN can be defined as a succession of linear transformations followed by non-linear activations (discussed in the next section Sec 2.3). Those elementary operations are differentiable and when thinking of the data flow can be viewed as a computation DAG to which backpropagation, reverse mode differentiation, can be applied.

In modern DL frameworks [1, 46], the AD is centered on the implementation of a Graph object with Nodes. Both entities possess a `forward()` and a `backward()` function. The forward pass calls the `forward()` function of each node of the graph by traversing it in order while saving the node output for differentiation. The backward pass traverses the graph recursively in backward order calling the `backward()` function responsible for computing the local gradient of the node operation and multiplying it by its output gradient following the chain rule. Nodes are in most frameworks referred to as Layers, the elementary building block of the NN operation chain.

Toy Implementation: Here is a simple implementation of such a computation graph for backpropagation and AD engines adapted from Micrograd by Andrej Karpathy [27]. The Node class is responsible for storing the value, the chained gradient, and additional information to trace the graph for the backward pass.

```
from dataclasses import dataclass

@dataclass
class Node:
    value: float
    grad: float = 0.0
    _backward = lambda: None
    _children: set[Node] = {}
    _op = ""
```

The Node can then be populated with elementary operations (`__add__`, `__mul__`) and functions (`tanh`).

```
import numpy as np


class Node:
    ...
    def __add__(self, other: Node) -> Node:
        out = Node(self.value + other.value, {self, other}, "+")

        def _backward() -> None:
            self.grad += out.grad
            other.grad += out.grad
        out._backward = _backward

        return out

    def __mul__(self, other: Node) -> Node:
        out = Node(self.value * other.value, {self, other}, "*")

        def _backward() -> None:
            self.grad += other.value * out.grad
            other.grad += self.value * out.grad
        out._backward = _backward

        return out

    def tanh(self) -> Node:
        act = np.tanh(self.value)
        out = Node(act, {self}, "tanh")

        def _backward() -> None:
            self.grad += (1.0 - act ** 2) * out.grad
        out._backward = _backward

        return out
```

Every elementary transformation needs to be differentiable and implements its own backward function using the chain rule. The chained gradient stored in the node is the multiplication of the local gradient with its output gradient computed when the parent node is encountered during the backward pass. The Node object needs to be extended with support for other elementary operations (e.g. `__pow__`, `__neg__`) and functions (e.g. `sigmoid`, `relu`) to be useful for DL.

We add the ability for a Node to compute its backward pass by first tracing all the current DAG operations recursively. The gradients can then be computed by initializing the first node (the last in the graph) gradient to 1. The backward call on the graph iteratively traverses the graph from end to start and applies the inner backward functions to compute the chain gradients along the way while storing them in their respective Node object.

```
class Node:
    ...
    def backward(self) -> None:
        trace, visited = [], {}
        def trace_graph(node: Node) -> None:
            if node not in visited:
                visited.add(node)
                for child in node._children:
                    trace_graph(child)
                trace.append(node)
        trace_graph(self)

        self.grad = 1.0
        for node in trace[::-1]:
            node._backward()
```

The simple Automatic Differentiation (AD) engine is now ready to perform forward and backward passes. The gradients stored in the node can then be used for Stochastic Gradient Descent (SGD) to update the weights of a Neural Network (NN) for example.

```
w1, w2 = Node(0.1), Node(0.2)  # Weights
a,  b  = Node(1.0), Node(0.0)  # Inputs

z = (w1 * a + w2 * b).tanh()   # Eager forward pass
z.backward()                   # Backward pass
```

Fortunatly open-source implementations of such engines are already available and extensively used by the DL community. They have the adantage to work at the Tensor level, not at the Scalar level like Micrograd, and offer support for accelerated hardware such as Graphical Processing Units (GPUs), Tensor Processing Units (TPUs), and Neural Processing Units (NPUs). In this dissertation, most examples are using the PyTorch [46] framework, a Python Tensor library written in C++ and equipped with a powerful eager mode reverse AD engine.

Eager or Graph Execution: Modern DL frameworks such as PyTorch [46] and Tensorflow [1] now propose two execution modes. An eager mode, where the graph is built dynamically and operations are applied immediately, and a graph mode where the computational graph has to be defined beforehand. Both modes come with advantages and inconveniences. Eager mode is useful for iterative development and provides an intuitive interface similar to imperative programming, it is easier to debug and offers natural control flows as well as hardware acceleration support. On the other side, graph mode allows for more efficient execution. The graph can be optimized by applying operations similar to the ones used in programming language Abstract Syntax Trees (ASTs). Graph edges can be merged into a single fused operation, and execution can be optimized for parallelization. It is often the preferred way for deployment where the execution time and memory are at stake.

2.3 Neural Networks

In the previous section, we described the general setup for ML, where one has to fit a model from a given function family $f \in \mathcal{F}$ on a given dataset $(X, Y) \in D$ optimized using +sdg and backpropagation. This section begins discussing a particular class of parameterized function f_θ called Neural Networks (NNs).

2.3.1 Perceptron

The Perceptron, introduced by Frank Rosenblatt in 1958 [51], is the building block of Neural Networks (NNs). It was introduced as a simplified model of the human neuron, containing three parts: dendrites handling incoming signals from other neurons, a soma with a nucleus responsible for signal aggregation, and an axone responsible for the transmission of the processed signal to other neurons. When the signal aggregation in the soma reaches a predefined threshold, the neuron activates. This phenomenon is called an action potential. Although this is not an accurate representation of the modern neuroscience state of knowledge, this simplified model was believed to be accurate at the time.

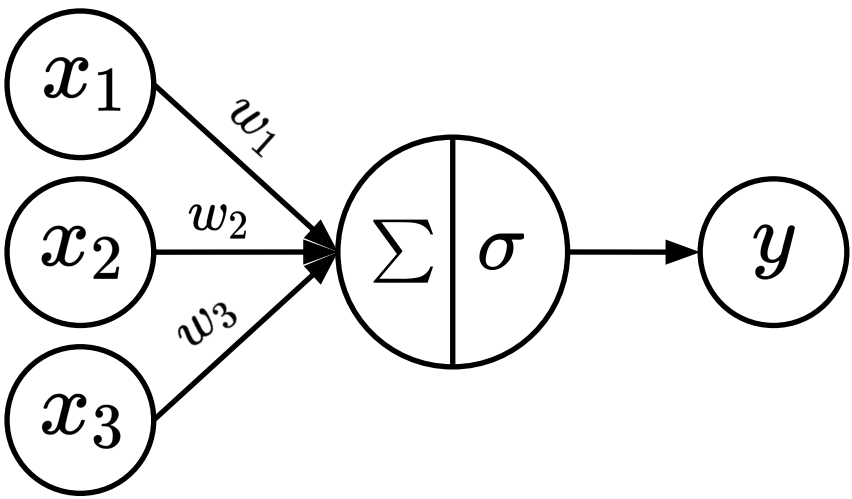

Figure 2.11: Diagram of a Perceptron with three inputs $\{x_1; x_2; x_3\}$. The perceptron computes an activated weighted sum of its inputs $y = \sigma(\sum_{i=1}^{3} w_i \cdot x_i)$ where σ, the activation function is a threshold function.

