Machine Learning for Exploration of Mechanical Structures Using Molecular Dynamics

Christer Dreierstad

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

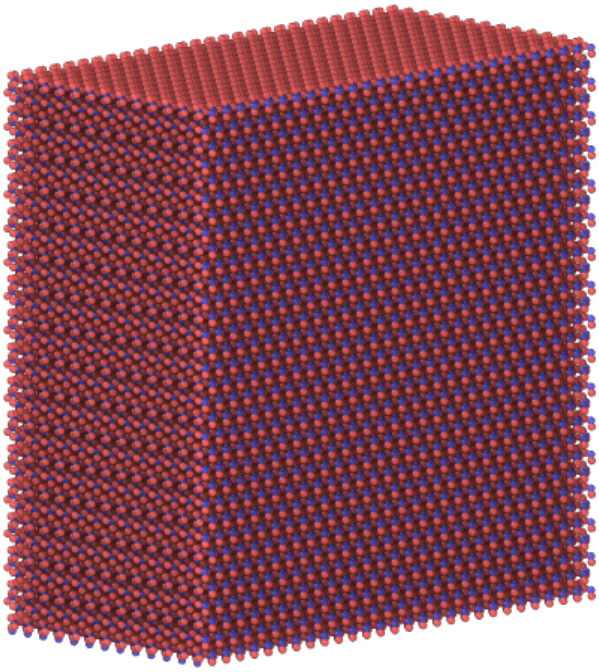
Department of Physics
Faculty of mathematics and natural sciences

UNIVERSITY OF OSLO

Spring 2021

Machine Learning for Exploration of Mechanical Structures Using Molecular Dynamics

Christer Dreierstad



Machine Learning for Exploration of Mechanical Structures Using Molecular Dynamics

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

Printed: Reprosentralen, University of Oslo

Abstract

In this study we found that we can predict the strength of α-quartz material structures using convolutional neural networks.

We performed molecular dynamics simulations on α-quartz for different material structures. We used a pseudo-random noise algorithm called Simplex noise to create different geometries. The material structures are made by removing particles in a specific region of the α-quartz system according to the geometry generated by the noise algorithm. During the molecular dynamics simulation we applied a shearing force on the α-quartz system until failure. We measured the yield shear stress at the first instance of failure in the material. The measured yield shear stress was found to be in the range of 0.5 – 3.2 GPa for different material structures.

We used supervised machine learning to create a model that predicts the yield shear stress given a geometry. The geometries were represented by two dimensional binary images. Each image is associated with value for yield shear stress found by molecular dynamics simulation. The model was made using a convolutional neural network, where the images served as the regressor and the yield shear stress as the predictor. The model achieved an R^2 score of 0.99, compared $R^2 = 0.86$ for a third degree polynomial. The model was used to predict the yield shear stress for millions of unseen images, which represents material structures. We chose a subset of the new geometries based on the predicted yield shear stress and simulated these with molecular dynamics simulations to find the yield shear stress. Initially the data set was relatively small, and then we iteratively increased it by generating new samples through prediction and molecular dynamics simulations. We found that the model could accurately predict the average yield shear stress for a chosen subset of structures when targeting a specific yield shear stress and porosity. The robustness of the method was tested by targeting different yield shear stresses and exposing the model to new types of geometries. This method can be used to design material structures with specific properties.

Contents

Acknowledgements

This has been a very tough time to be a student, and I am eternally grateful for the support and encouragement from my supervisors Anders Malthe-Sørenssen and Henrik Andersen Sveinsson, this has helped me to keep focused, motivated and inspired. I am very thankful for how the University of Oslo and the Centre for Computing in Science Education handled the pandemic, especially allowing master students early access to the university.

Thank you Nicolai for long walks, and for listening and engaging when I had the need to clear my head. This has been very helpful. Thank you Hans for the team work throughout the thesis.

Most importantly, thank you Marthe for supporting me, and especially for enabling me to focus fully on my thesis the last months. The lockdown would have been so much harder without you. I apologize for the times I have been cranky, and I thank you for all the times you have "uncranked" me. I cannot express how much I look forward to enjoying the summer with you!

Chapter 1

Introduction

1.1 Motivation

Controlling the strength of materials can be important for industrial applications. In some cases we might want to maximize the strength of a material, while at the same time keeping it as light as possible. Performing experiments for various material structures until you reach a desired strength can be costly and time consuming. By applying scientific methods we can create models that predict the strength of materials given a material structure. Such methods can reduce the cost and be time saving when creating materials with specific properties.

In this study we focus on searching through material structures using machine learning methods. We create different structures for a sample of α-quartz by removing regions in the system, resulting in the system becoming porous. The mechanical properties of the porous system depends on how we remove the particles. We aim to create a model that predicts mechanical properties as a function of the material structure.

Artificial neural networks is an up-and-coming branch of machine learning. Up until recent the computational power has not been strong enough to efficiently use these types of methods, or to build significant amount of data for certain applications, such as molecular dynamics. We are now seeing a massive exploration of applications for neural networks[1], as we are learning how versatile these methods are with applications within geosciences [7, 48], anomaly detection of energy consumption in buildings [59], applications related to Covid-19 [20, 23, 39, 53], forecasting price of electricity [32] and prediction of chloride diffusivity in concrete [45], to name a few applications. Convolutional neural networks is a branch of machine learning which performs regression on inputs structured in grids, often these grid-like inputs are images. It is often used in image classification, in which it has been shown that it can outperform humans for specific cases [18]. Convolutional neural networks works well when there are spatial dependencies in the images. In this study we will create a two dimensional representation of α-quartz structures. The methods

[1]We will refer to artificial neural networks as neural networks through this thesis. We will always refer to the artificial and not the biological neural network.

we use are similar to those proposed by Hanakata et al. [15] to predict mechanical properties of different structures of graphene. Graphene is a sheet of carbon atoms, which shows interesting mechanical properties for different structures. When subjected to a force pulling the graphene sheet in opposite direction it buckles perpendicular to the direction of the force.

Using neural networks for designing new material structures with specific properties can be an important tool. It can prove valuable for the industry when designing materials which need to have certain properties, or it can be used to study processes in fields such as geology and biology.

Molecular dynamics is a method for simulating movements of atoms in a system. The system can be modelled as a specific material. There exists multiple software packages for performing molecular dynamics simulations which aim at performing efficient calculations. Molecular dynamics can be used to predict mechanical properties for materials, by simulating a system under a mechanical load forward in time. Like neural networks molecular dynamics has greatly benefited from the increase of computational power. Today we can simulate systems up to 100 nm cubes, considering both mechanical and chemical processes, using reactive molecular dynamics potentials to describe the interactions between the atoms. For a system of 100 000 particles and a simulation time of 1.5 ns we can expect a run time of about 10 minutes on modern hardware, this makes it feasible to run thousands of simulations, both for creating data and to verify the results of a model.

The material we are studying is α-quartz, a brittle crystalline silicone dioxide. We vary the material structure and subject the system to a shearing force until material failure. By removing particles from regions of the system we are creating local local points of weakness, as well as a global weakness as we are making the material porous. The strength of the system depends on how we remove particles, and how many particles we are removing. Since there is some dependence on how we remove particles, we want to explore if a convolutional neural network can learn these through regression on images representing the varying material structures. Hanakata et al. [15] showed that this is possible for a graphene kirigami.

The method we apply for creating material structures, called Simplex noise, resembles contours we can find in nature. Simplex noise is a successor of the popular noise algorithm Perlin noise. Simplex and Perlin noise is often used for creating terrain in computer games, and the algorithms are implemented in modern licensed and open source computer game engines.

1.2 Objective of Study

The aim of this study was to simulate a thousands of different material structures using LAMMPS[2]. These structures are to be used as the foundation for creating a machine learning model which accurately predict

[2]LAMMPS is a molecular dynamics simulator, which will be presented in section 3.2.

a physical property of the material, given a material structure. The goals are presented in a fuller format below.

1. Create a forward model for a specific material using LAMMPS, allowing for different material structures. Extract relevant physical properties.
2. Simulate hundred to thousands of systems and collect and store the resulting data.
3. Design and train a convolutional neural network based on the simulated data. The input to the network is the initial structure on a sufficiently coarse-grained level, and the output is the effective material properties of interest.
4. Use the trained network to scan the space of possible structures.
5. Evaluate the structures found and test predictions of the neural network by direct simulations.
6. Modify the methods or design improved initial sampling set in order to provide as good output as possible.

