Predicting Frictional Properties of Graphene Kirigami Using Molecular Dynamics and Neural Networks

Designs for a negative friction coefficient

Mikkel Metzsch Jensen

Thesis submitted for the degree of
Master in Computational Science: Materials Science
60 credits

Department of Physics
Faculty of mathematics and natural sciences

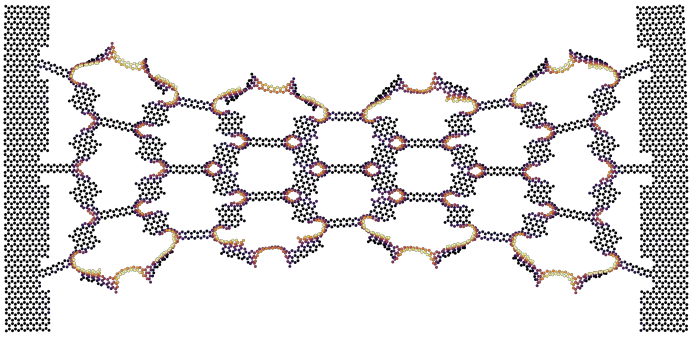
UNIVERSITY OF OSLO

Spring 2023

Predicting Frictional Properties of Graphene Kirigami Using Molecular Dynamics and Neural Networks

Designs for a negative friction coefficient

Mikkel Metzsch Jensen



Predicting Frictional Properties of Graphene Kirigami Using Molecular Dynamics and Neural Networks

http://www.duo.uio.no/

Printed: Reprosentralen, University of Oslo

Abstract

Various theoretical models and experimental results propose different governing mechanisms for friction at the nanoscale. We consider a graphene sheet modified with Kirigami-inspired cuts and under the influence of strain. Prior research has demonstrated that this system exhibits out-of-plane buckling, which could result in a decrease in contact area when sliding on a substrate. According to asperity theory, this decrease in contact area is expected to lead to a reduction of friction. However, to the best of our knowledge, no previous studies have investigated the friction behavior of a nanoscale Kirigami graphene sheet under strain. Here we show that specific Kirigami designs yield a non-linear dependency between kinetic friction and the strain of the sheet. Using molecular dynamics simulations, we have found a non-monotonic increase in friction with strain. We found that the friction-strain relationship does not show any clear dependency on contact area which contradicts asperity theory. Our findings suggest that the effect is associated with the out-of-plane buckling of the graphene sheet and we attribute this to a commensurability effect. By mimicking a load-strain coupling through tension, we were able to utilize this effect to demonstrate a negative friction coefficient on the order of -0.3 for loads in the range of a few nN. In addition, we have attempted to use machine learning to capture the relationship between Kirigami designs, load, and strain, with the objective of performing an accelerated search for new designs. Although this approach yielded some promising results, we conclude that further improvements to the dataset are necessary in order to develop a reliable model. We anticipate our findings to be a starting point for further investigations of the underlying mechanism for the frictional behavior of a Kirigami sheet. For instance, the commensurability hypothesis could be examined by varying the sliding angle in simulations. We propose to use an active learning strategy to extend the dataset for the use of machine learning to assist these investigations. If successful, further studies can be done on the method of inverse design. In summary, our findings suggest that the application of nanoscale Kirigami can be promising for developing novel friction-control strategies.

Acknowledgments

The task of writing a master's thesis is a demanding and extensive project which I could not have done without the support of many good people around me. First of all, I want to thank my supervisors Henrik Andersen Sveinsson and Anders Malthe-Sørenssen for the assistance in this thesis work. I am especially grateful for the weekly meetings with Henrik and the inspiring discussions had as we unraveled the discoveries related to the topic of this thesis. I remember that I initially asked for an estimate of how much time he had available for supervision and the answer was something along the lines of "There are no limits really, just send me an email and we figure it out". This attitude captures the main experience I have had working with Henrik and I am profoundly grateful for the time and effort he has put into this project. I hope that he did not regret this initial statement too much, because I have certainly been taken advantage of it. I also want to thank Even Marius Nordhagen for technical support regarding the use of the computational cluster. In that context, I also want to acknowledge the Center for Computing in Science Education (CCSE) for making these resources available.

I would like to express my gratitude to all the parties involved in making it possible for me to write my thesis from Italy. I am particularly grateful for the flexibility shown by my supervisors and for the support of Anders Kvellestad, who allowed me to work remotely as a group teacher. I would also like to thank Scuola Normale Superiore for providing me with access to their library.

I realize that it is a commonly used cliché to express gratitude for the support of loved ones. However, I want to highlight the exceptional role played by my fiancé, Ida, who deserves the main credit for enabling me to maintain a healthy state of mind. She has provided me with a solid foundation for a fulfilling life that enables me to pursue secondary objectives, such as an academic career. I look forward to spending the rest of my life with you.

In this thesis, I have used the formal pronoun "we" mainly as a customary habit related to the formalities of scientific writing in a team. Nonetheless, I have realized that this usage is more fitting as I have not been working alone on this project. I have received support all the way from colleagues and friends at the University of Oslo, my family residing in Denmark, and my life partner who slept beside me every night here in Italy. They are the "good people around me" who have made this thesis possible.

Acronyms

CNN Convolutional Neural Network. 12, 13

EMA Exponetial Moving Average. 10

GA Genetic Algorithm. 19

GAN Generative Adversarial Networks. 2

MD Molecular Dynamics. 1, 2, 3, 4, 7

ML Machine Learning. 2

MSE Mean Squared Error. 8

RMSProp Root Mean Square Propagation. 10, 12

SGD Stochastic Gradient Descent. 9

Contents

Chapter 1

Introduction

1.1 Motivation

Friction is the force that prevents the relative motion of objects in contact. In our everyday life, we recognize it as the inherent resistance to sliding motion. Some surfaces appear slippery and some appear rough, and we know intuitively that sliding down a snow-covered hill is much more exciting than its grassy counterpart. Without friction, it would not be possible to walk across a flat surface, lean against the wall without falling over or secure an object by the use of nails or screws [1, p. 5]. It is probably safe to say that the concept of friction is integrated into our everyday life to such an extent that most people take it for granted. However, the efforts to control friction date back to the early civilization (3500 B.C.) with the use of the wheel and lubricants to reduce friction in translational motion [2]. Today, friction is considered a part of the wider field *tribology* derived from the Greek word *tribos* meaning "rubbing". It includes the science of friction, wear and lubrication [2]. The most compelling motivation to study tribology is ultimately to gain full control of friction and wear for various technical applications. Especially, the reduction of friction is of great interest since this can be utilized to improve energy efficiency in mechanical systems with moving parts. Hence, it has been reported that tribological problems have a significant potential for both economic and environmental improvements [3]:

> "On global scale, these savings would amount to 1.4% of the GDP annually and 8.7% of the total energy consumption in the long term." [4].

On the other hand, the reduction of friction is not the only sensible application for tribological studies. Controlling frictional properties, besides minimization, might be of interest in the development of a grasping robot where finetuned object handling is required. While achieving a certain "constant" friction response is readily obtained through appropriate material choices, we are yet to unlock the full capabilities to alter friction dynamically on the go. One example from nature inspiring us to think along these lines is the gecko feet. More precisely, the Tokay gecko has received a lot of attention in scientific studies aiming to unravel the underlying mechanism of its "togglable" adhesion properties. Although the gecko can produce large adhesive forces, it retains the ability to remove its feet from an attachment surface at will [5]. This makes the gecko able to achieve a high adhesion on the feet when climbing a vertical surface while lifting them for the next step remains relatively effortless. For a grasping robot, we might consider an analog frictional concept of a surface material that can change from slippery to rough on demand depending on specific tasks; slippery and smooth when interacting with people and rough and firmly gripping when moving heavy objects.

In recent years an increasing amount of interest has gone into the studies of the microscopic origins of friction, due to the increased possibilities in surface preparation and the development of nanoscale experimental methods. Nano-friction is also of great concern for the field of nano-machining where the frictional properties between the tool and the workpiece dictate machining characteristics [3]. With concurrent progress in computational capacity and development of Molecular Dynamics (MD), numerical investigations serve as an invaluable tool for getting insight into the nanoscale mechanics associated with friction. This simulation-based approach can be considered as a "numerical experiment" enabling us to create and probe a variety of high-complexity systems which are still out of reach for modern experimental methods.

In materials science such MD-based numerical studies have been used to explore the concept of so-called *metamaterials* where the material compositions are designed meticulously to enhance certain physical properties [6–11]. This is often achieved either by intertwining different material types or removing certain regions completely. In recent papers by Hanakata et al. [6, 7], numerical studies have showcased that the mechanical properties of a graphene sheet, yield stress and yield strain, can be altered through the introduction of so-called *Kirigami* inspired cuts into the sheet. Kirigami is a variation of origami where the paper is cut additionally to being folded. While these methods originate as an art form, aiming to produce various artistic objects, they have proven to be applicable in a wide range of fields such as optics, physics, biology, chemistry and engineering [12]. Various forms of stimuli enable direct 2D to 3D transformations through the folding, bending, and twisting of microstructures. While original human designs have contributed to specific scientific applications in the past, the future of this field is highly driven by the question of how to generate new designs optimized for certain physical properties. However, the complexity of such systems and the associated design space makes for seemingly intractable[1] problems ruling out analytic solutions.

Earlier design approaches such as bioinspiration, looking at gecko feet for instance, and Edisonian, based on trial and error, generally rely on prior knowledge and an experienced designer [9]. While the Edisonian approach is certainly more feasible through numerical studies than real-world experiments, the number of combinations in the design space rather quickly becomes too large for a systematic search, even when considering the computation time on modern-day hardware. However, this computational time constraint can be relaxed by the use of machine learning (ML) which has been proven successful in the establishment of a mapping from the design space to physical properties of interest. This gives rise to two new styles of design approaches: One, by utilizing the prediction from a trained network we can skip the MD simulations altogether resulting in an *accelerated search* of designs. This can be further improved by guiding the search according to the most promising candidates. For instance, as done with the *genetic algorithm* based on mutation and crossing. Another more sophisticated approach is through generative methods such as *Generative Adversarial Networks* (GAN) or diffusion models. The latter is being used in state-of-the-art AI systems such as OpenAI's DALL·E2 [13] or Midjourney [14]. By working with a so-called *encoder-decoder* network structure, one can build a model that reverses the prediction process. This is often referred to as *inverse design*, where the model predicts a design based on physical target properties. In the papers by Hanakata et al. [6, 7] both the accelerated search and the inverse design approach was proven successful to create novel metamaterial Kirigami designs with the graphene sheet.