Similarly, the Perceptron computes a weighted sum of its inputs and activates if a certain threshold is reached (see Fig 2.11). The Perceptron is parametrized by the weights representing the importance attributed to the incoming inputs and are part of the parameters θ that are trained on a given dataset. It can be viewed as a learned linear regressor followed by a non-linear activation, historically a threshold function, a function σ that activates $\sigma(x) = 1$ when $x > 0.5$ and $\sigma(x) = 0$ otherwise (see Lst 2.3.1).

```
def perceptron(self, x: Tensor, W: Tensor) -> Tensor:
    return (x * self.W.T) > 0.5
```

The objective of a perceptron is to learn a hyperplane, a plane with $n-1$ dimensions where n is the number of inputs, that can perform binary classification, separate two classes. However, as mentioned by Marvin L. Minsky and al. in their controversial book Perceptrons [42], a hyperplane regressor cannot solve a simple XOR problem (see Fig 2.12).

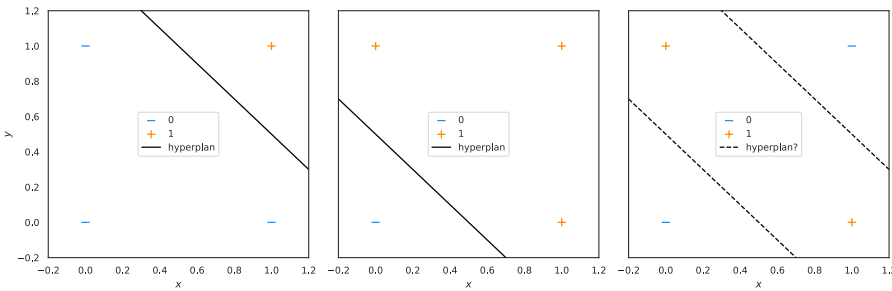

Figure 2.12: Illustration of the Perceptron's decision hyperplane when trained to solve the AND problem on the left, the OR problem in the middle, and the XOR problem on the right. The first two problems are linearly sperable, thus adapted for a Perceptron. However, a single perceptron cannot solve the XOR problem as it is not linearly separable.

2.3.2 Multi-Layer Perceptron

The real value of the Perceptron comes when assembled into a hierarchical and layer-wise architecture, a Neural Network (NN). By repeating matrix multiplications (linear transformations) and non-linearities the network is able to handle non-linear problems and act as a universal function approximator [23]. This arrangement of layered perceptrons is called a Multi-Layer Perceptron (MLP) (see Fig 2.13).

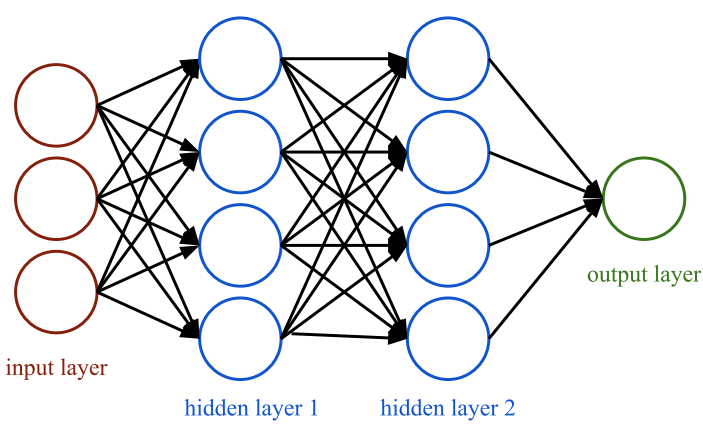

Figure 2.13: Diagram of a 3-layer Multi-Layer Perceptron (MLP). When using the matrix formulation, this arrangement of neurons can be summarized into a single expression $y = \sigma(\sigma(x \cdot W_1^T) \cdot W_2^T) \cdot W_3^T$.

A MLP with Identity as its activation function is useless as its chain of linear transformations can be collapsed into a single one. Since the advent of the Perceptron, the literature has moved away from using threshold functions as activations. Common activation functions are the sigmoid $\sigma(x) = \frac{1}{1+e^{-x}}$, tanh $tanh(x) = \frac{e^z - e^{-z}}{e^z + e^{-z}}$, Rectified Linear Unit (ReLU) $ReLU(x) = max(x, 0)$ functions and variants presenting additional properties such as infinite continuity, gradient smoothness, and more (see Fig 2.14).

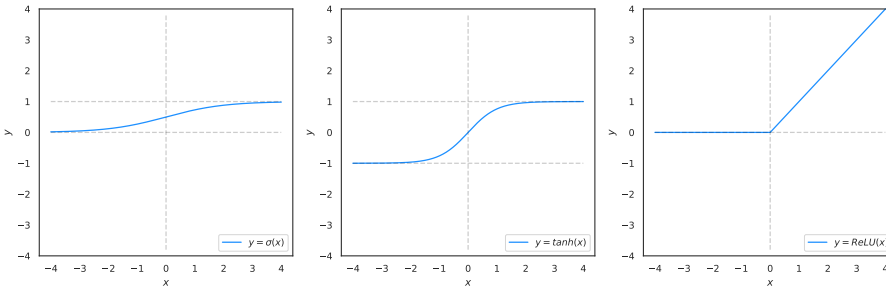

Figure 2.14: Activation functions. Sigmoid $\sigma(x) = \frac{1}{1+e^{-x}}$ acts as a filter $y \in [0; 1]$, tanh $tanh(x) = \frac{e^z - e^{-z}}{e^z + e^{-z}}$ acts as a normalization compressor $y \in [-1; 1]$, ReLU $ReLU(x) = max(x, 0)$ folds all negatives down to zero $y \in [0; +\infty]$.

MNIST Classifier: A classic toy example showing the capabilities of MLPs is the handwritten digit classification challenge on the Modified National Institute of Standards and Technology (MNIST) dataset [59]. MNIST contains $60,000$ training and $10,000$ test examples. It has been written by high school students and gather 28×28 centered black and white handwritten digits from 0 to 9 (see Fig 2.15).

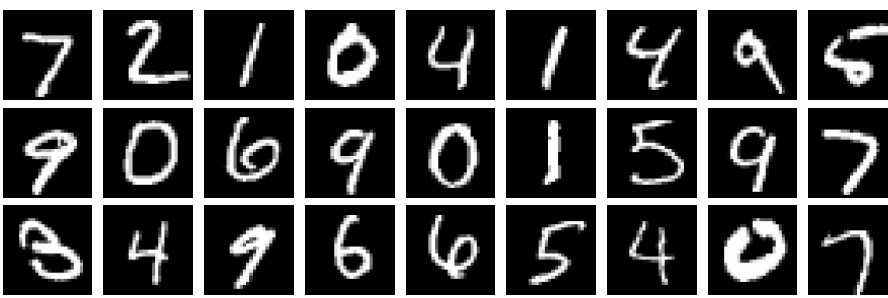

Figure 2.15: First 27 handwritten digits from the Modified National Institute of Standards and Technology (MNIST) dataset. The digits are stored as 28×28 centered black and white images.