1.3 My Contributions

The goals for the study were met, and we also touched upon some other subjects during our exploration of different methods.

In this study we have created a model which predicts the strength of various material structures of α-quartz. We extracted the yield shear stress as the physical property to build our machine learning method on. We have explored various sampling methods for creating material structures, and we ended up with a pseudo-random noise algorithm that creates structures that are familiar to what we find in nature.

The initial material structures were represented at a coarse-grained level. We found no noticeable effects of using coarse-grained representations of the material structures, compared to using fine-grained representations.

The model was built by considering a fairly small data set, where we had to consider how we sampled a very large space of possible material structures. This type of material structure exploration can be cost effective and time efficient method for designing materials with specific properties, as we do not have to consider large amounts of data for the method to yield good results.

We found that we can design material structures under the criterion that they have a specific strength, and we found that we can also set a criterion on the porosity of the system while predicting the strength.

When testing different convolutional neural networks we found that certain methods seem to be more in line with observed physics than others.

We found that the system follows a well known theory concerning fracture of brittle materials. The system is initialized with a penny shaped crack of varying radii. The theory concerns a force which is pulling on the system until fracture, while our study considered a force which is shearing the system.

The model was found to have decent performance when we tested the trained model on material structures which were made using other methods for creating noise.

1.4 Structure of Thesis

The thesis is divided in three parts, covering I) theory and methods, II) results, implementation and discussion, III) summary and conclusions and an appendix.

The first part will introduce some relevant topics before introducing the methods which were used for the molecular dynamics simulations and machine learning. For the machine learning part we have chosen to include the methods which are not only a part of the final model, but also methods that were explored. The exploration of machine learning methods has played a part in how we arrived at the final model. We will also introduce some concept that are relevant for generating and processing of the data.

The second part presents and discuss our results and finding throughout this study. We have chosen to present the results in a way that we believe outlines the thought process and evolution of arriving at our final machine learning model, our choice of material and how we decided to create the material structures.

The third part discusses some of the findings in this study and puts them in perspective, and we discuss some some potential future work.

The appendix contains some additional results we did not consider important enough to present in part II.

Part I

Theory and Methods

Chapter 2

Background

The machine learning model will predict the strength of a specific material structure of α-quartz. We will remove regions of the system, making it porous. To find the strength of the system we perform molecular dynamics simulations, subjecting the system to an outer force until failure.

In the following chapters we will introduce some topics that will be relevant when discussing molecular dynamics, the target value for our machine learning model and when discussing the strength of our system, for various material geometries.

In the process of writing this chapter we used the following literature Buehler [6], Anderson [4], Zehnder [57], Gdoutos [11] and Timoshenko [51] to gather information about Griffith's theory, stress and strain. Other literature were used to gather specific information, this will be stated appropriately in the text.

2.1 Porosity

If a medium consists of one or more regions that are unoccupied by the medium itself, then it is considered a porous medium. We will refer to the unoccupied regions as voids, in some branches of natural sciences these voids are called pores. There can be one or more such voids contributing to the total void of the porous medium. Porosity is a measurement of how porous a medium is, and it is the relative volume of the total void, defined as

$$\phi = \frac{V_{void}}{V_{total}}, \tag{2.1}$$

If there are more than one voids, then V_{voids} is the sum of volumes of the voids.

2.2 Stress

When a system is perturbed by an outer force which causes a deformation of the body, internal forces between particles arise. Deformation of a body can arise due to compression, twisting or pulling in various ways,

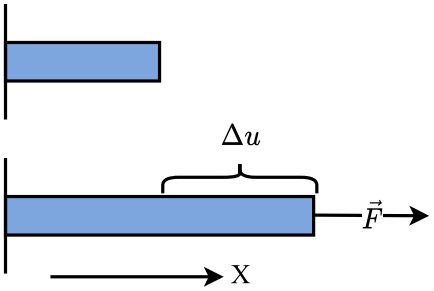


Figure 2.1: Visualization of deformation of a rod due to a force **F**, which acts along the direction the rod is elongated. Δu is the elongation of the rod.

all potentially causing the particles inside the body to move relative to one another, giving rise to internal forces. Stress is the magnitude of the external force acting on a body per unit area

$$\sigma = \frac{F}{A}\,. \tag{2.2}$$

There are two types of stresses; normal and shearing stress. Normal stress can be either tensile and compressive, which is simply put the opposite of one another. Tensile stress is a result of applying a force that pulls on the system, increasing the length along the direction of the force. In Figure 2.1 we can see an example of elongation of a rod due to a force pulling on the rod. Compressive stress is a result of forces compressing the body, decreasing the length along the direction of the applied force. If we consider an isotropic system that is subjected to a force that is stretching the system uniformly along a surface normal, the force will act continuously over a cross section with the same surface normal.

2.2.1 Components of Stress

As discussed above we have different types of stress, which are defined by how the force is acting on the body. The components of stress can be expressed as

$$\sigma_{ij} = \frac{F_j}{A_i}\,, \tag{2.3}$$

where A_i is the area of the surface with normal n_i, e.g. A_x has surface normal n_x, and associated area $A_{yz} = L_y L_z$. In Figure 2.2 we can see the components of stress. *Tensile stress* is a measure of an external force acting along the normal of a cross-sectional area, and it is denoted σ. There are three components of tensile stress σ_i where $i = x, y, z$. In Figure 2.1 we can see an example of a rod subjected to tensile stress. *Shearing stress* is a measure of an external force acting in a cross-sectional area, and it is denoted τ. Due to conservation of angular momentum we have, the shear stress components are symmetric, i.e. $\tau_{ij} = \tau_{ji}$ for $i, j = x, y, z$ [51, p. 5]. In Figure 2.3 we can see the components of shear stress on a square, while in Figure 2.4 we can see an example of shear deformation with $\tau_{yx} > 0$

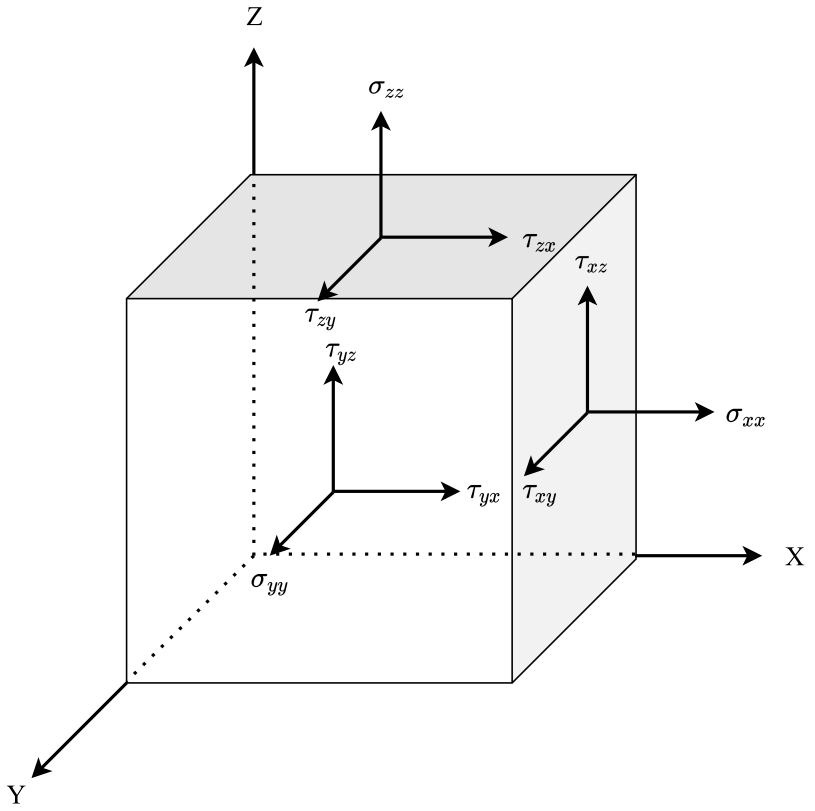


Figure 2.2: Visualization of the positive components of stress σ_{ij}, $i, j = x, y, z$.