Hanakata et al. attribute the variation in mechanical properties to the non-linear effects arising from the out-of-plane buckling of the sheet. Since it is generally accepted that the surface roughness is of great importance for frictional properties it can be hypothesized that Kirigami-induced out-of-plane buckling can also be exploited for the design of frictional metamaterials. For certain designs, we might hope to find a relationship between the stretching of the sheet and frictional properties. If significant, this could give rise to an adjustable friction beyond the point of manufacturing. For instance, the grasping robot might apply such a material as artificial skin for which stretching or relaxing of the surface could result in a changeable friction strength.

In addition, the Kirigami graphene properties can be explored through a potential coupling between the strain and the normal load, through a nanomachine design, with the aim of altering the friction coefficient. This invites the idea of non-linear friction coefficients which might in principle also take on negative values. This would constitute a rarely found property which is mainly observed for the unloading phase of adhesive surfaces [15] or in the loading phase of particular heterojunction materials [16, 17].

To the best of our knowledge, Kirigami has not yet been implemented to alter the frictional properties of a nanoscale system. However, in a recent paper by Liefferink et al. [18] it is reported that macroscale Kirigami can be used to dynamically control the macroscale roughness of a surface through stretching. They reported that the roughness change led to a changeable frictional coefficient by more than one order of magnitude. This supports the idea that Kirigami designs can be used to alter friction, but we believe that taking this concept to the nanoscale would involve a different set of governing mechanisms and thus contribute to new insight in this field.

1.2 Goals

In this thesis, we investigate the prospects of altering the frictional properties of a graphene sheet through the application of Kirigami-inspired cuts and stretching of the sheet. With the use of molecular dynamics (MD)

[1] In computer science we define an *intractable* problem as a problem with no *efficient* algorithm to solve it nor any analytical solutions. The only way to solve such problems is the *brute-force* approach, simply trying all possible combinations, which is often beyond the capabilities of computational resources.

simulations, we evaluate the frictional properties of various Kirigami designs under different physical conditions. Based on the MD results, we investigate the possibility to use machine learning for the prediction of frictional properties and subsequently using the model for an accelerated search of new designs. The main goals of the thesis can be summarized as follows.

1. Design an MD simulation procedure to evaluate the frictional properties of a Kirigami graphene sheet under specified physical conditions.
2. Develop a numerical tool to generate various Kirigami designs, both by seeking inspiration from macroscale designs and by the use of a random-walk-based algorithm.
3. Investigate the frictional behavior under varying strain and load for different Kirigami designs.
4. Develop and train a machine learning model to predict the MD simulation results and perform an accelerated search of new designs with the goal of optimizing certain frictional properties.

1.3 Contributions

By working towards the goals outlined above (Sec. 1.2), I have discovered a non-linear relationship between the kinetic friction and the strain for certain Kirigami patterns. This phenomenon was found to be associated with the out-of-plane buckling of the Kirigami sheet but with no clear relationship to the contact area or the tension in the sheet. I found that this method does not provide any mechanism for a reduction in friction, in comparison to a non-cut sheet. However, the straining of certain Kirigami sheets allows for a non-monotonic increase in friction. The relationship to normal load was proven negligible in this context and I have demonstrated that a coupled system of load and strain (through sheet tension) can exhibit a negative friction coefficient in certain load ranges. Moreover, I have created a dataset of roughly 10,000 data points for assessing the employment of machine learning and accelerated search of Kirigami designs. I have found, that this approach might be useful, but that it requires an extended dataset in order to produce reliable results for a search of new designs.

During my investigations, I have built three numerical tools, in addition to the usual scripts for data analysis, which are available on Github [19]. The tools are summarized in the following.

- I have written a LAMMPS-based [20] tool for simulating and measuring the frictional properties of a graphene sheet sliding on a substrate. The code is generally made flexible with regard to the choice of sheet configuration, system size, simulation parameters and MD potentials, which makes it applicable for further studies on this topic. I have also built an automized procedure to carry out multiple simulations under varying parameters by submitting jobs to a computational cluster via an ssh connection. This was done by adding minor additions to the Python package developed by E. M. Nordhagen [21].
- I have generated a Python-based tool for generating Kirigami patterns and exporting these in a compatible format with the simulation software created. The generation of molecular structures is done with the use of ASE [22]. Our software includes two classes of patterns inspired by macroscale designs and a random walk algorithm which allows for a variety of different designs through user-defined biases and constraints. Given our system size of choice, the first two pattern generators are capable of generating on the order of 10^8 unique designs while the random walk generator allows for significantly more.
- I have built a machine-learning tool based on Pytorch [23] which includes setting up the data loaders, a convolutional network architecture, a loss function, and general algorithms for training and validating the results. Additionally, I have written several scripts for performing grid searches and analyzing the model predictions in the context of the frictional properties of graphene.

All numerical implementations have been originally developed for this thesis except for the libraries mentioned above along with common Python libraries such as Numpy and Matplotlib.

1.4 Thesis structure

The thesis is divided into two parts. In Part I we introduce the relevant theoretical background, and in Part II we present the numerical implementations and the results of this thesis.

Part I contains a description of the theoretical background related to Friction (**??**), Molecular Dynamics (**??**) and Machine Learning (Chapter 2). In **??** we formulate our research questions in the light of the friction theory.

In Part II, we begin by presenting the system in **??** which includes a definition of the main parts of the system and the numerical procedures related to the MD simulation. Here we also present the generation of Kirigami designs. In **??** we carry out a pilot study where we evaluate the simulation results for various physical conditions and compare a non-cut sheet to two different Kirigami designs. In **??**, we further explore the Kirigami patterns through the creation of a dataset and the employment of machine learning and an accelerated search for new designs. In **??**, we use the results from the pilot study to demonstrate the possibility to achieve a negative friction coefficient for a system with coupled load and strain. Finally, in **??**, we summarize our results and provide an outlook for further studies. Additional figures are shown in **??**, **??** and **??**.

Part I

Background Theory

Chapter 2

Machine Learning

We will use machine learning to predict the friction resulting from the straining and loading of a given Kirigami sheet. To this end, we will generate data through MD simulations that will serve as the ground truth for training a machine learning model. The advantage of using machine learning for this purpose is that it can significantly speed up the exploration of new configurations compared to that using full MD simulations. However, there is no guarantee that the machine learning model can accurately capture the physical mechanisms governing our system. Hence, a key objective is to assess the viability of this approach in the study of Kirigami friction, which we will pursue using traditional machine-learning methods. In this chapter, we introduce the key concept behind machine learning and some of the concepts and techniques relevant to our implementation. For the numerical implementation, we will use the machine learning framework PyTorch [23].

2.1 Neural network

The neural network, or more precisely the *feed-forward dense neural network*, is one of the original concepts in machine learning arising from the attempt of mimicking the way neurons work in the brain [24, 25]. The neural network can be considered in terms of three major parts: The input layer, the so-called *hidden layers* and the output layer as shown in Fig. 2.1. The input is described as a vector $\mathbf{x} = x_0, x_1, \ldots, x_{n_x}$ where each input x_i is usually denoted as a *feature*. The input features are densely connected to each of the *nodes* in the first hidden layer as indicated by the straight lines in Fig. 2.1. Each line represents a weighted connection that can be adjusted to configure the importance of that feature. Similar dense connections are present throughout the hidden layers to the final output layer. For a given note $a_j^{[l]}$ in layer l the input from all nodes in the previous layer $l-1$ are processed as

$$a_j^{[l]} = f\left(\sum_i w_{ij}^{[l]} a_i^{[l-1]} + b_j^{[l]}\right),$$

where $w_{ij}^{[l]}$ is the weight connection node $a_i^{[l-1]}$ of the previous layer to the node $a_j^{[l]}$ in the current layer. Note that having the weight belong to layer l as opposed to $l-1$ is simply a notation choice. $b_j^{[l]}$ denotes a bias and $f(\cdot)$ is the so-called *activation function*. The activation function provides a non-linear mapping of the input to each node. Without this, the network will only be capable of approximate linear functions [24]. Two common activation functions are the *sigmoid*, mapping the input to the range $(0, 1)$, and the *ReLU* which cuts off negative contributions

$$\text{Sigmoid:} \quad f(z) = \frac{1}{1+e^{-z}}, \qquad \text{ReLU:} \quad f(z) = \begin{cases} z & z > 0 \\ 0 & z \leq 0 \end{cases}.$$

Often the same activation function is used throughout a network, except for the output layer where the activation function is usually omitted or the sigmoid is used for classification tasks. The whole process of sending data through the model is called *forward propagation* and constitutes the mechanism for mapping an input $\mathbf{x}$ to the model output $\hat{\mathbf{y}}$. In order to get useful predictions we must *train* the model which involves tuning the model parameters, i.e. the weights and biases.

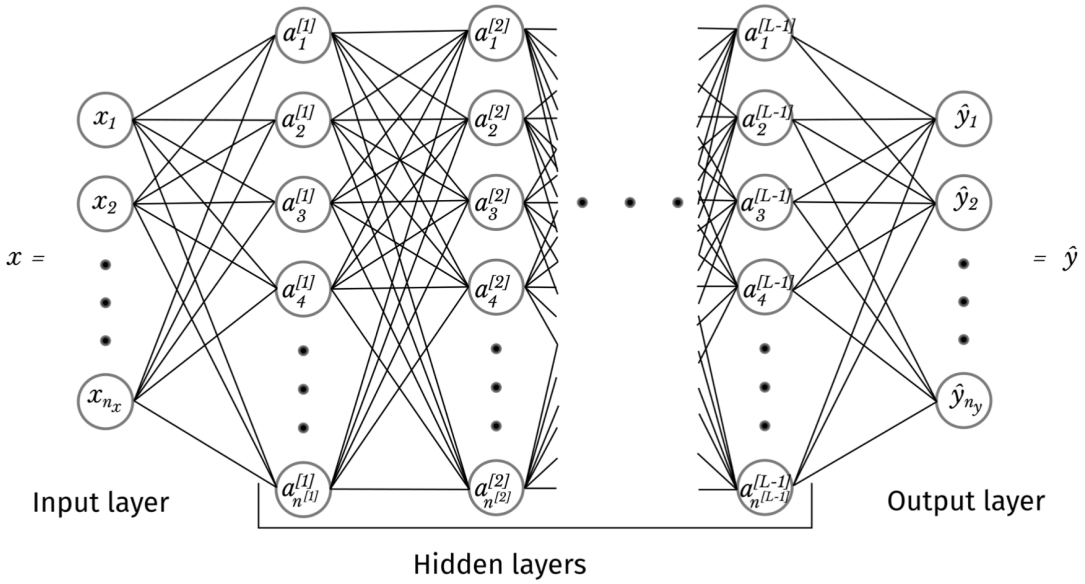


Figure 2.1: Illustration of a general feed-forward dense neural network with n_x input features and n_y outputs. Reproduced from [26].