Training a MLP on such a challenge is simple and effective. With little training, parameters (according to the DL standards), and no hyperparameter tweaking, a vanilla 3-layer NN with ReLU activations can achieve 97.5 accuracy on the test set. The inputs however need to be transformed before ingestion by the model as MLPs are constrained to 1-dimensional input vectors. The following demonstrates how to implement such a model and train it on MNIST.

```
from torch.utils.data import (Subset, DataLoader)
from torchvision.datasets.mnist import MNIST
from torchvision.transforms.functional import to_tensor

# Load MNIST images as Tensors and Normalize [0; 1]
T = lambda x: to_tensor(x).float().flatten()
dataset = MNIST("dataset", train=True,  transform=T.ToTensor())
testset = MNIST("dataset", train=False, transform=T.ToTensor())

# Split dataset in Train and Validation Splits
n, split = len(dataset), int(np.floor(0.8 * len(dataset)))
train_idxs = np.random.choice(range(n), size=split, replace=False)
valid_idxs = [idx for idx in range(n) if idx not in train_idxs]
trainset = Subset(dataset, indices=train_idxs)
validset = Subset(dataset, indices=valid_idxs)

# Mini Batch Loaders (Shuffle Order for Training)
trainloader = DataLoader(trainset, batch_size=1_024, shuffle=True )
validloader = DataLoader(validset, batch_size=1_024, shuffle=False)
testloader  = DataLoader(testset,  batch_size=1_024, shuffle=False)
```

The first step consists in loading the MNIST dataset and applying preprocessing to the data for preparing the ingestion by the model. The images need to be transformed into a normalized tensor and flatten to form a 1-dimensional vector. The datasets are split into a training set, a validation set, and a test set. A mini-batch loader is then used to wrap the dataset and load multiple input and output pairs at the same time.

```
from torch.nn import (Linear, Module, ReLU, Sequential)
from torch.optim import (AdamW, Optimizer)

# Model and Optimizer
model = Sequential(
    Linear(28 * 28, 128), ReLU(),
    Linear(    128, 128), ReLU(),
    Linear(    128,  10),
)
optim = AdamW(model.parameters(), lr=1e-2)
```

Then, the model is defined as a sequence of three linear layers (linear transformations with a bias for the intercept) and ReLU activations except for the last one responsible for outputting the logits, used for computing the loss, here the cross entropy for multi-class classification. The enhanced SGD optimizer, Adam, is then initialized with the model's weight and a learning rate ϵ. AdamW is a variant of Adam with a corrected weight decay term for regularization.

```
from torch import Tensor
from torch.nn.functional import cross_entropy

# Perform one Step and estimate Metrics
def step(
    model: Module,
```

```
    optim: Optimizer,
    imgs: Tensor,
    labels: Tensor,
    split: str,
) -> Tuple[float, float]:
    logits = model(imgs)                    # Prediction
    loss = cross_entropy(logits, labels)    # Mean Loss
    n_correct = logits.argmax(dim=-1)       # Correct Predictions

    # Train if split is "train"
    if split == "train":
        loss.backward()
        optim.step()
        optim.zero_grad(set_to_none=True)

    return loss.item(), n_correct.item()
```

The `step` function is responsible for performing one training step when the given split is set to `"train"` and computes the metrics used for monitoring. In our case, we monitor the average loss and the accuracy of the model. For a more complete evaluation, other metrics such as the F-1 score, the perplexity, the recall, and a confusion matrix can be evaluated. They are here omitted for the sake of illustration and simplicity.

```
# Train for 10 epochs
for epoch in range(10):
    # Training
    model.train()
    loss, acc = 0, 0
    for imgs, labels in trainloader:
        metrics = step(model, optim, imgs, label, "train")
        loss += metrics[0] / len(trainloader)
        acc  += metrics[1] / len(trainloader.dataset)
    print(f"[Train] Epoch {epoch}, loss: {loss:.2e}, acc: {acc * 100:.2f}%")

    # Validation
    model.eval()
    with torch.inference_mode():
        loss, acc = 0, 0
        for imgs, labels in validloader:
            metrics = step(model, optim, imgs, label, "valid")
            loss += metrics[0] / len(validloader)
            acc  += metrics[1] / len(validloader.dataset)
    print(f"[Valid] Epoch {epoch}, loss: {loss:.2e}, acc: {acc * 100:.2f}%")

# Test
model.eval()
with torch.inference_mode():
    loss, acc = 0, 0
    for imgs, labels in testloader:
        metrics = step(model, optim, imgs, label, "test")
```

```
        loss += metrics[0] / len(testloader)
        acc  += metrics[1] / len(testloader.dataset)
print(f"[Test] loss: {loss:.2e}, acc: {acc * 100:.2f}%")
```

Finally, the model is trained for 10 epochs, the number of times the entire dataset is looped through. This number was arbitrarily chosen to correspond with the loss saturation when the model does not improve much. A training loop is divided into a few steps, a training phase where one continuously performs a training step followed by a validation step to monitor generalization, and when stopped, a test phase to monitor model generalization without bias. This last step prevents trying to overfit the validation set specifically and should be performed at the very end. An example of training history is shown in Fig 2.16. In this example, the model reaches 97.5 accuracy. By spending time tweaking the hyperparameters (the model's weights, the learning rate, the number of epochs, ...), the model can be improved further.

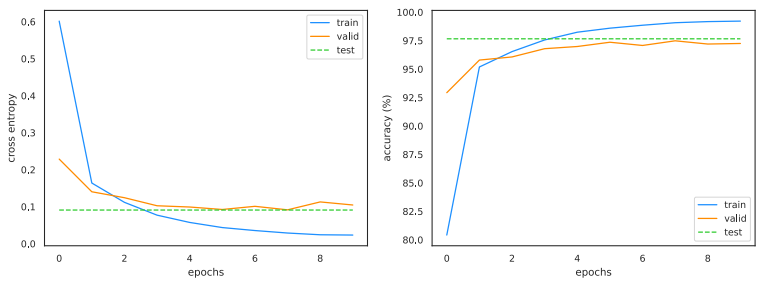

Figure 2.16: Training history of a 3-layer Multi-Layer Perceptron (MLP) with 128 neurons in every layer on the MNIST dataset. The average loss (cross-entropy) on the left, and the accuracy on the right are displayed for the training, validation, and test splits.

2.3.3 Convolutional Neural Network

While MLPs can be viewed as universal function approximators, they scale poorly with respect to high dimensional inputs such as images, videos, sound representations such as a spectrogram, volumetric data, and long sequences. For example, if we consider a small RGB image of size $256 \times 256 \times 3$, the input of a MLP would be a 1-dimensional vector of size $196,608$. The input layer of a MLP with 64 neurons would already mean that the network contains more than $12,582,912$ parameters. For this reason, researchers have created specialized NNs with biases in their architecture inspired by cognitive and biophysical mechanisms. Convolutional Neural Networks (CNNs) (ConvNets) are such a NN specialized in handling spatially correlated data such as images.