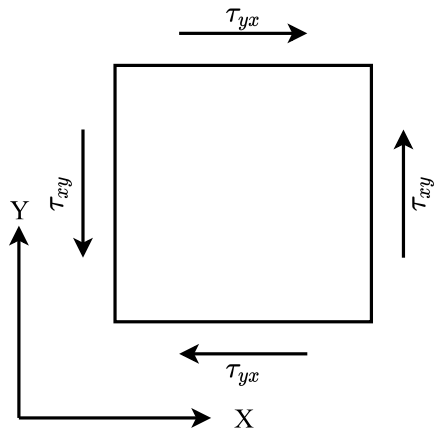


Figure 2.3: Visualization of components of shear stress for a two dimensional case.

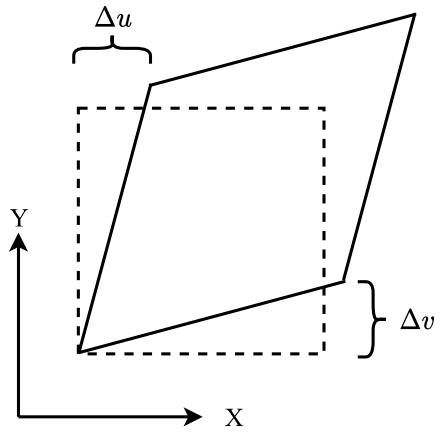


Figure 2.4: Visualization of a box (dotted) that has been subjected to shear deformation. Δu and Δv shows the elongation, along axes x and y, respectively. The elongation is also called the tilt factor.

Yield Stress

During loading the system might arrive at a point where the material is changed in a non-reversible way, this is known as *yield stress*. Depending on the system, yield may not be the same as the ultimate strength of the material. We define the strength of a material as the point where the material yields due to being subjected to an outer force, causing the material to deform, the force per area that causes the system to yield is the yield stress. The ultimate strength is the point of failure for the system, whereas yield strength could be a point where the system has a local failure that preserves the overall integrity of the body. In Figure 2.5 we can see an example of a stress-strain plot, also known as a loading curve. The yield stress is found as the (first local) maxima.

2.3 Strain

Deformation of a body that is subjected to a stress, which results in an extension of the body in some way, is measured in strain. Strain is defined as

$$\epsilon = \frac{\text{extension due to deformation}}{\text{length prior to deformation}} = \frac{\Delta u}{L}, \tag{2.4}$$

for some extension Δu and length L of the body. In Figure 2.1 and 2.4 shows the Δu for the case of tensile and shear strain, respectively.

2.3.1 Components of Strain

We define the displacement of particles in a body as u, v and w, along the axes x, y and z, respectively. For the case of *tensile strain* elongation defined as the change in the length of the body along the force. Tensile strain is

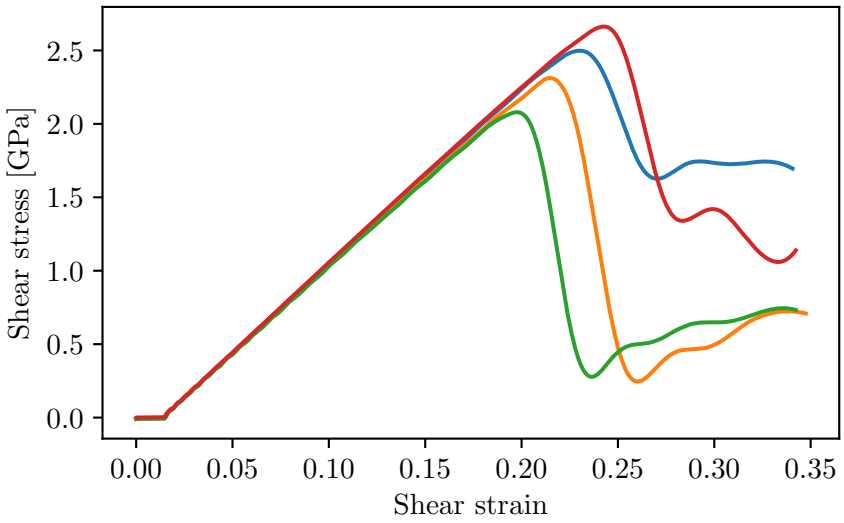


Figure 2.5: Loading curves for five random geometries. Shear stress was smoothed with Savitzky-Golay filter. Yield shear stress is the global maxima in these samples.

denoted ϵ, and the components are given as

$$\epsilon_x = \frac{\partial u}{\partial x}, \quad \epsilon_y = \frac{\partial v}{\partial y}, \quad \epsilon_z = \frac{\partial w}{\partial z}. \tag{2.5}$$

The equations above explain the change in elongation for a given direction. *Shear strain* is not as easily defined as tensile strain. The body can be deformed in such a way that a point A located in the body is displaced along two axes at once, for a single strain component. Figure 2.4 shows a scenario where there is elongation along x and y axes. Shear strain is denoted γ and is given by

$$\gamma_{xy} = \frac{\partial u}{\partial y} + \frac{\partial v}{\partial x}, \quad \gamma_{xz} = \frac{\partial u}{\partial z} + \frac{\partial w}{\partial x}, \quad \gamma_{yz} = \frac{\partial v}{\partial z} + \frac{\partial w}{\partial y}. \tag{2.6}$$

2.4 The Griffith Theory of Brittle Fracture

Griffith's theory concerns crack growth in a brittle material that is subjected to tensile loading. Griffith studied the relationship between strength and the size of the largest flaw in a system, he proposed that the greatest stress intensity was related to the largest flaw. The experiments Griffith carried out was done by creating an elliptical through-width crack in a sample of glass, and then subjecting it to tensile loading until failure. It was found that the yield stress was inversely proportional to $\sqrt{a}$, where a is the half length of the induced flaw (hereby called crack), as discussed by Griffith [13].

When applying load to a material the tension between the atoms increases. If we increase the load to a given point we will observe that atomic bonds start to break, which leads to the material cracking. When a crack propagates the material is deformed in a non-reversible fashion, which we know to be the point of yield, as discussed in section 2.2.1. Upon loading we are increasing the elastic energy of the system, and when the system fails some energy stored in the system is spent creating new surfaces Buehler [6, p. 189-190]. We consider an initial crack with area A.

The Griffith energy balance states that during a change in the crack area dA, the change in energy is

$$\frac{dU}{dA} = \frac{dW_p}{dA} + \frac{dW_s}{dA} = 0\,, \tag{2.7}$$

which can also be written

$$-\frac{dW_p}{dA} = \frac{dW_s}{dA}\,, \tag{2.8}$$

where W_p is the potential energy that arises due to loading (internal strain energy) has the same magnitude as W_s, the work needed to create new surfaces. W_s is related to area of the crack and the surface energy γ_s, which is a material specific constant. For an initial crack of area A W_s takes the form

$$W_s = 2A\gamma_s\,. \tag{2.9}$$

The factor 2 in the equation above comes from the fact that as a crack grows, there are two new surfaces being created. We can interpret W_s as a threshold for crack growth which the potential energy has to surpass in order for crack growth.

As previously mentioned Griffith found that the yield stress was inversely proportional to $\sqrt{a}$. For an initial crack area of $2aB$, where B is the width of the crack, using the solution $W_p = W_0 - \frac{\pi\sigma^2 a^2 B}{E}$ as presented by Anderson [4, p. 29-30] based on the work of Griffith [13], for the potential energy, it was found that the failure stress followed

$$\sigma_f = \sqrt{\frac{2E\gamma_s}{\pi a}}\,, \tag{2.10}$$

where E is Youngs' modulus. We will use Griffith's theory to study if our system, being deformed by shearing, follows a well known theory for brittle fracture as a result of tensile stress.

Chapter 3

Molecular Dynamics

Molecular dynamics (MD) is a method that approximates the equations of motion for a set of atoms by assuming they interact through classical potential functions and move according to Newtons 2nd law. We will refer to the atoms and molecules as particles. The trajectories of the particles are calculated numerically by solving Newtons equations of motion. Depending on the system we wish to study, the equations of motion takes on different forms. In some cases we only need the force acting on the particles to decide the trajectories, in other cases we need a set of modified equations of motion, which serve a specific purpose, as we will see below.

The total energy of a system can be described as the sum of the kinetic energy K and potential energy U

$$E = K + U\,. \tag{3.1}$$

The kinetic energy of the system is defined as the sum of the mass and velocity of each particle, multiplied by $1/2$

$$K = \frac{1}{2}\sum_{i}^{N} m_i \mathbf{v}_i^2\,. \tag{3.2}$$

The choice of potential energy function is critical for our simulations to yield results that stay as close to what we would expect in an experiment.