The model training relies on two core concepts: *backpropagation* and *gradient descent* optimization. First, we define the error associated with a model prediction, otherwise known as the *loss*, through the *loss function* $L(\hat{\mathbf{y}}, \mathbf{y})$ that evaluates the model output $\hat{\mathbf{y}}$ against the ground truth $\mathbf{y}$. For a continuous scalar output, we might simply use the mean squared error (MSE)

$$L_{\text{MSE}} = \frac{1}{n_y} \sum_{i=1}^{n_y} (y_i - \hat{y}_i)^2. \tag{2.1}$$

For a binary classification problem, meaning that the output is either True or False (1 or 0), a common choice is binary cross entropy (BSE)

$$L_{\text{BSE}} = -\sum_{i=1}^{n_y} \Big[y_i \log(\hat{y}_i) + (1 - y_i) \log(1 - \hat{y}_i) \Big] = \sum_{i=1}^{n_y} \begin{cases} -\log(\hat{y}_i), & y_i = 1 \\ -\log(1 - \hat{y}_i), & y_i = 0. \end{cases} \tag{2.2}$$

Without going into details with the derivation we can convince ourselves that the error is minimized for the correct prediction and maximized for the worst prediction. When the ground truth is $y_i = 1$ we get the negative loss contribution $-\log(\hat{y}_i)$ where a correct prediction $\hat{y}_i \to 1$ yields $L_i \to 0$. For a wrong prediction $\hat{y}_i \to 0$ the loss contribution will diverge $L_i \to \infty$. Similar applies to the case of $y_i = 0$ with opposite directions.

Given a loss function, we can calculate the loss gradient $\nabla_\theta L$ with respect to each of the model parameters θ, being the weights and biases in the model. This is called *backpropagation* since we follow the propagation of the errors as we go back through the model layers. We calculate the gradients using the derivative chain rule. These gradients express how each parameter is connected to the loss and the overall idea is then to "nudge" each parameter in the right direction to reduce the loss. We usually denote a full cycle of forward propagation, backpropagation and an update of all model parameters as one *epoch*. We calculate the updated parameter θ_t for epoch t using the *gradient descent* method

$$\theta_t = \theta_{t-1} - \eta \nabla_\theta L(\theta_t). \tag{2.3}$$

Gradient descent is analog to taking a step in parameter space in the direction that yields the biggest decrease in the loss. If we imagine a simplified case with only two parameters θ_1 and θ_2 we can think of these as longitude and latitude coordinates on a map and the loss being the terrain height. The negative gradient $-\nabla_\theta L(\theta_t)$

represents the direction of the steepest loss decline perpendicular to the contour lines shaped by the loss function terrain as shown in Fig. 2.2. Notice, however, that state-of-the-art models in general contain on the order of 10^6–10^9 parameters [27] which poses some challenges for the visualization. The length of each step is proportional to the gradient norm $||\nabla_\theta L(\theta_t)||$ and the learning rate η. There are three main flavors to the gradient descent: Batch, stochastic and mini-batch gradient descent. In *batch gradient descent* we simply calculate the gradient based on the entire dataset by averaging the contribution from each data point before updating the parameters. This gives the most robust estimate of the gradient and thus the most direct path through parameter space in terms of minimizing the loss function as indicated in Fig. 2.2a. However, for big datasets, this calculation can be computationally heavy as it must carry the entire dataset in memory at once. A solution to this issue is provided by *stochastic gradient descent* (SGD) which considers only one data point at a time. Each data point is chosen randomly with replacement and the parameters are updated based on the corresponding gradient. This leads to more frequent updates of the parameters and a more "noisy" path through parameter space as shown in Fig. 2.2b. Under some circumstances, this might compromise the precision. However, the presence of noise can increase the likelihood of avoiding local minima in parameter space. The *mini-batch gradient descent* serves as a middle ground between the above-mentioned methods by dividing the full dataset into a subset of mini-batches. Each parameter update is then based on the gradient within a mini-batch. By choosing a suitable batch size we get the robustness of the (full) batch gradient descent and the computational efficiency and resistance to local minima of the SGD method. We will use the mini-batch method for our implementation.

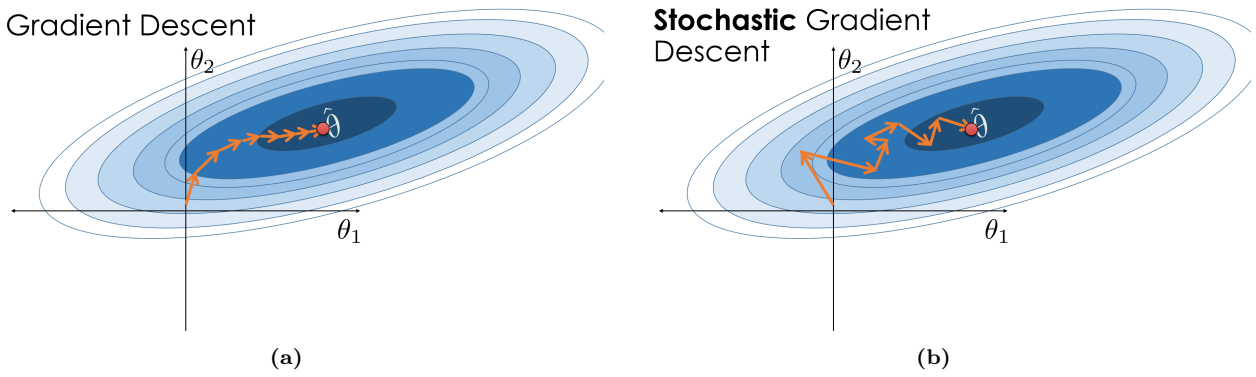


Figure 2.2: Qualitative illustration of the gradient descent method for a simplified problem with only two parameters θ_1 and θ_2. The blue shade and lines indicate the contour map of the loss function with the darker shade denoting a lower loss. (a) The batch gradient descent method resulting in a relatively straight path toward the optimal parameters. (b) The stochastic gradient descent method resulting in a more noisy path toward the optimal parameters. Reproduced from [28].

2.1.1 Optimizers

The name *optimizers* generally refers to a variety of gradient descent methods. In our study, we will use the ADAM (adaptive moment estimation) optimizer [29]. ADAM combines several "tricks in the book" which we will introduce in the following.

One considerable extension of the gradient descent scheme is the introduction of a momentum term m_t such that we get

$$\theta_t = \theta_{t-1} - m_t, \qquad m_t = \alpha m_{t-1} + \eta \nabla_\theta L(\theta_t), \tag{2.4}$$

with $m_0 = 0$. If we introduce the shorthand $g_t = \nabla_\theta L(\theta_t)$ we find

$$\begin{aligned} m_1 &= \alpha m_0 + \eta g_1 = \eta g_1 \\ m_2 &= \alpha m_1 + \eta g_2 = \alpha^1 \eta g_1 + \eta g_2 \\ m_3 &= \alpha m_2 + \eta g_3 = \alpha^2 \eta g_1 + \alpha \eta g_2 + \eta g_3 \\ &\vdots \\ m_t &= \eta \left(\sum_{k=1}^{t} \alpha^{t-k} g_k \right). \end{aligned} \tag{2.5}$$

Hence m_t is a weighted average of the gradients with an exponentially decreasing weight. This act as a memory of the previous gradients and aid to pass local minima and to some degree plateaus in the parameter space. It also provides a general steadiness to the gradient descent which counteracts the transition from batch to mini-batch gradient descent. A variation of momentum can be achieved with the introduction of the exponential moving average (EMA) which builds on the recursion

$$\begin{aligned} \mathrm{EMA}(g_1) &= \alpha \overbrace{\mathrm{EMA}(g_0)}^{\equiv 0} + (1-\alpha) g_1 \\ \mathrm{EMA}(g_2) &= \alpha \mathrm{EMA}(g_1) + (1-\alpha) g_2 \\ &\vdots \\ \mathrm{EMA}(g_t) &= \alpha \mathrm{EMA}(g_{t-1}) + (1-\alpha) g_t = \sum_{k=0}^{t} \alpha^{t-k} (1-\alpha) g_t, \end{aligned}$$

which is similar to that of momentum Eq. (2.5), but with the explicit weighting by $(1-\alpha)$. The second moment of the exponential moving average is utilized in the root mean square propagation method (RMSProp) which is motivated by the issue of passing long loss plateaus in the parameter space. Since the size of the updates are otherwise proportional to the norm of the gradient

$$\theta_{t+1} = \theta_t - \eta g_t \implies ||\theta_{t+1} - \theta_t|| = \eta ||g_t||,$$

we might get the idea of normalizing the gradient step by the norm $||g_t||$. However, this does not immediately solve the problem of long plateaus as we need to consider multiple past gradients. Hence, this can be done with the use of the EMA. When reentering a steep region again we need to "quickly" downscale the gradient steps which can be achieved more efficiently by using the squared norm $||g_t||^2$ for the EMA which makes it more sensitive to outliers. From this motivation, the RMSProp update scheme is given

$$\theta_t = \theta_{t-1} - \eta \frac{g_t}{\sqrt{\mathrm{EMA}(||g_t||^2)} + \epsilon}, \tag{2.6}$$

where ϵ is simply a small number to avoid division by zero issues and thus ensure numerical stability.