Convolution: The core component of a ConvNet is the convolution operation. A +CNN operates by convolving (rolling) a set of parametrized filters on the input. If we reconsider our $W_1 \times H_1 \times D_1 = 256 \times 256 \times 3$, convolving a single filter of size $F_W \times F_H \times D_1 = 3 \times 3 \times 3$ would require sliding the filter across the entire input image tensor and computing the dot product of the overlapping tensor chunk and the filter. This operation results in what is called an activation

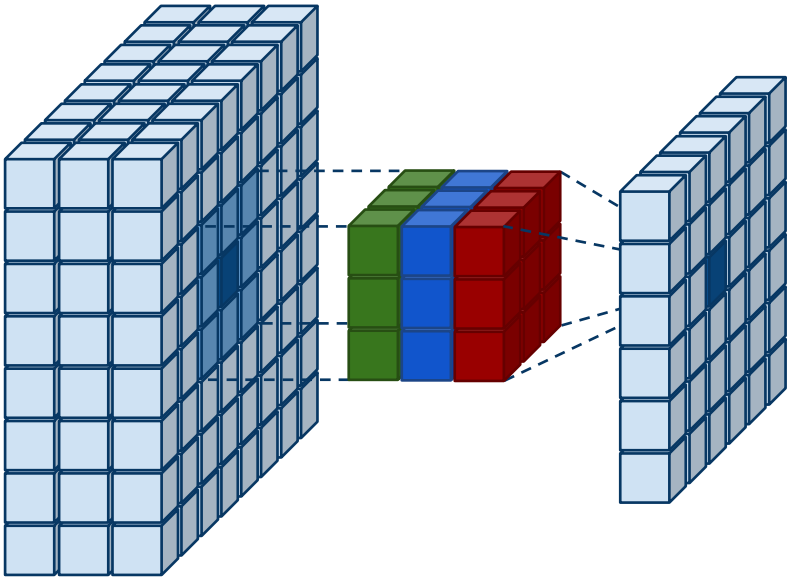

Figure 2.17: Illustration of a single $3 \times 3 \times 3$ filter convolution in the middle applied to a $8 \times 8 \times 3$ input tensor on the left. The result is a $6 \times 6 \times 1$ activation map on the right. The filter receptive field is drawn in dashed lines. This convolution is applied in valid model, no passing was applied to the input resulting in a lower resolution output tensor.

map, or feature map. The filter can be convolved in different configurations. The stride S defines the hop size when rolling the filter over the input, and the padding P defines the additional border added to the input tensor in order to parkour the input border (252 unique positions for the filter in the 256 image, 256 positions with a padding of 1 on each side of the input). A CNN convolves multiple parametrized filters K in a single convolution operation. Given a convolution setting, the operation requires $F_H \times F_W \times D_1 \times D_2$ parameters and outputs a feature map tensor of size $W_2 = (W_1 - F_W + 2P_W)/S + 1$, $H_2 = (H_1 - F_H + 2P_H)/S + 1$, and $D_2 = K$ (see Fig 2.17). The different filters are responsible for looking for the activation of different patterns in the input. The Convolution layer introduces the notion of weight sharing enabled by the sliding filter (neurons) and reduces computation by a large margin in comparison to a standard MLP layer.

Pooling: It is common to follow convolution layers by pooling layers to reduce the dimensionality when growing the ConvNet deeper. The pooling layer reduces its input by applying a reduction operation. The reduction operation can be taking the `max`, `min`, or `average`, of a rolling window. This operation does not involve any additional parameter and is applied channel-wise. If we consider a max-pooling operation with a 2×2 kernel and a stride of 2, the output becomes half the size of the input. It also has the benefit of making the CNN more robust to scale and translation. It is sometimes more strategic to make use of stride instead of adding pooling layers. It has the same benefit of reducing the feature map size while avoiding an additional operation.

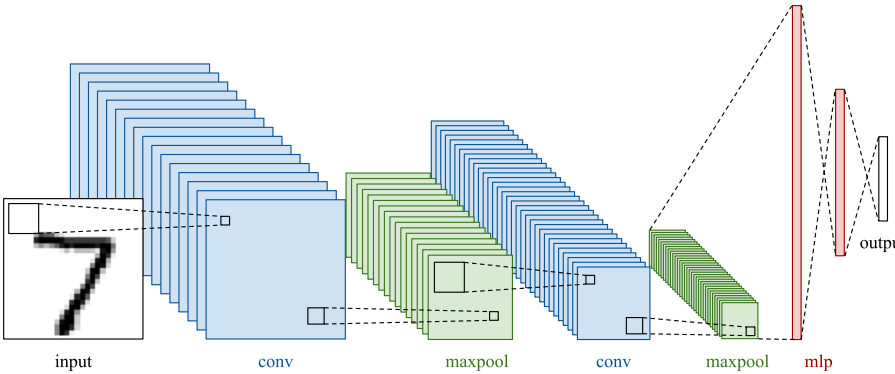

Figure 2.18: Illustration of a small Convolutional Neural Networks (CNNs) containing a convolution (conv) layer, a max-pooling (maxpool), and another convolution followed by another max-pooling. The last feature map is then flattened into a 1-dimensional vector and used as the input for the Multi-Layer Perceptron (MLP) classifier.

ConvNet: Finally, a CNN is assembled by stacking multiple convolution layers and pooling layers. When the feature maps are small enough, the final feature map is flattened and passed to an additional MLP in charge of the classification or regression. This combination of a parametric convolutional feature extractor and a MLP is what we call a ConvNet.

Figure 2.19: Visualization of VGG16's four first activation maps (feature maps). The input image is left and the activations are shown in order of the layers top to bottom and left to right. Credit https://images.all4ed.org/

Feature Maps: The feature maps learned by a CNN are hierarchical. In the first layers, the learned filters are focusing on simple features such as lines, diagonals, and arcs, and act as edge detectors. The deeper the layers are, the more complex the features are because they are resulting from a succession of combinations from previous activations (see Fig 2.19).

Finetuning: Training CNNs on bigger and more diverse datasets allows learning more general filters increasing the likelihood that the network will perform on out-of-domain data. In practice, CNNs are not often trained from scratch. Such a process requires the use of expensive dedicated hardware and hours of training. However, thanks to the open-source mindset of the DL field, big actors often share the weights of such models referred to as pretrained models, or foundation models [3].

Foundation models can be further refined through smaller training on smaller and specialized datasets containing few good-quality examples. This process is called finetuning and is less expensive and time-consuming than full training. One method for finetuning consists in removing the classification head of a pre-trained model such as VGG16 [58] and replacing it with a new one adapted to the number of classes required for the task. The pretrained weights are then frozen (not updated during training), and the new weights are trained following a standard supervised-learning procedure (see Lst 2.3.3).

```
from torchvision.models import (vgg16, VGG16_Weights)

# Import pretrained VGG16 model
model = vgg16(weights=VGG16_Weights.DEFAULT)

#Freeze pretrained features weights
for param in model.features.parameters():
    param.requires_grad = False

# Replace the classifier head
model.classifier = Sequential(
    Linear(512 * 7 * 7, 512), ReLU(),
    Linear(512,         512), ReLU(),
    Linear(512, num_classes),
)
```

MNIST Classifier: Let us reconsider the MNIST toy classification example and replace the MLP with a CNN. The model is divided in two sections, the feature extractor made out of two convolutional and max-pooling layers with 5×5 filters, the middle layer responsible for flattening the feature maps down to a 1-dimensional vector fed to the classifier head, a 3-layer MLP similar to the first one. The training procedure is left unchanged, the number of parameters is approximately similar, a little less for the CNN, and the number of epochs is the same. The input is however not flattened as the CNN consumes a full image tensor.

```
from collections import OrderedDict
from torch.nn import (Conv2d, Flatten, Linear, MaxPool2d, ReLU, Sequential)
from torchvision.transforms.functional import to_tensor

# Load MNIST images as Tensors and Normalize [0; 1]
T = lambda x: to_tensor(x).float()
...

# Model
```

```
model = Sequential(OrderedDict(
    features=Sequential(
        Conv2d(1,  6, 5), ReLU(), MaxPool2d(2),
        Conv2d(6, 16, 5), ReLU(), MaxPool2d(2),
    ),
    flatten=Flatten(),
    classifier=Sequential(
        Linear(256, 128), ReLU(),
        Linear(128,  64), ReLU(),
        Linear( 64,  10),
    ),
))
...
```

The CNN is able to achieve a 99 accuracy on the test set early during training (epoch 5). The CNN is a more robust, specialized, and thus more efficient architecture for handling images. The training history can be observed in Fig 2.20.