This chapter is mainly based on the following literature Timoshenko [51], Buehler [6], Frenkel [10] and Hünenberger [24].

3.1 Potentials

How the particles interact is decided through the function we choose for the potential energy. In the simplest case we can use a potential function which decides the repulsive and attractive forces between two particles, as a function of distance. The Lennard-Jones (LJ) potential

$$U_{LJ}(r_{ij}) = 4\epsilon\left[\left(\frac{\sigma}{r_{ij}}\right)^{12} - \left(\frac{\sigma}{r_{ij}}\right)^{6}\right], \tag{3.3}$$

contains pairwise repulsive and attractive forces between particle i and j. The first term in the brackets above decides the strength of the repulsion, while the latter decides the strength of the attraction. The LJ potential models the basics of interacting particles, they attract each other at long range and repel each other at short range.

In recent time the increase in computational power has made it possible to study increasingly larger system containing an increasing amount of particles. We can also consider more complex potential functions that considers interactions between more than two particles, bonded interactions and chemical reactions.

3.1.1 AIREBO

For the graphene study we apply the same potential as Hanakata et al. [15] did in their study. Adaptive Intermolecular Reactive Empirical Bond Order (AIREBO) was proposed by Stuart et al. [49] as an extension to the REBO potential, which does not include terms for non-bonded repulsion and dispersion. The AIREBO potential

$$U = \frac{1}{2}\sum_{i}\sum_{i\neq j}\left[U_{ij}^{REBO} + U_{ij}^{LJ} + \sum_{k\neq i,j}\sum_{l\neq j,k} U_{kijl}^{tors}\right], \tag{3.4}$$

consists of three terms:

1. U_{ij}^{REBO} – energy from inter- and intramolecular interactions
2. U_{ij}^{LJ} – energy due to Lennard-Jones interactions
3. U_{kijl}^{tors} – energy due to torsional/dihedral interactions

The subscripts indicate interactions between particles i and j, and for the latter i, j, k and l.

3.1.2 Vashishta

For the SiO_2 part we considered the Vashishta potential proposed by Vashishta et al. [52] in 1990. In 1994 Nakano et al. [37] proposed a different two-body term for the Coulomb interaction, adding an exponential decay of distance over a given decay length. For the amorphous silica we used the parameters from the paper published in 1994, which were established looking at porous amorphous silica. For the alpha quartz system we used the parameters from a paper published in 1997 by Broughton et al. [5], which were established looking at α-quartz. The Vashishta potential consists of a two terms

$$U = \sum_{i<j} u_{ij}^{(2)}(r_{ij}) + \sum_{i,j<k} u_{jik}^{(3)}(\boldsymbol{r}_{ij}, \boldsymbol{r}_{ik}) \tag{3.5}$$

where $u_{ij}^{(2)}$ is the potential energy from two-body interactions between particles i and j, while $u_{jik}^{(3)}$ is for the three-body interactions between

particles i, j and k. r_{ij} is distance between particles i and j. The two-body term is

$$u_{ij}^{(2)}(r_{ij}) = \frac{H_{ij}}{r^{\eta_{ij}}} + \frac{Z_i Z_j}{r_{ij}} e^{-r_{ij}/\lambda} - \frac{\alpha_i Z_j^2 + \alpha_j Z_i^2}{2r_{ij}^4} e^{-r_{ij}/\xi}\,, \tag{3.6}$$

where the first term is due to short range repulsion,

$$H_{ij} = A\left(\sigma_i + \sigma_j\right) \tag{3.7}$$

is the strength, and η_{ij} the exponent, of the steric repulsion. σ_i is the ionic radius of particle i. The second term is the Coulomb interaction, where Z_i is the effective charge of particle i. The last term is due to charge induced dipole interactions, where α_i is the electronic polarizability of particle i and λ and ξ are the decay lengths. The three-body term

$$\begin{aligned} u_{jik}^{(3)}(\boldsymbol{r}_{ij}, \boldsymbol{r}_{ik}) = B_{jik} \exp\left(\frac{1}{r_{ij} - r_0} + \frac{1}{r_{ik} - r_0}\right) \left(\frac{\boldsymbol{r}_{ij} \cdot \boldsymbol{r}_{ik}}{r_{ij} r_{ik}} - \cos\bar{\theta}_{jik}\right)^2 \\ \times \Theta\left(r_0 - r_{ij}\right) \Theta\left(r_0 - r_{ik}\right)\,, \end{aligned} \tag{3.8}$$

where Θ is the Heaviside function, B_{jik} is strength of the three-body interaction. H_{ij}, B_{ijk}, λ, Z_i, η, α, r_0, ξ are parameters which are specific to the system (e.g. SiO_2).

3.2 LAMMPS Molecular Dynamics Simulator

LAMMPS – Large-scale Atomic/Molecular Massively Parallel Simulator is a free open-source software which performs molecular dynamics simulations efficiently, and is well optimized for parallel computing. With a few simple lines of code, and some basic understanding of MD, one can perform MD simulations, which makes it a very accessible tool.

To create a LAMMPS script we need to declare a system and its state, before we decide on a set of rules for integrating the equations of motion. To create a system of particles we first create a so-called simulation box of a given size, which is essentially the domain for our MD simulation. We then decide on the positions for our particles, which is typically done by defining a lattice, which defines the coordinates for the particles inside the simulation box. When simulating a specific material we decide the lattice by the unit cell of a for the material. Units cells will be discussed in the next section. The atoms are then placed according to the lattice. When the particles are placed we can decide the velocities of the particles. The velocities are set by choosing an initial temperature of the system, following the equipartition principle, which relates the average kinetic energy, hence the average velocity, of the particles with the temperature. At this point we have a system containing particles displaced according to a given unit cell. We then choose a specific potential function and decide the ensemble, i.e. the equations of motion, we wish the system to follow.

3.3 Integrating Equations of Motion

In molecular dynamics we integrate Newton's equation of motion

$$\boldsymbol{a}_i = \frac{\boldsymbol{F}_i}{m} \tag{3.9}$$

to obtain the position and velocity of particle i given a force $\boldsymbol{F}_i$ which is acting upon the particle. The force is described by our choice of potential function, which will be discussed in sec. 3.1. The force is $\mathbf{F}_i$ is found by $\mathbf{F}_i = -\nabla U_i$, where U_i is the potential function, hence $\mathbf{F}$ is a conservative force. Since the force acting on the system is conservative, the total energy of the system is conserved, ignoring thermostats and barostats. Since thermostats and barostats change the system in some way, they are performing work on the system. When integrating we would ideally use an algorithm that preserves energy both long- and short-term, but no such algorithm exists, as stated by Frenkel [10, p. 72]. The precision of our calculations are bounded by how many decimals the computer is able to represent a number by, which is known as machine precision. As will be discussed in sec 3.11 we can influence the precision of our calculations by adjusting the time step. A large time step results in lower computational time, as there are fewer calculations pr. total running time, but the calculations may end up with a poor precision. With a large time step we may not catch particles until they are very close to each other, which may result in a large local increase of particle velocity due to intermolecular interaction. By choosing a reasonably low time step we can increase the precision at the cost of increased computational time. By precision of calculations we mean by how much does the calculated trajectory deviate from the true trajectory, the true trajectory being the exact solution of Eq. (3.9). Verlet algorithm is known to have moderate short-term energy conservation, while the long-term energy drift is small Frenkel [10, p. 72]. The default integrator in LAMMPS is Velocity Verlet.

3.3.1 Velocity Verlet

When choosing the algorithm for integrating the equations of motion we can influence the short- and long-term energy drift. Velocity Verlet is accurate in position and velocity up to $\mathcal{O}(\Delta t^4)$ [10, p. 76], it preserves energy well and is time reversible. Velocity Verlet updates the particle positions, given initial velocities and an expression for the acceleration, found by Eq. (3.9). The velocities are calculated at half time step, which is used to calculate the velocities at full time step. Below is the equations in the order of which they are calculated. The equations are as follows

$$\mathbf{r}(t + \Delta t) = \mathbf{r}(t) + \mathbf{v}(t)\Delta t + \mathbf{a}(t)\frac{\Delta t^2}{2} \tag{3.10}$$

$$\mathbf{v}(t + \frac{\Delta t}{2}) = \mathbf{v}(t) + (\mathbf{a}(t) + \mathbf{a}(t + \Delta t))\frac{\Delta t}{2} \tag{3.11}$$

$$\mathbf{v}(t + \Delta t) = \mathbf{v}(t + \frac{1}{2}\Delta t) + \mathbf{a}(t + \Delta t)\frac{\Delta t}{2}\,, \tag{3.12}$$

where we insert $\mathbf{a}(t) = \mathbf{F}(t)/m$.