ADAM merges the idea of a first-order EMA for the momentum m_t, and a second-order EMA v_t for gradient normalization similar to the root mean square propagation technique in Eq. (2.6)

$$\begin{aligned} m_t &= \beta_1 m_{t-1} + (1-\beta_1) g_t, \\ v_t &= \beta_2 v_{t-1} + (1-\beta_2) g_t^2. \end{aligned}$$

Since m_t and v_t are initially set to zero ADAM introduces the scaling terms $(1-\beta_1^t)$ and $(1-\beta_2^t)$ to correct for a bias towards zero. The ADAM scheme is given [29]

$$\theta_{t+1} = \theta_t - \eta \frac{\hat{m}_t}{\sqrt{\hat{v}_t} + \epsilon}, \qquad \hat{m}_t = \frac{m_t}{1-\beta_1^t}, \qquad \hat{v}_t = \frac{v_t}{1-\beta_2^t}. \tag{2.7}$$

2.1.2 Weight decay

By adding a so-called *regularization* to the loss function we can penalize high magnitudes of the model parameters, usually intended for the model weights. This is motivated by the idea of preventing overfitting during training,

which we will address in more detail in Sec. 2.3. The most common way to regularize the loss function is by the use of L2 regularization, adding the squared l^2 norm $||\theta||_2^2$, where $||\theta||_2 = \sqrt{\theta_1^2 + \theta_2^2 + \ldots}$, to the model. The loss and gradient then become

$$L_{l^2}(\theta) = L(\theta) + \frac{1}{2}\lambda||\theta||_2^2 \tag{2.8}$$

$$\nabla_\theta L_{l2}(\theta) = \nabla_\theta L(\theta) + \lambda\theta, \tag{2.9}$$

where $\lambda \in [0, 1]$ is the weight decay parameter. The name *weight decay* relates to the fact that some practitioners only apply this penalty to the weights in the model, but we will include the biases as well (standard in PyTorch). Following the original gradient descent scheme Eq. (2.9) we get

$$\theta_{t+1} = \theta_t - \eta g_t - \eta\lambda\theta_t = \theta_t \underbrace{(1 - \eta\lambda)}_{\text{weight decay}} - \eta g_t. \tag{2.10}$$

We notice that choosing a large weight decay ($\lambda \to 1$) will downscale the model parameters while choosing a low weight decay ($\lambda \to 0$) yields the original gradient descent scheme. Note that we will use the weight decay principle in combination with ADAM. In Eq. (2.10) we have simply used the original gradient descent scheme Eq. (2.3) since this makes it easier to demonstrate the consequences of introducing the L2 regularization into the loss function Eq. (2.8).

2.1.3 Parameter distributions

In order to get optimal training conditions it has been found that the initial state of the weights and biases are important [30]. First of all, we must initialize the weights by sampling from some distribution. If the weights are set to equal values the gradient across a layer would be the same. This results in a complexity reduction as the model can only encode the same values across the layer. Further, we want to consider the gradient flow during training. Especially for deep networks, networks with many layers, we must pay attention to the problem of *vanishing* or *exploding* gradients. If we for instance consider the sigmoid activation function and its derivative

$$f(z) = \frac{1}{1 + e^{-z}}, \qquad f'(z) = \frac{df(z)}{dz} = \frac{e^{-z}}{(1 + e^{-z})^2} = \frac{e^z}{(1 + e^z)^2},$$

we notice that for large and small input values z we get $f(z \to \pm\infty) \to 0$. However, even a small finite gradient can vanish throughout a deep network as the calculation of the gradient involves the chain rule. This gives rise to a gradient that potentially gets smaller and smaller for each layer it passes in the backpropagation. A similar problem can be found with the ReLU activation function which contributes toward a gradient of zero for inputs $z < 0$. This can be mitigated by the so-called leaky RelU which maps the $z < 0$ to a small negative slope $a < 0$ as $f(z) = az$. On the other hand, we have exploding gradients, which are simply a result of the chain rule gradient calculation. For a sufficiently deep network, the gradient can grow exponentially large and sometimes result in a numerical overflow. One approach to mitigate this issue is with the use of gradient clipping, where all gradients above a certain value are manually set to a predefined maximum number. While there exist techniques to accommodate the problem of vanishing or exploding gradients as mentioned above, they both benefit from a properly initialized set of weights. That is, we want the gradients across a given layer to have a zero mean while the variance is similar between layers in the model. This balanced gradient flow is more likely to happen if we initialize the weight by the same criteria [30]. The specific actions to achieve this depends on the model architecture, including the choice of activation functions. For instance, using the ReLU activation functions it was found that the node standard deviation will depend on the number of input nodes from the previous layer $N^{[l-1]}$ as $\sqrt{N^{[l-1]}/2}$ [31]. Thus we can sample the weights from a zero mean normal distribution $N \sim (0, 2/N^{[l-1]})$ with a standard deviation $\sqrt{2/N^{[l-1]}}$ to ensure a balanced weight initialization. This particular case (using the ReLU activation function) is part of the Kaiming initialization scheme [31] which is standard in Pytorch. The bias is initialized from a similar consideration.

Batch normalization is another technique that can help reduce the issue of poor gradient flow. Furthermore, it can benefit by speeding up convergence and making the training process more stable [32]. In general, model parameters are modified throughout training meaning that the range of values coming from a previous layer will shift (internal covariate shift), even though the same training data is fed through the network repeatedly. By scaling the inputs for a given layer, for each mini-batch, we can mitigate this problem and make for a more

standardized input range. For layer l we calculate the mean $\mu^{[l]}$ and variance $(\sigma^{[l]})^2$ across the layer with nodes $x_1^{[l]}, x_2^{[l]}, \ldots, x_m^{[l]}$ for each mini-batch of size m as

$$\mu^{[l]} = \frac{1}{m}\sum_i^m x_i^{[l]}, \qquad (\sigma^{[l]})^2 = \frac{1}{m}\sum_i^m (x_i^{[l]} - \mu^{[l]})^2.$$

We then perform a normal scaling of the inputs within the batch

$$\hat{x}_i^{[l]} = \frac{x_i^{[l]} - \mu^{[l]}}{\sqrt{(\sigma^{[l]})^2 + \epsilon}},$$

where ϵ is a small number to ensure numerical stability (similar to what we used for RMSProp gradient descent). In the final step, the input values are rescaled as

$$\tilde{x_i}^{[l]} = \gamma^{[l]}\hat{x}_i^{[l]} + \beta^{[l]}$$

with trainable parameters γ and β [32].

2.1.4 Learning rate decay strategies

Until now we have assumed a constant learning rate, but many training schemes use a changing learning rate beyond the adaptiveness included in the optimizers covered so far. Under some circumstances, it can be beneficial to start with a higher learning rate to speed up the initial part of training and then lower the learning rate for the final part [33]. One straightforward strategy is a step-wise learning rate decay where the learning rate is reduced by a factor $\gamma \in (0, 1)$ every K steps. A more smooth decrease can be achieved with a polynomial decay $\eta_t = \eta_0/t^\alpha$ for $\alpha > 0$. More advanced approaches use multiple cycles of increasing and decreasing cycles. We will mainly concern ourselves with a one-cycle policy for which we start at an intermediate value, increase toward a maximum bound and then decrease toward a final lower learning rate bound. We do this by following a cosine function that is shifted and scaled to increase towards the maximum bound for the first 30% of the training length and decrease toward the lower learning rate bound for the remaining epochs.

2.2 Convolutional Neural Network

Convolutional Neural Networks (CNNs) build upon many of the same concepts as introduced with the feed-forward neural network in Sec. 2.1. The difference lies in its specialization for a spatially correlated input, such as pixels in an image. In a dense neural network, every node is connected to each of the nodes from the previous layers which is not ideal for image recognition. For instance, if we want the model to recognize images of animals the dense network will be very sensitive to where that animal is placed within the frame. The CNN is motivated by the idea of capturing spatial relations in the input, but without being sensitive to the relative placement within the input, i.e. being translational invariant. This is achieved by having a so-called *kernel* or *filter* which slides over the images[2] as it processes the input. The overall flow of data for a typical convolutional network including a final fully connected neural network is illustrated in Fig. 2.3.

A convolutional layer contains multiple kernels, each consisting of a set of trainable weights and a bias. Each kernel will produce a separate output channel to the resulting *feature map* layer. The kernel has a 2D spatial size, specific to the model architecture, and a depth that matches the number of input channels to the layer. For instance, a typical RGB image will have three channels, while the number of channels usually increases for each layer in the model. The kernel lines up with the image and calculates the feature map output as a dot product between the weights in the kernel and the aligning subset of the input. This is done for each input channel and summed up with the addition of a bias as illustrated in Fig. 2.4b. The kernel then slides over by a step size given by the *stride* parameter and repeats the calculation. Choosing a stride of 2 or higher results in a reduction of the output spatial size. If we want to preserve the spatial size we must keep a stride of one and additionally

[2]Note, that we will be using the word "image" as a reference for a spatially dependent input, but in reality, it does not have to be an actual image in the classical sense.

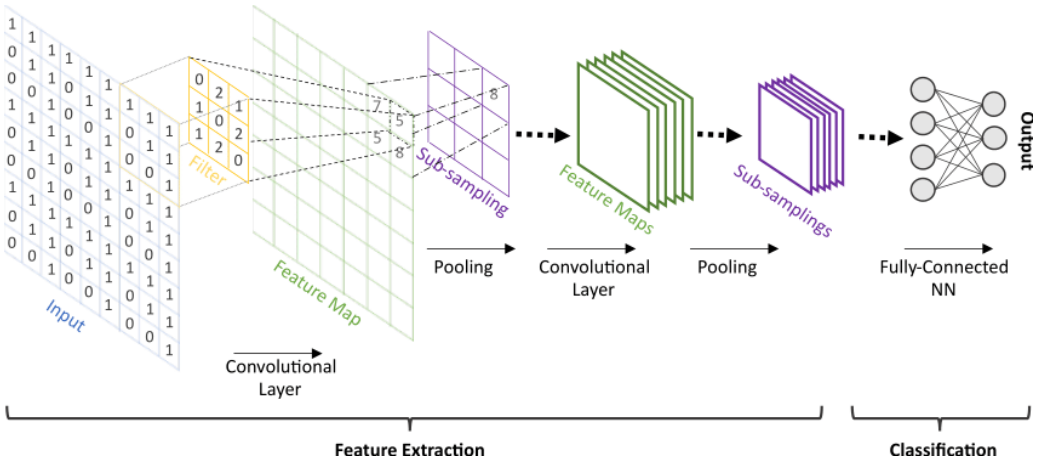


Figure 2.3: Representation of a Convolutional Neural Network (CNN). The CNN performs automatic spatial feature extraction from images by successively applying feature filters that create feature maps (Convolutional layer) and compressing these maps (Pooling layer). Based on the final feature maps, a fully-connected neural network does a prediction, which can be a classification or regression. Figure and caption reproduced from [34].