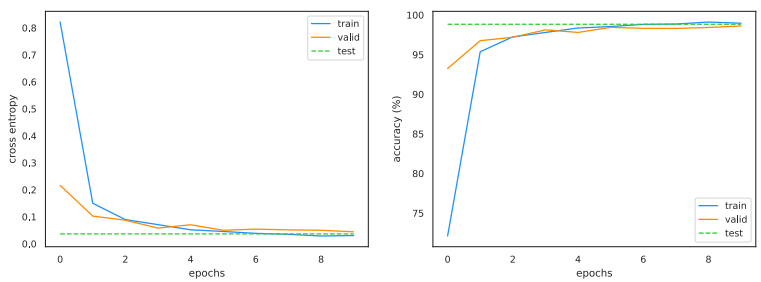

Figure 2.20: Training history of a Convolutional Neural Network (CNN) made out of a sequence of two convolutions followed by max-pooling and a 3-layer Multi-Layer Perceptron (MLP) classifier on the MNIST dataset. The average loss (cross-entropy) on the left, and the accuracy on the right are displayed for the training, validation, and test splits.

2.4 Generative Architectures

In this section, we extend our Deep Learning (DL) architecture toolbox with generative AI architectures such as the Autoencoder (AE) (see Sec 2.4.1), the Variational Autoencoder (VAE) (see Sec 2.4.2), the Generative Adversarial Network (GAN) (see Sec 2.4.3), and the Denoising Diffusion Model (DDM) (see Sec 2.4.4) with a strong focus on image generation. Similarly to the previous sections, the Modified National Institute of Standards and Technology (MNIST) dataset is used for illustrative purposes.

This section does not only discuss the technical details of those architectures but also compares them on three criteria, generation inference speed, generation variance, and generation quality and complexity.

2.4.1 Autoencoders

Autoencoders (AEs) are part of a family of feedforward NNs for which the input tensor is the same as the output tensor. They encode (compress), the input into a low-dimensional code in a latent space, and then decode (reconstruct) the original input from this compressed representation (see Fig 2.21). An AE is built using two network parts, an encoder E, NN that reduces the input dimension, a decoder D that recovers the input x from the reduced tensor z, and a reconstruction objective. This architecture can be viewed as a dimensionality reduction technique but can be used as a generative model. By feeding the decoder D with arbitrary latent codes z, one can generate unseen data points $\hat{x}$ similar to the training distribution by interpolation. Additional training objectives can be used to disentangle the latent representation so that the data points are organized mindfully in the latent space, semantically for example.

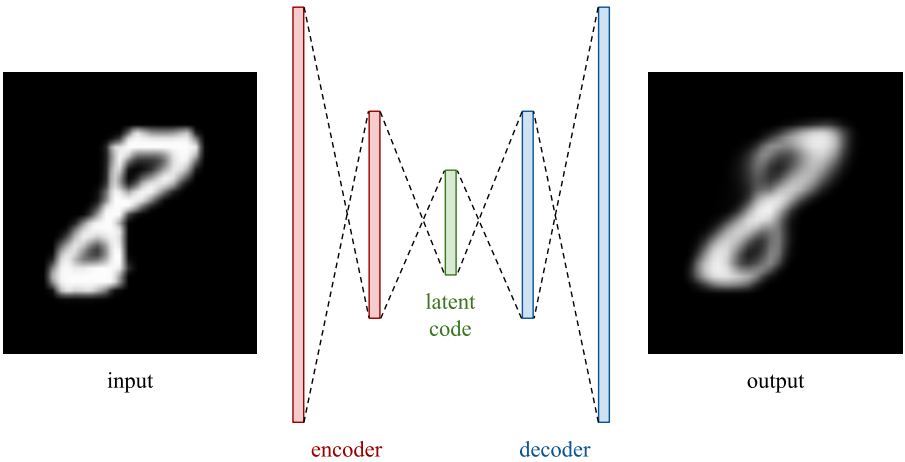

Figure 2.21: Autoencoder (AE) architecture.

Properties: Compared to a traditional compression method, AEs are tied to their training data. They are trained to learn data-specific features useful for in-domain compression not for out-of-domain. An AE trained on MNIST cannot be used for compressing photos of faces. Such architecture cannot be considered a lossless compression algorithm. The reconstruction is most of the time degraded. One strong advantage of using an AE is that they do not require complex data preparation. They are part of the unsupervised training family, where labeled data is not needed for training, and in this case, self-supervised learning where the target output is built synthetically from the input.

MNIST Digit Image Generation: Let us consider MNIST and train a small Autoencoder (AE) to compress handwritten digits to 32 latent codes. Our AE is made out of a small MLP encoder and decoder both with two inner layers with a hidden dimension of 128.

```
from collections import OrderedDict
from torch.nn import (Linear, ReLU, Sequential, Sigmoid)
```

```
# Model definition
model = Sequential(OrderedDict(
    encoder=Sequential(
        Linear(28 * 28, 128), ReLU(),
        Linear(    128,  32), ReLU(),
    ),
    decoder=Sequential(
        Linear( 32,     128), ReLU(),
        Linear(128, 28 * 28), Sigmoid(),
    ),
))
```

The model is trained on 10 epochs with no hyperparameter tuning using the `binar_cross_entropy` objective function as the dataset contains black and white images normalized in the $[0; 1]$ range.

```
from torch.nn.functional import binary_cross_entropy

# Compute loss
loss = binary_cross_entropy(model(x), x)
```

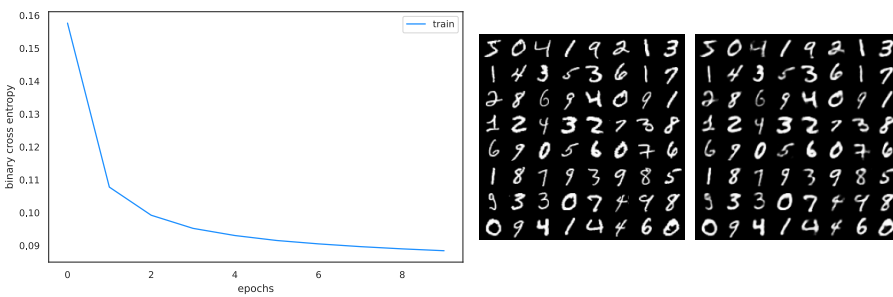

Figure 2.22: Training history of a small 2-layer Autoencoder (AE). The binary cross entropy loss is shown on the left, a training sample in the middle, and its corresponding reconstruction on the right.

The result of the training can be observed in Fig 2.22. Despite little degradation, our model can reconstruct the handwritten digits from their latent code. The degradation is minimized by the fact that we are dealing with a toy dataset. The phenomenon can be observed by reducing the number of parameters of the network or the size of the latent space. To reconstruct the images, we first need to get a latent code, either by encoding an existing image, or by randomly initializing a latent vector in a reasonable range, and providing it to the decoder as shown below.

```
# Generate sample given latent-code
x_ = model.decoder(z)
```

The latent space can be observed in Fig 2.23 after being reduced to a 2-dimensional proxy space using Uniform Manifold Approximation and Projection (UMAP) [41] for visualization purposes. Our latent space is not organized in a way that we can visually distinguish between the digit classes.