3.4 Thermostat

During the previous section we arrived at a set of equations for updating the positions and velocities by using Newtons second law, given an expression for the force acting upon the particles. This integration scheme keeps the energy E and volume V of the system, as well as the number of particles N, constant. This is called the microcanonical ensemble . We assert no control over the pressure or temperature. In some cases we want to increase or decrease the temperature, such as when we are melting and cooling α-quartz to create amorphous silicate. Often we use thermostats to relax a system at given temperature, also known as thermalization.

The equipartition principle states that the average kinetic energy is related to the temperature of a system, it is given by the following equation

$$K = \langle K \rangle = \frac{1}{2} N_f k_B T \,. \tag{3.13}$$

We can solve the equation above the system temperature T, giving us a measure of the instantaneous temperature. N_f in the above equation is the number of degrees of freedom. We see that we can influence the temperature by putting a constraint on the velocities, which are present in the term for kinetic energy. The simplest form of a thermostat is essentially a scaling of the particle velocities by some parameter

$$\mathbf{v}_i = \lambda \mathbf{v}_i \,. \tag{3.14}$$

We consider the system to be connected to some heat bath, which has a constant temperature, resulting in the system temperature converging towards the temperature of the heat bath, as we allow for energy to be transported between the system and the heat bath. The scaling factor is given by

$$\lambda = \sqrt{1 + \frac{\Delta t}{\tau} \left(\frac{T}{T_0} - 1 \right)} \,, \tag{3.15}$$

where T_0 is the temperature of the heat bath, Δt is our time step and τ is a parameter that describing the strength of the coupling to the heat bath. In practice we set the temperature of the heat bath to the temperature we wish the system to stabilize at. This scheme for thermalization is known as the *Berendsen thermostat*, which has not been used in this study, but it serves well as an introduction to thermostats, due to its simplicity. Berendsen thermostat does not produce results that are consistent with the canonical ensemble. A better approach will be discussed in the coming section.

3.4.1 Nosé-Hoover Thermostat

A different method for thermalization is done by a so-called extended system method. The extended system method was proposed by Andersen

[3], and it laid the ground work for methods such as proposed by Nosé [38] and Hoover [22] for controlling the temperature, as well as methods for controlling the pressure, which we will see below.

The idea behind the Nosé-Hoover thermostat is to add a degree of freedom to the system, which has a velocity and an associated "mass". We can think of this as adding a single particle to the system, which interacts with the particles of the physical system. For each interaction between particle in the extended system and the particles in the physical system, the particles in the physical system has their velocity perturbed in such a way that the temperature of the approaches the desired temperature, i.e. the temperature of the heat bath. The magnitude of the "mass" decides the strength of the coupling to the external system, which can be considered as a heat bath, a concept we are familiar with. Nosé [38] proposed a modified Lagrangian, adding an extra degree of freedom s and two terms to the Lagrangian[1] . There were also virtual variables introduced

$$\mathbf{r}_i' = \mathbf{r}_i \,, \tag{3.16}$$

$$\mathbf{p}_i' = \frac{\mathbf{p}_i}{s} \,, \tag{3.17}$$

$$t' = \int \frac{1}{s} dt \,, \tag{3.18}$$

where the variables on the left hand side are the real variables, marked with a prime. The Lagrangian then reads

$$\mathcal{L} = \frac{1}{2} \sum_{i=1}^{N} m_i s^2 \dot{\mathbf{r}}_i^2 - \mathcal{U}(\mathbf{r}^N) + \frac{1}{2} Q \dot{s}^2 - g k_B T_0 \ln s \,, \tag{3.19}$$

where Q is the parameter associated with the "mass" of s, T_0 is the temperature of the heat bath and g are the degrees of freedom for the physical system, i.e. not including the degree of freedom s, which is connected to the extended system. The third and forth terms in the Lagrangian above are the kinetic and potential energy associated with s. The energy associated with s is essential for the method to produce the canonical ensemble. $\mathcal{U}$ is the potential energy of the system, while the first term is the kinetic energy. The magnitude of Q decides the strength of the coupling to the heat bath. If again think of the particles of the extended and the physical system colliding, we can easily understand that increasing the "mass" Q leads to quicker convergence to the desired temperature of the heat bath.

Hoover [22] showed that the equations from Nosé [38] can be simplified by adding a thermodynamic friction coefficient $\xi = s' p_s'/Q$. We see from Eq. (3.18) that for the virtual variables we have a varying time step, for simulations it is not recommended to use a varying time step, and we therefore use the real variables. The equations of motion for the real

[1]Lagrangian $\mathcal{L}$ is an energy equation, $\mathcal{L}(r, \dot{r}) = K(\dot{r}) - U(r)$, where K is the kinetic and U is the potential energy. This method of thermalization is called the extended-system method,

variables, with the simplifications made by Hoover [22] are

$$\dot{\mathbf{r}}_i = \frac{\mathbf{p}_i}{m_i}\,, \tag{3.20}$$

$$\dot{\mathbf{p}}_i = -\frac{\partial \mathcal{U}(\mathbf{r}^N)}{\partial} - \xi \mathbf{p}_i\,, \tag{3.21}$$

$$\dot{\xi} = \left(\sum_i \frac{p_i^2}{m_i} - \frac{L}{\beta}\right)\frac{1}{Q}\,, \tag{3.22}$$

$$\frac{\dot{s}}{s} = \frac{d\ln s}{dt} = \xi\,. \tag{3.23}$$

3.4.2 Langevin Thermostat

Langevin thermostat is another method for controlling the temperature via scaling of the particle velocities, but in a different way than we saw for the Berendsen thermostat. The scaling is done by a modification to the equations of motion, adding two terms to Newtons second law. The Langevin equation is given by

$$\mathbf{a}_i(t) = \frac{\mathbf{F}_i(t)}{m_i} - \gamma_i(t)\mathbf{v}_i(t) + \frac{\mathbf{R}_i(t)}{m_i}\,, \tag{3.24}$$

where $\mathbf{R}_i$ is a stochastic force and γ_i is a positive atomic friction coefficient. The stochastic force has the following properties, as listed in Hünenberger [24], $\mathbf{R}_i$ for particle i is uncorrelated with particle velocity $\mathbf{v}_i(t')$ and force $\mathbf{F}_i(t')$ for $t' < t$, the time average of $\mathbf{R}_i$ is zero; $<\mathbf{R}_i> = 0$ and it must follow

$$\langle R_{ik}(t) R_{jl}(t')\rangle = 2m_i\gamma_i k_B T_0 \delta_{ij}\delta_{kl}\delta(t'-t)\,, \tag{3.25}$$

where T_0 is temperature of the heat bath, $\delta_{..}$ is the Kronecker Delta function, and $\delta(\cdot)$ is the Dirac Delta function. The equation above tells us that the mean square components of $\mathbf{R}_i$ evaluate to $2m_i\gamma_i k_B T_0$ and that the component $R_{ik}(t)$ along axis k is uncorrelated with $R_{jl}(t')$ along axis l, unless $i = j$, $k = l$ and $t' = t$.

3.5 Barostat

To regulate the pressure of a system in an MD simulation one most often changes the volume of the system. Barostatting is a normal procedure for relaxing a system, or to allow the system to undergo thermal expansion or contraction during a heating or cooling process. During the simulation where we are creating amorphous silica we allow for thermal contraction as the system enter the temperature range where it solidifies.