apply *padding* to the input images, such that we can achieve one kernel position for each input "pixel". The spatial size of the feature map is given as

$$N_d^{[l]} = \left\lfloor \frac{N_d^{[l-1]} - F_d + 2P}{S} + 1 \right\rfloor, \tag{2.11}$$

for padding P, stride S, spatial size of the kernel filter F_d, spatial size of the input $N_d^{[l-1]}$, for dimension $d = \{x, y\}$ and layer l. The *down-sampling* is often done through a pooling layer. A pooling layer is reminiscent of a kernel, but instead of calculating the output as a dot product, it calculates the mean (mean pooling) or the max value (max pooling) of the values within its scope. For instance, by using a max pooling of size 2×2 and stride 2 we essentially half the dimensions of the image as dictated by Eq. (2.11). CNNs will often use repeating series of convolution (applying a kernel), pooling and then an activation function. Most architectures aim to down-sample the spatial input while increasing the number of channels throughout the model layers. This results in a smaller set of features extracted from the input which then can then be fed into a dense network, or *fully connected* connected neural network, as also illustrated in Fig. 2.3. The convolution part aims to handle the transition from a spatial input to some internal features. For a model which recognizes animals, we would perhaps think of features such as the number of legs, size, color and so on. In practice, however, the network will not give readily interpreted features for the processing in the fully connected layer, but the concept is still the same.

For a CNN, we often consider the *receptive field*. The receptive field relates to the spatial size of the input that affects a given node in the feature map of a given layer in the model. This term is often used in reference to the output nodes. Fig. 2.5 illustrates the receptive field for a 1D representation of a CNN with repetitive use of a kernel of width 2 and stride 1. Going from the output and backward, we see that the output layers are connected to two nodes in the previous layer. Each of these nodes is connected to two nodes in the layer before that, however with one of them overlapping due to the stride of 1. By back-tracking all the way to the input layer we see that this corresponds to a receptive field of $D = 5$. This means that a single output node is only affected by the 5 input nodes within its receptive field. By increasing the filter size and the stride the receptive field will grow a lot faster than shown in this example. If we assume a constant filter size throughout the network F, a stride S_l from layer $l-1$ to l we get that the receptive field D_l with respect to a certain layer in a given spatial dimension is

$$D_l = D_l + \left[(F_l - 1) \cdot \prod_{i=1}^{l-1} S_i \right], \tag{2.12}$$

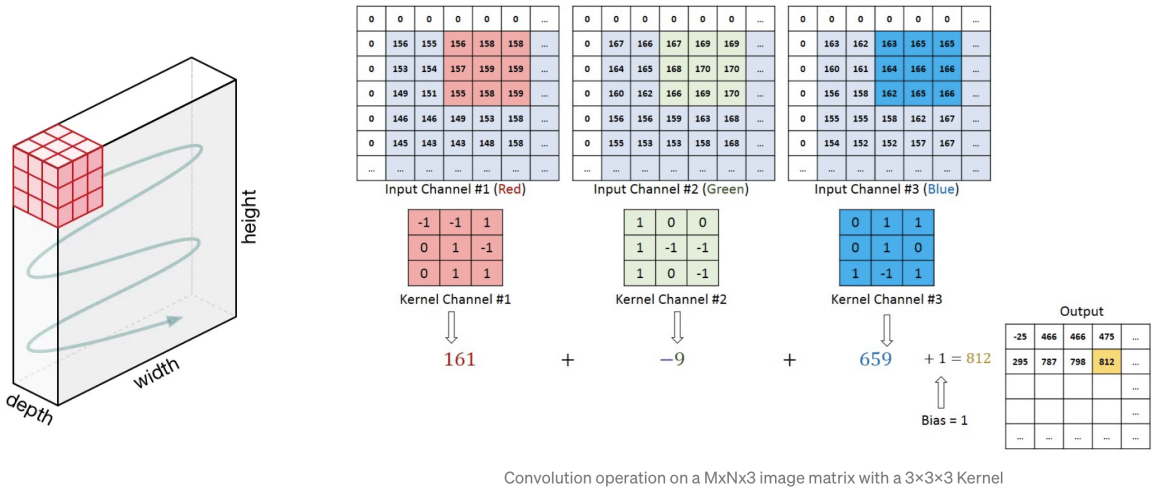


Figure 2.4: Illustration of the convolution procedure for a 3×3 kernel with a depth of 3 matching an RGB input with 3 channels (a) The kernel goes through the various positions aligning with the input as sketched with the arrow. The precise number of positions along this path depends on the stride parameter and the choice of padding. (b) A specific example of the calculations involved in the convolution process. Here a rim of zero padding is included. For each kernel position, a dot product is calculated between the aligning input and the kernel weights for each corresponding channel. This results in an output value for each channel which is then summed up and added with a bias to constitute the final output in the feature map. Reproduced from [35].

with $D_0 = 1$ and $l = 0$ as the input layer. Note that by convention, the product of zero elements is 1, such that for the first layer, the product is 1. Eq. (2.12) apply for both spatial dimensions individually.

The receptive field is important for understanding the connectivity in the model since the output will be completely independent of the inputs and feature maps outside the receptive field. Furthermore, we differentiate between the *theoretical* receptive field and the *effective* receptive field. The effective receptive field will have a Gaussian distribution within the theoretical receptive field because the nodes in the center of the receptive field will have more connections leading to the output, as seen in Fig. 2.5. Thus, in practice, the effective receptive field will be smaller than the theoretical. Implementations like dilated convolutions, which make the filter expand in circumference and skip positions within the filter, can be used to further increase the effective field.

2.2.1 Training, validation and test data

So far, we have simply considered the concept of *training data* as a means to update the model parameters. Yet, we want to evaluate the model performance as it improves. The problem arises immediately from the fact that a complex model can fit about any function. More precisely, it has been proven that a deep convolutional neural network is universal (it follows the universal approximation theorem), meaning that it can approximate any continuous function to an arbitrary accuracy when the depth of the network is large enough [37]. Thus for a complex model, it is just a matter of time (epochs) before the model eventually finds a good approximation for the training data. However, we want the model to learn general trends and not to "memorize" all the data points which are known as *overfitting*. While the predictions for the training data can grow arbitrarily good in most cases, the performance on unseen data within the same domain will yield poor performance in the case of overfitting. The common way to address this issue is by putting aside a subset of the data, the so-called *validation* data, which we use to validate the model performance during and after training. By keeping this validation set separate from the training data we can get a more reliable performance estimate for the model. It is crucial to use a random partitioning in order to ensure an equal distribution of data across both sets. To strike a balance between the quality of training and validation, a commonly used partitioning ratio is usually around 20:80 in favor of the training set which we will adapt as well. A third data set often forgotten is the

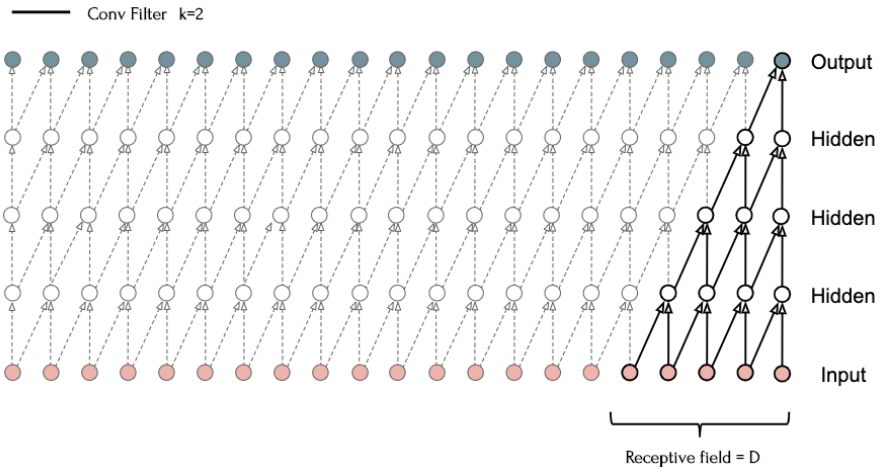


Figure 2.5: An illustration of the receptive field D with respect to an output node in a 1D convolutional network. This example uses a width of 2 for the kernel and a stride of 1. Reproduced from [36].

test set. While the validation set should be kept unseen from the model training, the test set should be kept unseen from the model developer when choosing the model architecture and hyper-parameters. We define a hyper-parameter as a variable to be set prior to the actual application of the learning algorithm, one that is not selected by the learning algorithm itself [38]. This includes parameters such as learning rate, momentum and weight decay, but not the weights and biases as these are updated by the learning algorithm. When adjusting the hyper-parameters we will use the performance on the validation set as a guiding metric. Hence, our choices can eventually lead to high-level overfitting through the hyper-parameter choices. Hence, we should denote a test set for the final evaluation of our model which has not been considered before the end. Formally, this is the only reliable performance metric for the model. However, it is important that this set also possesses the same data distribution to ensure a reliable performance estimate.

2.3 Overfitting and underfitting

The balance between underfitting and overfitting is essential in model training and it is closely related to the complexity of the model and the selected hyper-parameters. A typical textbook visualization of underfitting and overfitting is given in Fig. 2.6a.

As we begin to train or model both the training and validation loss is decreasing. At a certain point, the model may begin to not only capture general trends but also specific trends in the training data, which may be related to noise. This marks the transition from underfitting to overfitting where the validation loss will increase due to a loss of generalization. However, the training loss will continue its steady decline as the training procedure optimize exclusively for the training data loss. *Early stopping* can be utilized to detect this transition and stop the training early in an attempt to hit the sweet spot between under- and overfitting. We will use a variation of this technique which is to store the best model based on the validation performance history. For this approach, we let the training finish and then store the model corresponding to the best validation score found during the entirety of the training. In principle, we can "get lucky" and find the model settings at a state that is overfitted for the validation set. However, we consider this to be highly unlikely for a reasonably big amount of data and a complex model with many model parameters to tune.