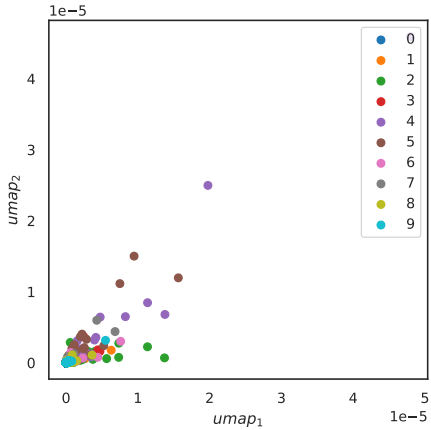

Figure 2.23: Trained Autoencoder (AE) latent space visualization. The latent code, originally a 32-dimensional tensor, is reduced to a 2-dimensional space for visualization purposes. The data points represent the encoded latent code of images from the MNIST dataset and are colored based on their corresponding label (digit). The latent space is not organized in a way that allows us to visually separate these classes.

2.4.2 Variational Autoencoders

Due to a lack of latent space regularization as shown in the previous sub-section, AE cannot be used without any hacking to generate, or produce unseen samples. A vanilla AE does not encode any structure on the latent space. It is trained only for reconstruction and is thus subject to high overfitting resulting in a meaningless structural organization of the latent codes. The Variational Autoencoder (VAE) architecture [30] is one answer to this issue. It can be viewed as a special AE hacked by adding a regularization objective enabling generation by exploring the learned and structured latent space.

Regularization: VAEs are topologically similar to AE. They possess an encoder to compress the input into a latent code, and a decoder to reconstruct the signal from it. However, instead of encoding the input as a single point, it encodes it as a distribution in the latent space. In practice, the distribution used is chosen to be close to a normal distribution. The encoder is changed to output the parameters of this distribution, the mean μ, and the variance σ^2. σ^2 is often replaced by a proxy $\rho = log(\sigma^2)$ to enforce positivity and stability. The new inference scheme is changed for $\hat{x} = D(z)$, where the latent code $z \sim \mathcal{N}(E(x)_\mu, exp(E(x)_\rho))$.

Probabilistic Formulation: Let us consider the VAE as a probabilistic model. x, our data, is generated from the latent variable z that cannot be observed. In this framework, the generation steps are the following: z is sampled from the prior distribution $p(z)$, and x is sampled from the conditional likelihood $x \sim p(x|z)$. In this setting, the probabilistic decoder is $p(x|z)$, and the probabilistic encoder is $p(z|x)$. The Bayes theorem allows expressing a natural relation between the prior $p(z)$, the likelihood $p(x|z)$, and the posterior $p(z|x)$:

$$p(z|x) = \frac{p(x|z)p(z)}{p(x)} = \frac{p(x|z)p(z)}{\int p(x|u)p(u)du} \tag{2.6}$$

A standard Gaussian distribution is often assumed for the prior $p(z)$, and a parametric Gaussian for the likelihood $p(x|z)$ with its mean being defined by a deterministic function $f \in F$ and a positive constant $c \cdot I$ for the covariance. In this setting:

$$\begin{aligned} p(z) &\sim \mathcal{N}(0, I) \\ p(x|z) &\sim \mathcal{N}(f(z), cI),\ f \in F,\ c > 0 \end{aligned} \tag{2.7}$$

These equations (see Eqns 2.6, 2.7) define a classical Bayesian Inference problem. This problem is however intractable because of the denominator's integral $\int p(x|u)p(u)du$ and thus requires the use of approximation techniques.

Variational Inference: In statistics, Variational Inference (VI) is one of the techniques used to approximate complex distributions. It consists in setting a parametrized distribution family, in our case Gaussians with its mean and covariance, and searching for the best approximation of the target distribution in this family. To search for the best candidate, we use the Kullback-Leibler Divergence (KL-Divergence) between the approximation and the target and minimize it with Gradient Descent (GD).

Let us approximate the posterior $p(z|x)$ using VI with a Gaussian distribution $q_x(z)$ with a mean $g(x) \in G$ and covariance $h(x) \in H$ where $q_x(z) \sim \mathcal{N}(g(x), h(x))$. we can now look for the optimal g^* and h^* minimizing the KL-Divergence between the target and the approximation:

$$\begin{aligned} (g^*, h^*) &= \underset{(g,h)\in G\times H}{arg\ min}\ KL(q_x(z)||p(z|x)) \\ &= \underset{(g,h)\in G\times H}{arg\ min}\ (E_{z\sim q_x}\ log\ q_x(z) - E_{z\sim q_x}\ log\ \frac{p(x|z)p(z)}{p(x)}) \\ &= \underset{(g,h)\in G\times H}{arg\ min}\ (E_z\ log\ q_x(z) - E_z\ log\ p(z) - E_z\ log\ p(x|z) + E_z\ log\ p(x)) \\ &= \underset{(g,h)\in G\times H}{arg\ min}\ (E_z[log\ p(x|z) - KL(q_x(z)||p(z)]) \\ &= \underset{(g,h)\in G\times H}{arg\ min}\ (E_z\ log\ p(x|z) - KL(q_x(z)||p(z)) \\ &= \underset{(g,h)\in G\times H}{arg\ min}\ (E_z[-\frac{||x-f(z)||^2}{2c}] - KL(q_x(z)||p(z)) \end{aligned} \tag{2.8}$$

This rewrite of the objective equations demonstrates a natural tradeoff between the data confidence $E_z[-\frac{||x-f(z)||^2}{2c}]$ and the prior confidence $KL(q_x(z)||p(z))$. The first term describes a reconstruction loss where the decoder parametrized by the function $f \in F$ has to recover the input x from the latent code z, and the second term a regularization objective between $q_x(z)$ and the prior $p(z)$ which is

gaussian. We can view the constant c as a strength parameter that can adjust how we favor the regularization.

Reparametrization Trick: The VAE architecture is trained to find the parameters of the functions f, g, and h by minimizing the VI objective (see Eq 2.8). The encoder is charged to output two vectors, one for representing $g(x)$ the mean, in the case of a Gaussian distribution μ, and the other representing the variance of the distribution $h(x)$, $\rho = log(\sigma^2)$. The latent code z is then sampled from the Gaussian distribution $z \sim \mathcal{N}(\mu, \sigma)$ and finally decoded to reconstruct the original input x.

There is however a catch. The sampling process is stochastic and thus not differentiable. And we know that a NN needs to be differentiable to be optimized using SGD. To solve this problem, Kingma et al. [30] propose to use what they call a reparametrization trick. It consists in sampling a surrogate standard Gaussian distribution $\zeta \sim \mathcal{N}(0, I)$ and scaling it by the output of the learned encoder μ and σ^2. This the process becomes:

$$\begin{aligned} E(x) &= (\mu, \rho) \\ \hat{x} &= D(\mu + \zeta \; exp(\rho)), \; \zeta \sim N(O, I) \end{aligned} \tag{2.9}$$

Performing the latent sampling using the reparametrization trick (see Eq 2.9) conserves the gradient flow. The VAE can thus be trained to learn a structured latent space that can be used to interpolate latent codes and decode them into samples similar to the training distribution.