Parrinello and Rahman [40] proposed a modified Lagrangian to allow for change in the shape and size of the simulation box during MD simulations. This laid the ground work for the equations of motion proposed by Martyna et al. [36], which follows the extended system

approach by Andersen [3]. The equations of motions are given as

$$\mathbf{v}_i = \frac{\mathbf{p}_i}{m_i} + \frac{p_\epsilon}{W}\mathbf{r}_i\,, \tag{3.26}$$

$$\mathbf{a}_i = \frac{\mathbf{F}_i}{m_i} - \left(1 + \frac{d}{dN}\right)\frac{p_\epsilon}{W}\mathbf{v}_i - \frac{p_\xi}{Q}\mathbf{v}_i\,, \tag{3.27}$$

$$\dot{V} = \frac{dVp_\epsilon}{W}\,, \tag{3.28}$$

$$\dot{p}_\epsilon = dV\left(P_{int} - P_{ext}\right) + \frac{d}{N_f}\sum_{i=1}^{N}\frac{\mathbf{p}_i^2}{m_i} - \frac{p_\xi}{Q}p_\epsilon\,, \tag{3.29}$$

$$\dot{\xi} = \frac{p_\xi}{Q}\,, \tag{3.30}$$

$$\dot{p}_\xi = \sum_{i=0}^{N}\frac{\mathbf{p}_i^2}{m_i} + \frac{p_\epsilon^2}{W} - \left(N_f + 1\right)k_BT\,, \tag{3.31}$$

$$\epsilon = \ln\left(\frac{V}{V(t=0)}\right)\,. \tag{3.32}$$

In the above equations P_{int} is the internal pressure and P_{ext} is the external pressure imposed on the system. The added degree of freedom which was introduced in Nosé-Hoover thermostat is related to ξ, which is virtual variable with associated "mass" Q. The variables ϵ, p_ϵ and W are related to the now introduced barostat, where W is an associated "mass" of ϵ.

3.6 Materials

To construct materials for a molecular dynamics simulations we must consider how the particles that make up the material are distributed. The unit cell describes how particles are distributed for the smallest possible three dimensional representation of a given material. By stacking unit cells we can create a system of a given size, as discussed in the creation of a system in the section above. To build a unit cell we require the length in all three directions, the angle between the axes of the unit cell, what type of atoms should be present and where they should be distributed.

3.6.1 Graphene

Graphene is a one unit cell thick sheet of carbon atoms. The carbon atoms are placed in a hexagonal lattice. When graphene is subjected to deformation it shows interesting behavior in the form of buckling in the direction that is normal to the applied force. This out-of-plane buckling has a big impact on the strength of the sample, as shown by the simulations of Hanakata et al. [15], where the yield stress was ranging from $\approx 10 - 110$ GPa yield stress.

3.6.2 Alpha-quartz

Quartz is one of the most common minerals on Earth, and it has been widely studied. There are two forms of quartz, namely α-quartz and

β-quartz. The transition from $\alpha \rightarrow \beta$ happens at 846K at atmospheric pressure. In this study we focus on α-quartz, a triclinic crystal. The temperature we perform the simulations at are 300K. To initiate the system we used molecular-builder[2], a Python package for building MD systems. Molecular-builder also has functionality for carving system to create voids or surfaces. We initiate the α-quartz system to be 140.77Å $\times$ 143.68Å $\times$ 71.74Å. With a unit cell size of 5.02Å $\times$ 5.02Å $\times$ 5.51Å we have $28 \times 28 \times 13$ unit cells in the system.

3.7 Radial Distribution Function

By looking at how particles are distributed throughout a system we can determine if a system is in a solid, liquid or gaseous state. The radial distribution function (RDF) $g(r)$ tells the probability of finding a particle at a distance r inside a shell with thickness dr, which can be seen in Figure 3.1. For three dimensions the shell is spherical, while for two dimensions it is circular. If we consider a crystalline system, we know that the distance between the particles are statistically fixed, and we therefore expect to only find particles at a given distance from one another. By statistically fixed we mean if we had time averaged the small movement of the particles, for the rest of this section we will not explicitly state that there is a time averaging. For non-crystalline solids the particles are unevenly distributed across the body, which causes the particles to vibrate more than for crystalline systems. For crystalline systems the distances are constant which results in minor particle vibrations. In Figure 3.2 we can see RDF for amorphous silica. We observe that for 300K there are peaks at certain distances, characterizing a solid, but they are not as sharp as for the crystalline system. At 4000K the system the RDF is smoothed out, meaning the particles are not fixed at certain position, but can move around, we can characterize the system as being in a liquid state. 1500K shows peaks at given distances, but they are not as prominent as for 300K. The first peak is due to the first minima of the potential function, and it describes the average shortest distance between the particles. For solids the particles have little room to move, as they are forced into fixed positions by internal forces. For liquid, and gas, the density is smaller than for solid, and the particles have more room to move. We expect the RDF for gas to be completely smooth. In the next section we will discuss how we created amorphous silica, where we used the RDF to determine the state of the system.

3.8 Creating Amorphous Silica

We refer to non-crystalline systems as being amorphous. We can create amorphous silica by heating silica to the point where it melts, and then rapidly cool the system to a solid state. The procedure of melting and

[2]Molecular-builder is being developed at CCSE by Henrik Andersen-Sveinsson et al. https://github.com/henriasv/molecular-builder.

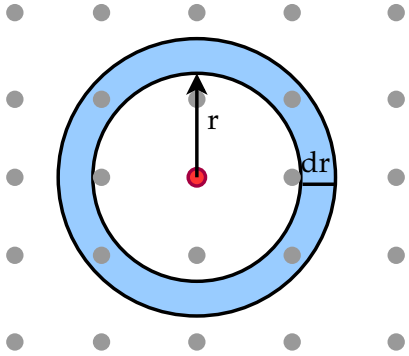


Figure 3.1: Visualization of the radial distribution function for a given radius r and shell thickness dr. The particles inside the shell, which is shaded blue, contributes to the value value of the RDF from the perspective of the red particle.

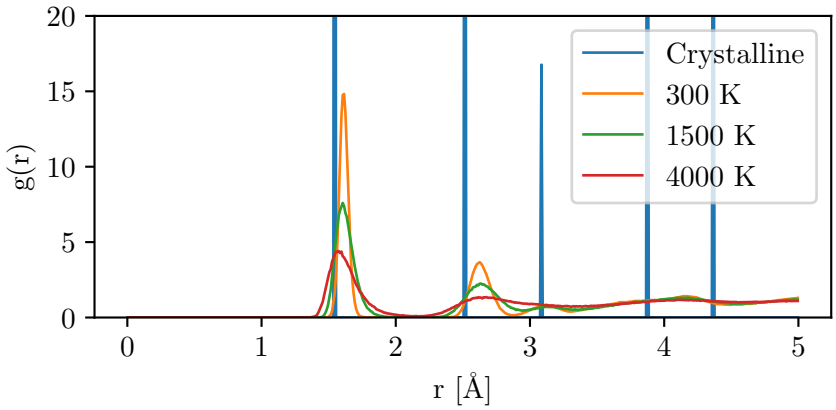


Figure 3.2: Radial distribution functions for amorphous silica.

cooling was based on the method outlined in the paper proposing the Vashishta potential by Vashishta et al. [52]. The system was initialized as a 142.4Å $\times$ 142.4Å $\times$ 71.2Å system of β-cristobalite. β-cristobalite is a crystalline silica. We heated the system from 300K to 4000K over a period of 36 ps with Nosé-Hoover thermostat by setting the temperature of the heat bath to the desired temperature. We then equilibriated the system at 4000K with Nosé-Hoover thermostat for 15 ps. Following this we cooled the system by Nosé-Hoover thermostat in steps of 500K down to 2000K. By observation using RDF we saw that the system begin solidifying between 2000K and 1500K. Further step wise cooling was done in steps of 1500K, 1000K, 600K and then 300K. The cooling was performing using a barostat to allow the volume of the system to change as the system sets. Following the cooling step we thermalized the system. The same procedure followed until the system was at 300K. In the final analysis we did not use the amorphous system. Figure 3.2 shows the RDF for this system.

3.9 Deformation

In molecular dynamics we can subject a system to a force such that the system undergoes some change to its body. The change can be reversible or non-reversible. A deformation is reversible if the body returns to the shape it had prior to a force being applied. A non-reversible deformation happens when there is some permanent change to the system, as we discussed when introducing yield stress in section 2.2.1. In Figure 3.3 we can see a comparison of initial configuration and the result of non-reversible deformation for a region for α-quartz. In the lower figure we see that there has been significant bond breakage. The material will return to how it looks in the upper figure if we remove the force causing deformation. In the following we will discuss deformation on the graphene system, followed by the SiO_2 system.