The underfitting and overfitting phenomena can also be thought of as a consequence of model complexity and not just training time. For a certain amount of epochs a simple model will yield underfitting and an overly complex model will yield overfitting. This can be expected to follow a similar qualitative trend as shown in Fig. 2.6a with the substitution of *model complexity* on the x-axis. Fig. 2.6b visualizes the concept of underfitting and overfitting in terms of the complexity regarding the fitting of a second-order polynomial. The figure shows how a simple linear function will make a crude approximation for the true curve, while an overly complex model will pick up the noise in the data and miss the general trend as well. However, the problem is that we do not know the true curve in practice. If we did, we would not need machine learning to approximate it

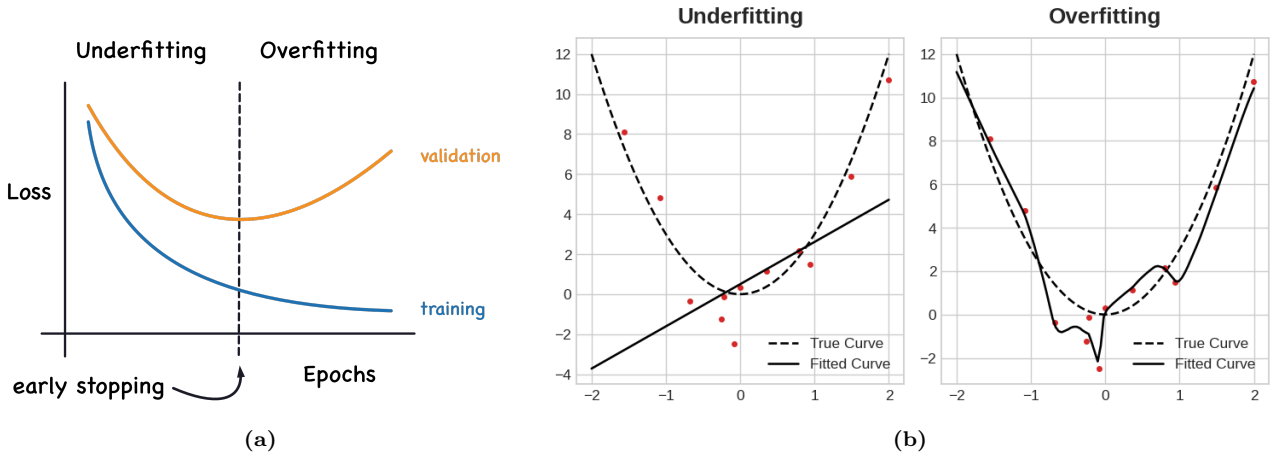


Figure 2.6: Illustrations regarding the concept of underfitting and overfitting. (a) A typical textbook representation of the transition from underfitting to overfitting during prolonged training. Initially, both the training and validation loss decreases for each epoch, but at some point, the validation will begin to increase again marking the beginning of overfitting. This motivates the idea of an early stopping point at the minimum of the validation loss curve. (b) A 2D example of fitting a curve (solid line) to a series of data points (red dots) generated from a second-order polynomial (dotted line) with additional noise. In the underfitting case, we fail to capture the general trend due to a low complexity in our fit, while in the overfitting case, our fit captures the noise in the data as well. Reproduced from [39].

in the first place. Without having additional insight into the governing source of the data the overfitting case from Fig. 2.6b seems to produce the most confident fit for all we know. Thus, it is important to utilize the validation loss in order to mitigate this problem.

2.4 Hypertuning

The training of a machine learning model revolves around tuning the model parameters such as weights and biases. However, as mentioned already, a handful of *hyper-parameters* remains for us to decide. First of all, we need to choose an architecture for the model. This includes high-level considerations such as whether to use a neural network or a convolutional network, but also lower-level considerations, such as the depth and the width of the model, i.e. the number of layers and the number of nodes/channels in a layer. In addition, we have to choose the loss function and the optimizer which come with hyper-parameters such as learning rate, momentum and weight decay. This extensive list of choices makes the designing of a functional machine learning procedure more complicated than simply hitting "run" for the learning algorithm. As N. Smith [33] puts it "Setting the hyper-parameters remains a black art that requires years of experience to acquire". In the following, we will review a general approach for choosing the learning rate, momentum and weight decay hyper-parameters based on the findings of [33]. The traditional approach is to perform a *grid search*, trying out different combinations of hyper-parameters in different training sessions, but this might rather quickly become computationally expensive and ineffective. In addition, hyper-parameters will depend on the training data, the model architecture and not at least each other. This makes it difficult to narrow down the choice one by one. N. Smith points to the fact that the validation loss can be examined early on for clues of either underfitting or overfitting.

The learning rate is often regarded as the most important hyper-parameter to tune [38]. Typical values are in the range $[10^{-6}, 1]$. Instead of simply running a grid search, we can perform a so-called *learning rate range test*. One then specifies the minimum and maximum learning rate boundaries for the range test and a learning rate step size. A range of 10^{-7} to 10 will most likely be appropriate, but the test will reveal this immediately. The idea is to vary the learning rate throughout the given range in small steps during a short pre-training. We will increase the learning rate for each iteration, i.e. each parameter update following a mini-batch, and thus we can run this test for a few epochs, or even a single one, depending on the number of mini-batches. For small learning rates, the model will converge slowly. As the learning rate approaches an appropriate value, the convergence speed will increase, which can be observed as a decrease in the validation loss. Eventually, the convergence will

stop and the validation loss will pass a minimum for which it will begin to diverge. This general behavior can be understood for the simplified 1D example of finding the minimum of a second-order polynomial as shown in Fig. 2.7. For small learning rates, the gradient descent update will provide a small step in the parameter space toward the minimum. However, if the learning rate is too small this will yield a slow convergence. On the other hand, if the learning rate becomes too large, we will effectively step past the minimum in one step. Each following step will overshoot the minimum further since the step length is proportional to the gradient of the loss, which leads to a diverging trend. From the learning rate range test, we can use the point of divergence as an upper bound for the learning rates when considering a cyclic learning rate scheme. For a constant learning rate scheme we can use the point of steppest decline of the validation loss as an estimate for the best learning rate choice [33].

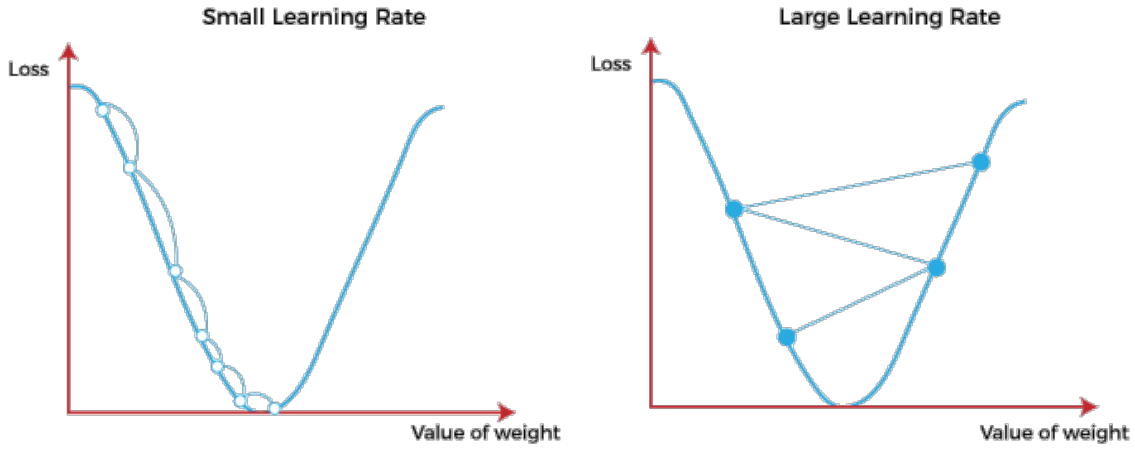


Figure 2.7: Qualitative illustration of the effect of learning rate choice for gradient descent on a simplified 1D problem. The left frame shows the gradient descent steps using a small learning rate where several steps are needed to approach the minimum of the loss. The right frame shows the corresponding steps with the choice of a too large learning rate which yields a diverging behavior. Reproduced from [40].

Next, we consider the choice of momentum. From the gradient descent scheme with momentum Eq. (2.4) we see that the momentum parameter α and the learning rate η have a similar effect on the parameter update

$$\theta_t = \theta_{t-1} - \eta g_t - \alpha m_{t-1},$$

since m_t is a moving average of the gradient g_t as well. Like the learning rate, we want to set the momentum value as high as possible without causing instabilities in the training. However, since the two are related to each other, they must also be considered together. N. Smith [33] reports that a momentum range test is not a useful approach to determine the optimal momentum value. Instead, he suggests doing a few short runs with different values of momentum, such as 0.99, 0.97, 0.95, and 0.9, to determine a suitable choice. By including momentum in the learning range test we can balance the learning rate accordingly. Moreover, for a cyclic learning rate scheme he reports that a cycling momentum scheme, reversed with respect to the learning rate, is beneficial. When the learning rate increase toward the upper bound the learning rate should decrease toward a lower bound and vice versa. Choosing a lower momentum bound of 0.80–0.85 is reported to give similar stable results [33].

Finally, we address weight decay. N. Smith [33] reports that weight decay is different from learning rate and momentum by the fact that weight decay is better chosen as a constant value as opposed to a cyclic scheme. However, the weight decay is dependent on the model complexity, learning rate and momentum choice. Hence, this can often be chosen after setting those. We can estimate a suitable choice by doing a rough grid search for values such as 0, 10^{-6}, 10^{-5} and 10^{-4} for complex architectures and 10^{-4}, 10^{-3} and 10^{-2} for more shallow architectures. Choosing the weight decay on the scale of exponents will often provide good enough precision in practice.

2.5 Prediction explanation

On a final note, we present a simple method for providing some insight into the prediction from a convolutional neural network. The high complexity of deep learning models limits our ability to understand the decision-making going into the model predictions beyond the input data. This is known as the *black box* problem. A lot of effort is currently being developed for making more transparent models, like decision trees with interpretable rules, and numerical tools for unpacking the inner workings of the model. We will consider a gradient-based method called *Grad-CAM* [41] which aims to highlight some of the important features of a convolutional network model prediction. The method utilizes the gradients for a certain feature map with respect to the loss.