MNIST Digit Image Generation: ...

2.4.3 Generative Adversarial Networks

Core Concepts

Latent Space: ...

MNIST Digit Image Generation: ...

2.4.4 Denoising Diffusion Models

Core Concepts

Latent Space: ...

MNIST Digit Image Generation: ...

2.5 Attention Machanism

2.5.1 Multihead Self-Attention

2.5.2 Large Language Models

Chapter 3

Methodology

3.1 Implementation

3.2 Objective Evaluation

3.3 Subjective Evaluation

3.4 Reproducibility

Part II

Core

Chapter 4

Contrib I (Find Catchy Explicit Name)

4.1 State of the Art

4.2 Method

4.3 Setup

4.4 Results

4.5 Summary

Chapter 5

Contrib II (Find Catchy Explicit Name)

5.1 State of the Art

5.2 Method

5.3 Setup

5.4 Results

5.5 Summary

Chapter 6

Contrib III (Find Catchy Explicit Name)

6.1 State of the Art

6.2 Method

6.3 Setup

6.4 Results

6.5 Summary

Chapter 7

Contrib IV (Find Catchy Explicit Name)

7.1 State of the Art

7.2 Method

7.3 Setup

7.4 Results

7.5 Summary

Part III

Reflection

Chapter 8

Ethical and Societal Impact

Chapter 9

Conclusion

References

[1] Abadi, M., Barham, P., Chen, J., Chen, Z., Davis, A., Dean, J., Devin, M., Ghemawat, S., Irving, G., Isard, M., et al. 2016. Tensorflow: A system for large-scale machine learning. *Osdi* (2016), 265–283.

[2] Anderson, J.R. 1992. Automaticity and the ACT theory. *The American Journal of Psychology*. 105, 2 (1992), 165–180. DOI:https://doi.org/10.2307/1423026[1].

[3] Bommasani, R., Hudson, D.A., Adeli, E., Altman, R., Arora, S., Arx, S. von, Bernstein, M.S., Bohg, J., Bosselut, A., Brunskill, E., et al. 2021. On the opportunities and risks of foundation models. *arXiv preprint arXiv:2108.07258*. (2021).

[4] Brown, T., Mann, B., Ryder, N., Subbiah, M., Kaplan, J.D., Dhariwal, P., Neelakantan, A., Shyam, P., Sastry, G., Askell, A., et al. 2020. Language models are few-shot learners. *Advances in neural information processing systems*. 33, (2020), 1877–1901.

[5] CHATGPT: Optimizing language models for dialogue: 2023. *https://openai.com/blog/chatgpt/*[2]. Accessed: 2023-01-26.

[6] Ci, Y., Ma, X., Wang, Z., Li, H. and Luo, Z. 2018. User-guided deep anime line art colorization with conditional adversarial networks[3]. *Proceedings of the 26th ACM international conference on multimedia* (New York, NY, USA, 2018), 1536–1544.

[7] Clip studio PAINT: *https://www.clipstudio.net/*[4]. Accessed: 2023-01-26.

[8] Cortes, C. and Vapnik, V. 1995. Support-vector networks. *Machine Learning*. 20, 3 (Sep. 1995), 273–297. DOI:https://doi.org/10.1007/BF00994018[5].

[9] Deng, J., Dong, W., Socher, R., Li, L.-J., Li, K. and Fei-Fei, L. 2009. ImageNet: A large-scale hierarchical image database[6]. *2009 IEEE conference on computer vision and pattern recognition* (2009), 248–255.

[10] Duchi, J., Hazan, E. and Singer, Y. 2011. Adaptive subgradient methods for online learning and stochastic optimization[7]. *Journal of Machine Learning Research*. 12, 61 (2011), 2121–2159.

[11] Frans, K. 2017. Outline colorization through tandem adversarial networks[8]. *CoRR*. abs/1704.08834, (2017).

[12] Fukushima, K. 1980. Neocognitron: A self-organizing neural network model for a mechanism of pattern recognition unaffected by shift in position. *Biological Cybernetics*. 36, 4 (Apr. 1980), 193–202. DOI:https://doi.org/10.1007/BF00344251[9].

[13] Furusawa, C., Hiroshiba, K., Ogaki, K. and Odagiri, Y. 2017. Comicolorization: Semi-automatic manga colorization[10]. *SIGGRAPH asia 2017 technical briefs* (New York, NY, USA, 2017).

[14] Goodfellow, I.J., Bengio, Y. and Courville, A. 2016. *Deep learning*. MIT Press.

[15] Goodfellow, I., Pouget-Abadie, J., Mirza, M., Xu, B., Warde-Farley, D., Ozair, S., Courville, A. and Bengio, Y. 2014. Generative adversarial nets[11]. *Advances in neural information processing systems* (2014).

[16] Hati, Y., Jouet, G., Rousseaux, F. and Duhart, C. 2019. PaintsTorch: A user-guided anime line art colorization tool with double generator conditional adversarial network[12]. *European conference on visual media production* (New York, NY, USA, 2019).

[17] Hati, Y., Thevenin, V., Nolot, F., Rousseaux, F. and Duhart, C. 2023. StencilTorch: An iterative and user-guided framework for anime lineart colorization. *Image and vision computing* (Cham, 2023), 1–17.

[18] He, K., Zhang, X., Ren, S. and Sun, J. 2016. Deep residual learning for image recognition. *Proceedings of the IEEE conference on computer vision and pattern recognition* (2016), 770–778.

[19] Hensman, P. and Aizawa, K. 2017. cGAN-based manga colorization using a single training image[13]. *2017 14th IAPR international conference on document analysis and recognition (ICDAR)* (Los Alamitos, CA, USA, Nov. 2017), 72–77.

[20] Hinton, G., Srivastava, N. and Swersky, K. Neural networks for machine learning: Overview of mini-batch gradient descent.

[21] Ho, J., Jain, A. and Abbeel, P. 2020. Denoising diffusion probabilistic models. *Advances in Neural Information Processing Systems*. 33, (2020), 6840–6851.

[22] Hochreiter, S. and Schmidhuber, J. 1997. Long Short-Term Memory. *Neural Computation*. 9, 8 (Nov. 1997), 1735–1780. DOI:https://doi.org/10.1162/neco.1997.9.8.1735[14].

[23] Hornik, K., Stinchcombe, M. and White, H. 1989. Multilayer feedforward networks are universal approximators. *Neural Networks*. 2, 5 (1989), 359–366. DOI:https://doi.org/10.1016/0893-6080(89)90020-8[15].

[24] Jackson, P. 1998. *Introduction to expert systems*. Addison-Wesley Longman Publishing Co., Inc.

[25] Jumper, J. et al. 2021. Highly accurate protein structure prediction with AlphaFold. *Nature*. 596, 7873 (Aug. 2021), 583–589. DOI:https://doi.org/10.1038/s41586-021-03819-2[16].

[26] Kandinsky, W. and Sadleir, M. 1977. *Concerning the spiritual in art*. Dover Publications.

[27] Karpathy, A. 2020. Micrograd[17].

[28] Kim, H., Jhoo, H.Y., Park, E. and Yoo, S. 2019. Tag2Pix: Line art colorization using text tag with SECat and changing loss[18]. *2019 IEEE/CVF international conference on computer vision (ICCV)* (2019), 9055–9064.