3.9.1 Graphene

Following the study by Hanakata et al. [15] we must allow for out-of-plane buckling when the sheet of graphene is deformed. To allow for out-of-plane buckling we increase the simulations box that surrounds the graphene sheet, such that it is not close to the atoms in the z-direction where the buckling is present. To subject the graphene sheet to tensile deformation we defined two groups of atoms along the perimeter along $(0, 0, 0)$ and $(L, 0, 0)$, where L is the system length in the x-direction. The groups are shown, color coded blue, in Figure 3.4. During equilibriation the forces and velocities were set to 0 for the aforementioned groups of atoms. To deform the system we initiated a linear velocity for the particles in the groups along the x-axis, such that the groups moved in opposite direction, creating tension among the remaining particles in the system. The linear velocity corresponds to the strain speed of the deformation. The force was calculated by LAMMPS' built in function for calculating the scalar value of the sum of a particle property, such as the force or velocity.

3.9.2 Alpha-quartz and Amorphous Silica

The SiO_2-system we considered deformation by shearing. The simplest, and often least complicated, procedure to perform shear deformation is to apply a deformation to the simulation box, rather than applying a force or velocity to a given group of particles. If applying a force or velocity to a group of particles we keep them steady (frozen) by either 1) not integrating the equations of motion for the particles in the group or 2) forcing the group to stay still by other commands. When a group is kept frozen the particles outside the group could have non-physical interaction with the frozen particles.

When applying a deformation with LAMMPS' built in function there are six parameters of the simulation box that are possible to change which causes a the simulations box to change. For an orthogonal simulation box we can change the parameters x, y and z corresponding to tensile

or compressive deformation by increasing or decreasing, respectively, the length of a given dimension. For a triclinic simulation box we can change the same parameters as for an orthogonal box, but also the tilt factors xy, xz and yz. The tilt factor is the displacement for a given dimension which causes an orthogonal box to transform into a parallelepiped. The vectors for a triclinic box is given by $\mathbf{a} = (xhi - xlo, 0, 0)$, $\mathbf{b} = (xy, yhi - ylo, 0)$ and $\mathbf{c} = (xz, yz, zhi - zlo)$ where the hi and lo values are the highest and lowest value for the respective dimension. The vectors **a**, **b** and **c** span the space that is the simulation box. For an orthogonal box the tilt factors $xy = xz = yz = 0$. Figure 3.5 shows an example of a triclinic box with tilt factor $xy > 0$.

When performing shear deformation we can choose to do so by engineering or true strain rate, which differ in how they change the tilt factor, i.e. how they change xy, xz and yz. Engineering strain rate changes the tilt factor by

$$T(t) = L_0 + T_0\dot{\gamma}(t - t_0)\,. \tag{3.33}$$

where L_0 is the initial length of the box perpendicular to the shearing (e.g. $L_{y,0}$ for xy deformation), T_0 is the initial tilt factor and $\dot{\gamma}$ is the shear strain rate. We observe that engineering strain rate gives a constant change of the tilt factor over time, while true strain rate is exponential, as we will see below. Since the change of the tilt factor is constant over time, the deformation, or change in some length of the system, is also constant over time. The true strain rate is defined by

$$T(t) = T_0 e^{\dot{\gamma}(t-t_0)}\,. \tag{3.34}$$

We observed that for engineering strain rate the yield stress was in many cases not as well defined as it was with true strain rate, due to how it behaves as the tilt factor grows with time. Since we needed to locate the point of yield we chose to use true strain rate for this study.

For the shear deformation we use the following command to apply a true shear strain rate on system, targeting the xy tilt factor:

```
fix fixdfrm all deform 1 xy trate ${strain_rate}
```

where the integer 1 means we apply the deformation each time step, trate stands for the true strain rate and the variable strain_rate was set to 0.004 for the amorphous system, while 0.02 for the alpha-quartz, since we forced the alpha-quartz system to be orthogonal.

3.10 Measurements in Molecular Dynamics

LAMMPS allows for outputting data from simulations in an easy manner. To retrieve data we specify what we want to measure, either per atom in a dump file, or an average over all particles for each time step. Below we will present how we measured stress, strain and using LAMMPS.

3.10.1 Collecting Data from LAMMPS

The output from LAMMPS can be chosen as per particle or averaged over all particles, both for each time step. The former is typically structured in a so-called dump file, while the latter in a log file. In both cases we can choose from a predefined set variables we wish to export, e.g. temperature, pressure, velocity or positions. We can also create groups of particles and extract an average over a given quantity, specific for the group. We also choose how often we want LAMMPS to write quantities to file, lower or increasing the resolution of our data, with the cost of size of the data file or time to read and process the data. We can also specify our own quantities to export via dump or log.

In our case we will primarily be working with log files to extract the quantities we are interested in, being the stress. Stress, as will be discussed in the next section, is found from the pressure tensor, whose components are averaged pressure for a given direction for all particles at each time step. The dump files have been used to visualize the systems in Ovito, a visualization tool for MD simulations, which support the format of LAMMPS dump files.

The log output contains columns of the data we are interested in, and also all lines of the code that has been executed in the LAMMPS script. We have created a program for extracting relevant columns of data. The program which can be found on the authors GitHub[3].

3.10.2 Stress

In LAMMPS we can extract the stress from the pressure tensor, where each element is calculated by the following equation

$$P_{ij} = \frac{1}{V}\sum_{k}^{N} m_k v_{k_i} v_{k_j} + \frac{1}{V}\sum_{k}^{N'} r_{k_i} f_{k_j}\,, \tag{3.35}$$

where $i, j = x, y, z$, N is the number of particles in the system, N' is the number of particles including ghost particles due to periodic boundary conditions, m_k is the mass of particle k, v_{k_l}, r_{k_l} and f_{k_l} are the l'th velocity, positional and force components of particle k respectively and V is the volume of the system. We find *tensile stress* and *shear stress* by choosing the correct indices in Eq. (3.35), as discussed in section 2.2.

Tensile strain

For tensile deformation the extension ΔL_i is found by $\Delta L_i = L_i - L_{i,0}$, where L_i is the new length of the system along axis i. In LAMMPS we can find the length L_i of the body by extracting the box length of the system, or measure the average distance between two groups of atoms located at either end of the system. The latter case is relevant for graphene, as we increased the box

[3] https://github.com/chdre/lammps-logfile-reader

size to allow for out-of-plane buckling. Tensile strain is then found by

$$\epsilon_{ii} = \frac{L_i - L_{i,0}}{L_i} \,. \tag{3.36}$$

Shear strain

To measure shear strain we must consider the tilt factor along an axis i. The tilt factor is the displacement of a face that changes an orthogonal box to a parallelepiped. The extension for shear deformation in ij is found by $\Delta L_i = T_{ij} - T_{ij,0}$, where T_{ij} is the tilt factor, and $T_{ij,0}$ is the initial tilt factor prior to deformation. For our case we are defoming by changing the xy tilt factor, the shear strain is then measured by

$$\gamma_{xy} = \frac{T_{xy} - T_{xy,0}}{L_y} \,. \tag{3.37}$$

For true strain rate the tilt factor, with initial value T_0, changes as

$$T(t) = T_0 \exp\left(\text{strain rate} \cdot \Delta t\right) \,. \tag{3.38}$$

3.10.3 Porosity

We can measure the porosity by counting the particles of a medium, if the density of medium is isotropic. We then find the porosity by

$$\phi = \frac{N_{void}}{N_{bulk}} \,, \tag{3.39}$$

where N_{bulk} refers to the number of particles in the bulk medium, bulk being the medium without any pores, and N_{void} is the number of particles that would have been in the void. In this study we are removing particles from bulk to produce a porous medium. We can find the number of particles in the void by either counting the number of particles that were removed, or counting the remaining particles after removal and subtracting this from total number of particles in bulk N_{bulk}. We chose the former by extracting the number of particles using atomic simulation environment[4].

3.11 Molecular Dynamics Precision

With all numerical calculations the precision is limited by machine precision and how fine we choose the resolution of the calculations to be. In our case we can decrease the time step to increase the precision at a cost of increased computational time. We aim to find the irreducible error, as was done by Hanakata et al. [15]. By running multiple series of identical simulations, which we expect to have the same yield stress and strain, we

[4]Molecular-builder relies on a Python package called Atomic Simulation Environment (ASE). ASE is compatible with LAMMPS data files, and it can be used to create atom objects. The atom objects allows for easy extraction of the number of particles, and other properties.

can calculate the root mean squared error (RMSE) of the quantities. This error corresponds to the molecular dynamics (MD) precision.