To begin, we forward propagate the input through the model. Next, we decide on a feature map of interest and calculate the gradients with respect to the loss of a certain target output. In classification tasks, one would often choose the predicted class with the highest score. By doing so, the gradients can be utilized to determine which parts of the feature map are more important for the prediction. We apply a ReLU activation layer to keep only the positive contributions. Since the convolutional layers preserve spatial information we can rescale the feature map gradients to make an input-sized heatmap allowing for an overlaid visualization on top of the input image. By using this method, we can obtain a visual indication of the specific regions in the image that plays a key role in the prediction. We can apply this technique to multiple layers of the model and even merge the outcomes as well for an analysis of the contribution from multiple feature maps.

Fig. 2.8 show an exemplary use, where the Grad-CAM analysis reveals the difference between a biased and unbiased model for the task of predicting professions. The biased model appears to be prioritizing the person's facial features over the objective cues provided by relevant equipment and work-related clothing.

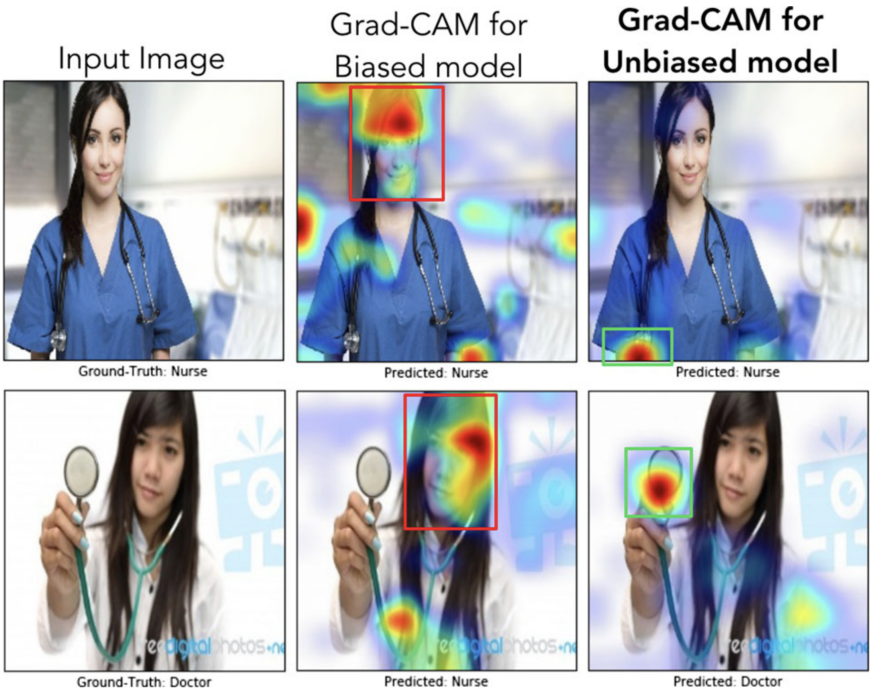


Figure 2.8: An example of the Grad-CAM method in use: In the first row, we can see that even though both models made the right decision, the biased model was looking at the face of the person to decide if the person was a nurse, whereas the unbiased model was looking at the short sleeves to make the decision. For the example image in the second row, the biased model made the wrong prediction (misclassifying a doctor as a nurse) by looking at the face and the hairstyle, whereas the unbiased model made the right prediction looking at the white coat, and the stethoscope. Figure and caption reproduced from [41].

2.6 Accelerated search using genetic algorithm

For the scope of finding new Kirigami designs which exhibit certain frictional properties, we are interested in utilizing a trained machine-learning model for further exploration. This reverses the design process as one has to find the right input to achieve a certain output. A possible strategy is to explore a range of inputs and use the model predictions as a guiding metric. One approach to this strategy is the genetic algorithm (GA) which is inspired by biological evolution mimicking the Darwin theory of the survival of the fittest [42]. GA is a population-based algorithm for which the basic elements are chromosome representation, fitness selection and biological-inspired operators. The chromosomes represent the genes for each individual in the population and typically take the form of a binary string. Each position within the chromosome is called a *locus* and has two possible values (0 or 1). A fitness function is defined to assign a score for all chromosomes based on some optimization objective. This plays a role for the biologically inspired operators for which the main ones are selection, mutation and crossover. Selection is the process of selecting chromosomes based on their fitness score for further processing. In mutation, some of the loci within a chromosome are changed and in crossover, chromosomes are merged to create offspring. GA has been implemented in many areas such as the traveling salesman problem [43], function optimization [44], adaptive agents in stock markets [45] and airport scheduling [46]. Wang et al. [47] note that a general drawback is a need for expertise when choosing parameters that match specific applications. They propose an accelerated genetic algorithm based on a Markov chain transition probability matrix to perform a guided search that reduces the number of parameter choices one has to make. The following introduction of this method is based on their work [47].

We define the binary population matrix $A_{ij}(t)$ at generation t, consisting of N rows denoting chromosomes $i \in \{0, 1, \ldots, N\}$ and L columns denoting the loci $j \in \{0, 1, \ldots L\}$. For our application, we let the locus represent an atom in the Kirigami pattern matrix which is flattened to fit the format of the population matrix. We carry forward the binary values with 0 meaning a removed atom and 1 a present atom. By the use of a fitness function $f(t)$, we sort the population matrix row-wise in descending order by fitness score, i.e. $f_i(t) \leq f_k(t)$ for $i \geq k$. In the spirit of Markov chains, we assume that some transition probability exists for the transition between the current state $A(t)$ and the next state $A(t+1)$. We assume that this transition probability only takes into account the mutation process, and thus we omit operators like crossover. For each generation, the chromosomes are sorted according to the fitness function and the chromosome at the i^{th} fittest place is assigned a ranking score $r_i(t)$ by some monotonic increasing ranking scheme. We take this to be

$$r_i(t) = \begin{cases} (i-1)/N', & i-1 < N' \\ 1, & \text{else} \end{cases},$$

with $N' = N/2$ from [47]. We assign a row mutation probability $a_i(t) = r_i(t)$ meaning that the probability for a mutation will increase towards the lower fitness scores. For the considerations of the mutation probability for the loci within each chromosome, we define the count of 0 and 1 states as $C_0(j)$ and $C_1(j)$ respectively. These are normalized as

$$n_0(j,t) = \frac{C_0(j)}{C_0(j) + C_1(j)}, \quad n_1(j,t) = \frac{C_1(j)}{C_0(j) + C_1(j)}.$$

We can thus describe the state of the j^{th} locus column as the state vector $\mathbf{n}(j,t) = (n_0(j,t), n_1(j,t))$. In order to direct the current population to a preferred state for locus j we consider the highest weight $W_i = 1 - r_i$ among the chromosomes for the case of the locus being 0 or 1 respectively. This corresponds to the targets

$$C_0'(j) = \max\{W_i | A_{ij} = 0;\ i = 1, \ldots, N\}$$
$$C_1'(j) = \max\{W_i | A_{ij} = 1;\ i = 1, \ldots, N\}.$$

These are normalized

$$n_0(j,t+1) = \frac{C_0'(j)}{C_0'(j) + C_1'(j)}, \quad n_1(j,t+1) = \frac{C_1'(j)}{C_0'(j) + C_1'(j)}, \tag{2.13}$$

to produce the target state vector $\mathbf{n}(j,t+1) = (n_0(j,t+1), n_1(j,t+1))$. This will serve as a direction for each locus to evolve in and thus we can formulate the Markov chain as

$$\begin{bmatrix} n_0(j,t+1) \\ n_1(j,t+1) \end{bmatrix} = \begin{bmatrix} P_{00}(j,t) & P_{10}(j,t) \\ P_{01}(j,t) & P_{11}(j,t) \end{bmatrix} \begin{bmatrix} n_0(j,t) \\ n_1(j,t) \end{bmatrix},$$

where the matrix represents the transition matrix. Since the probability must sum to one for the columns in the transition matrix we get

$$P_{01}(j,t) = 1 - P_{00}(j,t), \quad P_{11}(j,t) = 1 - P_{10}(j,t).$$

These conditions allow us to solve for the transition probability $P_{10}(j,t)$ in terms of the single variable $P_{00}(j,t)$

$$P_{10}(j,t) = \frac{n_0(j,t+1) - P_{00}(j,t)n_0(j,t)}{n_1(j,t)} \tag{2.14}$$

$$P_{01}(j,t) = 1 - P_{00}(j,t) \tag{2.15}$$

$$P_{11}(j,t) = 1 - P_{10}(j,t) \tag{2.16}$$

The remaining part is to define $P_{00}(j,t)$. We adopt the choice from [47] and start from $P_{00}(j,t=0) = 0.5$ and choose $P_{00}(j,t) = n_0(j,t)$ for the following generations $t > 0$. Thus for a locus $A_{ij}(t)$ we mutate it, changing the binary value, by the probability

$$p_{ij}(t) = \begin{cases} a_i(t)P_{01}(t), & A_{ij}(t) = 0 \\ a_i(t)P_{10}(t), & A_{ij}(t) = 1 \end{cases} \tag{2.17}$$

In summary, each generation update involves the following steps.

1. For generation t calculate the fitness score $f_i(t)$ of each chromosome i and sort the population matrix $A_{ij}(t)$ row-wise with descending score.

2. From the chosen ranking scheme $r_i(t)$ set the chromosome mutation probability to $a_i(t) = r_i(t)$ and the weighting of each row $W_i(t) = 1 - r_i(t)$.

3. Calculate the target states in Eq. (2.13) and the transition probabilities using Eq. (2.14) to (2.16) and $P_{00}(j,t=0) = 0.5$, $P_{00}(j,t>0) = n_0(j,t>0)$.

4. Mutate ($0 \to 1$ or $1 \to 0$) each locus $A_{ij}(t)$ by the probability p_{ij} given by Eq. (2.17).

Notice that this algorithm treats every locus as an independent gene, which means that we do not take the spatial dependencies in the Kirigami pattern into account.

Part II

Simulations

Appendices

Bibliography

[1] E. Gnecco and E. Meyer, *Elements of friction theory and nanotribology* (Cambridge University Press, 2015).

[2] Bhusnan, "Introduction", in *Introduction to tribology* (John Wiley & Sons, Ltd, 2013) Chap. 1.

[3] H.-J. Kim and D.-E. Kim, "Nano-scale friction: a review", International Journal of Precision Engineering and Manufacturing **10**, 141–151 (2009).