[29] Kingma, D.P. and Ba, J. 2014. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*. (2014).

[30] Kingma, D.P. and Welling, M. 2013. Auto-encoding variational bayes. *arXiv preprint arXiv:1312.6114*. (2013).

[31] Krizhevsky, A., Sutskever, I. and Hinton, G.E. 2012. ImageNet classification with deep convolutional neural networks[19]. *Advances in neural information processing systems* (2012).

[32] Laird, J.E. 2019. *The soar cognitive architecture*. The MIT Press.

[33] Le Cun, Y. 2019. *Quand la machine apprend: La révolution des neurones artificiels et de l'apprentissage profond*[20]. Odile Jacob.

[34] LeCun, Y., Boser, B., Denker, J.S., Henderson, D., Howard, R.E., Hubbard, W. and Jackel, L.D. 1989. Backpropagation Applied to Handwritten Zip Code Recognition. *Neural Computation*. 1, 4 (Dec. 1989), 541–551. DOI:https://doi.org/10.1162/neco.1989.1.4.541[21].

[35] Lecun, Y., Bottou, L., Bengio, Y. and Haffner, P. 1998. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*. 86, 11 (1998), 2278–2324. DOI:https://doi.org/10.1109/5.726791[22].

[36] Lieto, A. 2021. *Cognitive design for artificial minds (1st ed.)*[23]. Routledge.

[37] Liu, Y., Qin, Z., Wan, T. and Luo, Z. 2018. Auto-painter: Cartoon image generation from sketch by using conditional wasserstein generative adversarial networks. *Neurocomputing*. 311, (2018), 78–87. DOI:https://doi.org/10.1016/j.neucom.2018.05.045[24].

[38] Loshchilov, I. and Hutter, F. 2017. Decoupled weight decay regularization. *arXiv preprint arXiv:1711.05101*. (2017).

[39] McCarthy, J. 1978. History of LISP[25]. *History of programming languages*. Association for Computing Machinery. 173–185.

[40] McCarthy, J., Minsky, M.L., Rochester, N. and Shannon, C.E. 2006. A proposal for the dartmouth summer research project on artificial intelligence, august 31, 1955. *AI Magazine*. 27, 4 (2006), 12. DOI:https://doi.org/10.1609/aimag.v27i4.1904[26].

[41] McInnes, L., Healy, J. and Melville, J. 2018. Umap: Uniform manifold approximation and projection for dimension reduction. *arXiv preprint arXiv:1802.03426*. (2018).

[42] Minsky, M. and Papert, S. 1969. *Perceptrons: An introduction to computational geometry*. MIT Press.

[43] Mumford, M., Medeiros, K. and Partlow, P. 2012. Creative thinking: Processes, strategies, and knowledge. *The Journal of Creative Behavior*. 46, (Mar. 2012). DOI:https://doi.org/10.1002/jocb.003[27].

[44] Newell, A., Shaw, J.C. and Simon, H.A. 1959. *The processes of creative thinking*[28]. RAND Corporation.

[45] Paintman: *http://www.retasstudio.net/products/paintman/*[29]. Accessed: 2023-01-26.

[46] Paszke, A. et al. 2019. PyTorch: An imperative style, high-performance deep learning library. *Proceedings of the 33rd international conference on neural information processing systems*. Curran Associates Inc.

[47] Pelica paint: 2017. *https://petalica-paint.pixiv.dev/index_en.html*[30]. Accessed: 2023-01-26.

[48] Photoshop: *https://www.adobe.com/products/photoshop.html*[31]. Accessed: 2023-01-26.

[49] Qian, N. 1999. On the momentum term in gradient descent learning algorithms. *Neural Networks*. 12, 1 (1999), 145–151. DOI:https://doi.org/10.1016/S0893-6080(98)00116-6[32].

[50] Rombach, R., Blattmann, A., Lorenz, D., Esser, P. and Ommer, B. 2021. High-resolution image synthesis with latent diffusion models[33].

[51] Rosenblatt, F. 1958. The perceptron: A probabilistic model for information storage and organization in the brain. *Psychological Review*. 65, 6 (1958), 386–408. DOI:https://doi.org/10.1037/h0042519[34].

[52] Rumelhart, D.E., Hinton, G.E. and Williams, R.J. 1986. Learning representations by back-propagating errors. *Nature*. 323, 6088 (Oct. 1986), 533–536. DOI:https://doi.org/10.1038/323533a0[35].

[53] Russell, S.J. and Norvig, P. 2009. *Artificial intelligence: A modern approach*. Pearson.

[54] Saito, M. and Matsui, Y. 2015. Illustration2Vec: A semantic vector representation of illustrations[36]. *SIGGRAPH asia 2015 technical briefs* (New York, NY, USA, 2015), 5:1–5:4.

[55] Sangkloy, P., Lu, J., Fang, C., Yu, F. and Hays, J. 2017. Scribbler: Controlling deep image synthesis with sketch and color[37]. *2017 IEEE conference on computer vision and pattern recognition, CVPR 2017, honolulu, HI, USA, july 21-26, 2017* (2017), 6836–6845.

[56] Senior, A.W. et al. 2020. Improved protein structure prediction using potentials from deep learning. *Nature*. 577, 7792 (Jan. 2020), 706–710. DOI:https://doi.org/10.1038/s41586-019-1923-7[38].

[57] Silver, D. et al. 2016. Mastering the game of go with deep neural networks and tree search. *Nature*. 529, 7587 (Jan. 2016), 484–489. DOI:https://doi.org/10.1038/nature16961[39].

[58] Simonyan, K. and Zisserman, A. 2014. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556*. (2014).

[59] The MNIST database: *http://yann.lecun.com/exdb/mnist/*[40]. Accessed: 2023-02-07.

[60] Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A.N., Kaiser, Ł. and Polosukhin, I. 2017. Attention is all you need. *Advances in neural information processing systems*. 30, (2017).

[61] Vinyals, O. et al. 2019. Grandmaster level in StarCraft II using multi-agent reinforcement learning. *Nature*. 575, 7782 (Nov. 2019), 350–354. DOI:https://doi.org/10.1038/s41586-019-1724-z[41].

[62] Wolf, M.J., Miller, K. and Grodzinsky, F.S. 2017. Why we should have seen that coming: Comments on microsoft's tay "experiment," and wider implications. *SIGCAS Comput. Soc.* 47, 3 (Sep. 2017), 54–64. DOI:https://doi.org/10.1145/3144592.3144598[42].

[63] Zhang, A., Lipton, Z.C., Li, M. and Smola, A.J. 2021. Dive into deep learning. *arXiv preprint arXiv:2106.11342*. (2021).

[64] Zhang, L., Ji, Y., Lin, X. and Liu, C. 2017. Style transfer for anime sketches with enhanced residual u-net and auxiliary classifier GAN[43]. *2017 4th IAPR asian conference on pattern recognition (ACPR)* (2017), 506–511.

[65] Zhang, R., Zhu, J.-Y., Isola, P., Geng, X., Lin, A.S., Yu, T. and Efros, A.A. 2017. Real-time user-guided image colorization with learned deep priors. *ACM Trans. Graph.* 36, 4 (Jul. 2017). DOI:https://doi.org/10.1145/3072959.3073703[44].