[4] K. Holmberg and A. Erdemir, "Influence of tribology on global energy consumption, costs and emissions", Friction **5**, 263–284 (2017).

[5] B. Bhushan, "Gecko feet: natural hairy attachment systems for smart adhesion – mechanism, modeling and development of bio-inspired materials", in *Nanotribology and nanomechanics: an introduction* (Springer Berlin Heidelberg, Berlin, Heidelberg, 2008), pp. 1073–1134.

[6] P. Z. Hanakata, E. D. Cubuk, D. K. Campbell, and H. S. Park, "Accelerated search and design of stretchable graphene kirigami using machine learning", Phys. Rev. Lett. **121**, 255304 (2018).

[7] P. Z. Hanakata, E. D. Cubuk, D. K. Campbell, and H. S. Park, "Forward and inverse design of kirigami via supervised autoencoder", Phys. Rev. Res. **2**, 042006 (2020).

[8] L.-K. Wan, Y.-X. Xue, J.-W. Jiang, and H. S. Park, "Machine learning accelerated search of the strongest graphene/h-bn interface with designed fracture properties", Journal of Applied Physics **133**, 024302 (2023).

[9] Y. Mao, Q. He, and X. Zhao, "Designing complex architectured materials with generative adversarial networks", Science Advances **6**, eaaz4169 (2020).

[10] Z. Yang, C.-H. Yu, and M. J. Buehler, "Deep learning model to predict complex stress and strain fields in hierarchical composites", Science Advances **7**, eabd7416 (2021).

[11] A. E. Forte, P. Z. Hanakata, L. Jin, E. Zari, A. Zareei, M. C. Fernandes, L. Sumner, J. Alvarez, and K. Bertoldi, "Inverse design of inflatable soft membranes through machine learning", Advanced Functional Materials **32**, 2111610 (2022).

[12] S. Chen, J. Chen, X. Zhang, Z.-Y. Li, and J. Li, "Kirigami/origami: unfolding the new regime of advanced 3D microfabrication/nanofabrication with "folding"", Light: Science & Applications **9**, 75 (2020).

[13] OpenAI, *Dall-e2*, (2023) `https://openai.com/product/dall-e-2`.

[14] Midjourney, *Midjourney*, (2023) `https://www.midjourney.com`.

[15] Z. Deng, A. Smolyanitsky, Q. Li, X.-Q. Feng, and R. J. Cannara, "Adhesion-dependent negative friction coefficient on chemically modified graphite at the nanoscale", Nature Materials **11**, 1032–1037 (2012).

[16] B. Liu, J. Wang, S. Zhao, C. Qu, Y. Liu, L. Ma, Z. Zhang, K. Liu, Q. Zheng, and M. Ma, "Negative friction coefficient in microscale graphite/mica layered heterojunctions", Science Advances **6**, eaaz6787 (2020).

[17] D. Mandelli, W. Ouyang, O. Hod, and M. Urbakh, "Negative friction coefficients in superlubric graphite–hexagonal boron nitride heterojunctions", Phys. Rev. Lett. **122**, 076102 (2019).

[18] R. W. Liefferink, B. Weber, C. Coulais, and D. Bonn, "Geometric control of sliding friction", Extreme Mechanics Letters **49**, 101475 (2021).

[19] M. Metzsch, *Github repository*, `https://github.com/mikkelme/MastersThesis`.

[20] A. P. Thompson, H. M. Aktulga, R. Berger, D. S. Bolintineanu, W. M. Brown, P. S. Crozier, P. J. in 't Veld, A. Kohlmeyer, S. G. Moore, T. D. Nguyen, R. Shan, M. J. Stevens, J. Tranchida, C. Trott, and S. J. Plimpton, "LAMMPS - a flexible simulation tool for particle-based materials modeling at the atomic, meso, and continuum scales", Comp. Phys. Comm. **271**, 108171 (2022).

[21] E. M. Nordhagen, *LAMMPS simulator*, https://github.com/evenmn/lammps-simulator.

[22] A. H. Larsen, J. J. Mortensen, J. Blomqvist, I. E. Castelli, R. Christensen, M. Dułak, J. Friis, M. N. Groves, B. Hammer, C. Hargus, E. D. Hermes, P. C. Jennings, P. B. Jensen, J. Kermode, J. R. Kitchin, E. L. Kolsbjerg, J. Kubal, K. Kaasbjerg, S. Lysgaard, J. B. Maronsson, T. Maxson, T. Olsen, L. Pastewka, A. Peterson, C. Rostgaard, J. Schiøtz, O. Schütt, M. Strange, K. S. Thygesen, T. Vegge, L. Vilhelmsen, M. Walter, Z. Zeng, and K. W. Jacobsen, "The atomic simulation environment—a python library for working with atoms", Journal of Physics: Condensed Matter **29**, 273002 (2017).

[23] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, A. Desmaison, A. Kopf, E. Yang, Z. DeVito, M. Raison, A. Tejani, S. Chilamkurthy, B. Steiner, L. Fang, J. Bai, and S. Chintala, "Pytorch: an imperative style, high-performance deep learning library", in *Advances in neural information processing systems 32* (Curran Associates, Inc., 2019), pp. 8024–8035.

[24] J. Lederer, "Activation functions in artificial neural networks: a systematic overview", (2021).

[25] P. Shankar, "A review on artificial neural networks", **3**, 166–169 (2022).

[26] A. Binder, *Lecture materials. in5400 - machine learning for image analysis*, [Online; accessed 08/05/2023], (2022) https://www.uio.no/studier/emner/matnat/ifi/IN5400/v22/lecture-materials/in5400_2022_slides_lecture2.pdf.

[27] N. C. Thompson, K. Greenewald, K. Lee, and G. F. Manso, "The computational limits of deep learning", (2022).

[28] J. G. Sam Lau and D. Nolan, *Principles and techniques of data science, 11.4. stochastic gradient descent*, [Online; accessed 08/05/2023], (2020) https://www.samlau.me/test-textbook/ch/11/gradient_stochastic.html.

[29] D. P. Kingma and J. Ba, "Adam: a method for stochastic optimization", (2017).

[30] T. Salimans and D. P. Kingma, "Weight normalization: A simple reparameterization to accelerate training of deep neural networks", CoRR **abs/1602.07868** (2016).

[31] K. He, X. Zhang, S. Ren, and J. Sun, "Delving deep into rectifiers: surpassing human-level performance on imagenet classification", IEEE International Conference on Computer Vision (ICCV 2015) **1502**, 10.1109/ICCV.2015.123 (2015).

[32] S. Ioffe and C. Szegedy, "Batch normalization: accelerating deep network training by reducing internal covariate shift", in Proceedings of the 32nd international conference on international conference on machine learning - volume 37, ICML'15 (2015), 448–456.

[33] L. N. Smith, "A disciplined approach to neural network hyper-parameters: part 1 – learning rate, batch size, momentum, and weight decay", (2018).

[34] B. Z. Cunha, C. Droz, A. Zine, S. Foulard, and M. Ichchou, "A Review of Machine Learning Methods Applied to Structural Dynamics and Vibroacoustic", working paper or preprint, Apr. 2022.

[35] S. Saha, *A comprehensive guide to convolutional neural networks — the eli5 way*, [Online; accessed 08/05/2023], (2018) https://towardsdatascience.com/a-comprehensive-guide-to-convolutional-neural-networks-the-eli5-way-3bd2b1164a53.

[36] L. Guerdan, *Diving into temporal convolutional networks*, [Online; accessed 08/05/2023], (2019) https://lukeguerdan.com/blog/2019/intro-to-tcns/.

[37] G. Cybenko, "Approximation by superpositions of a sigmoidal function", Mathematics of Control, Signals and Systems **2**, 303–314 (1989).

[38] Y. Bengio, "Practical recommendations for gradient-based training of deep architectures", in *Neural networks: tricks of the trade: second edition*, edited by G. Montavon, G. B. Orr, and K.-R. Müller (Springer Berlin Heidelberg, Berlin, Heidelberg, 2012), pp. 437–478.

[39] R. Holbrook and A. Cook, *Overfitting and underfitting*, [Online; accessed 08/05/2023], https://www.kaggle.com/code/ryanholbrook/overfitting-and-underfitting.

[40] JavaTpoint., *Gradient descent in machine learning*, [Online; accessed 08/05/2023], https://www.javatpoint.com/gradient-descent-in-machine-learning.

[41] R. R. Selvaraju, M. Cogswell, A. Das, R. Vedantam, D. Parikh, and D. Batra, "Grad-CAM: visual explanations from deep networks via gradient-based localization", International Journal of Computer Vision **128**, 336–359 (2019).

[42] S. Katoch, S. S. Chauhan, and V. Kumar, "A review on genetic algorithm: past, present, and future", Multimedia Tools and Applications **80**, 8091–8126 (2021).

[43] R Jiang, K. Szeto, Y. Luo, and D. Hu, "Distributed parallel genetic algorithm with path splitting scheme for the large traveling salesman problems", in Proceedings of conference on intelligent information processing, 16th world computer congress (2000), pp. 21–25.

[44] K. Szeto, K. Cheung, and S. Li, "Effects of dimensionality on parallel genetic algorithms", in Proceedings of the 4th international conference on information system, analysis and synthesis, orlando, florida, usa, Vol. 2 (1998), pp. 322–325.

[45] K. Y. Szeto and L. Fong, "How adaptive agents in stock market perform in the presence of random news: a genetic algorithm approach", in Intelligent data engineering and automated learning—ideal 2000. data mining, financial engineering, and intelligent agents: second international conference shatin, nt, hong kong, china, december 13–15, 2000 proceedings 2 (Springer, 2000), pp. 505–510.

[46] K. L. Shiu and K. Y. Szeto, "Self-adaptive mutation only genetic algorithm: an application on the optimization of airport capacity utilization", in Intelligent data engineering and automated learning–ideal 2008: 9th international conference daejeon, south korea, november 2-5, 2008 proceedings 9 (Springer, 2008), pp. 428–435.

[47] G. Wang, C. Chen, and K. Y. Szeto, "Accelerated genetic algorithms with markov chains", in *Nature inspired cooperative strategies for optimization (nicso 2010)*, edited by J. R. González, D. A. Pelta, C. Cruz, G. Terrazas, and N. Krasnogor (Springer Berlin Heidelberg, Berlin, Heidelberg, 2010), pp. 245–254.