To measure the precision we chose three different systems where each had a different cut density. Cut density the density of grid cells where particles were removed, which is an estimate of the porosity. The chosen cut densities were 0.35, 0.5 and 0.65. The cuts were manually made and system of choice was α-quartz. We performed MD simulations for each system 10 times, each time with a different seed for initializing the velocities. These runs should be identical, if we relax the system for a very long time. We have chosen a period of 100 ps to relax the system, as we have done for the data we apply on the CNN. We then compare the yield stress and corresponding strain with the mean value by RMSE

$$RMSE(\sigma^y) = \sqrt{\frac{1}{N}\sum_{i=0}^{N}\left(\sigma^y - \bar{\sigma^y}\right)^2}, \tag{3.40}$$

which was found to be 0.025 GPa. The same procedure follows for strain, which was found to be 0.001.

3.12 Load Balancing

When LAMMPS distributes work to different cores the particles are, by default, distributed evenly among the cores in a non-random fashion. The particles are distributed by dividing the domain into a grid, where each grid cell contains the workload for each core. This minimizes the communication that is necessary between the cores, since we maximize the neighboring particles for each core. Since particles are permitted to move, they can also change what grid cell they are in, which means the computations are moved to a different core. Load imbalance occurs when one or more tasks are more time consuming than others, such that the cores are given an uneven amount of work. With a large load imbalance a part of the cores could be left performing no calculations while waiting for others cores to finish their tasks. The goal with load balancing is to evenly distribute the work across the cores, this is done either manually by distributing parts of the work by dividing the grid in a specific way or automatically. By performing load balancing in LAMMPS we balance the amount of particles on each core with the goal of a homogeneous distribution of particles across the cores. In LAMMPS we can do this automatically by the following command

```
fix ID group-ID balance Nfreq thresh style args
```

where we can choose the frequency (Nfreq) of when to balance, threshold (thresh, the load imbalance) for when to perform load balancing and how to decide the load balancing by style, which also is a decider for the grid we wish to load balance on. In our case we cannot change the grid of where to do load balancing over (by keyword rcb) due to our simulation box not being orthogonal. The keyword shift takes three arguments which

lets us choose which dimension(s), maximum number of iterations through the chosen dimension(s) and the stopping load balance when performing a load balance. The chosen threshold was 1.05, corresponding to a load imbalance of 5% and the frequency of checks each 1000 iterations. For the load balance we chose 10 iterations in all dimensions xyz with a stopping criterion of 1.05.

```
fix fixloadbal all balance 1000 1.05 shift xyz 10 1.05
```

The simulations are executed on Nvidia Tesla P100 GPUs, which have 3584 CUDA cores. The system consists of 108108 particles, which means there are ideally $108108/3584 \approx 30$ particles for each core to handle. With the load balancing factor of 1.05 means load balancing will occur when there are more than ≈ 31.5 particles on a core.

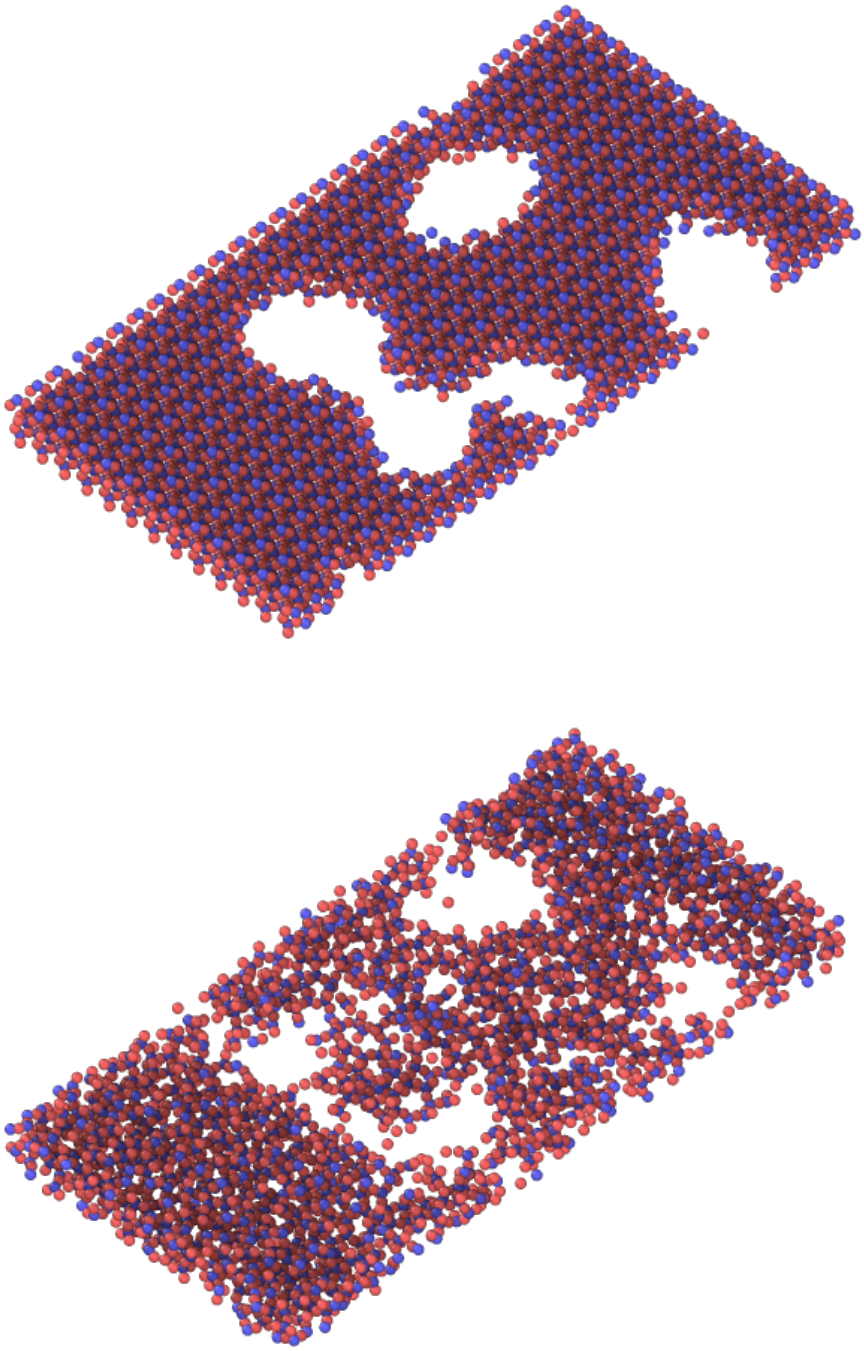


Figure 3.3: Comparison of a slab for α-quartz which has been subjected to deformation. Size of the slab is 14.07nm $\times$ 0.71nm $\times$ 7.17nm. Full system shown in Figure 3.6. The lower figure shows the result from non-reversible deformation. Figures made with Ovito [50].

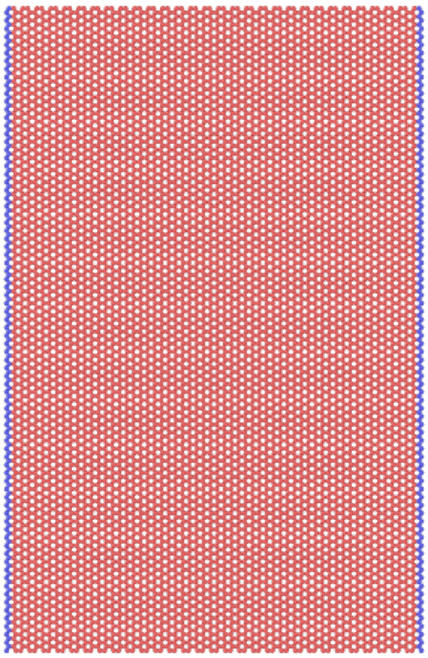


Figure 3.4: Graphene sheet with edge groups color coded blue. The edge groups were given a velocity such that the sheet was subjected to tensile deformation. Figure made with ovito [50]. System size in 12.5nm $\times$ 19.3nm.

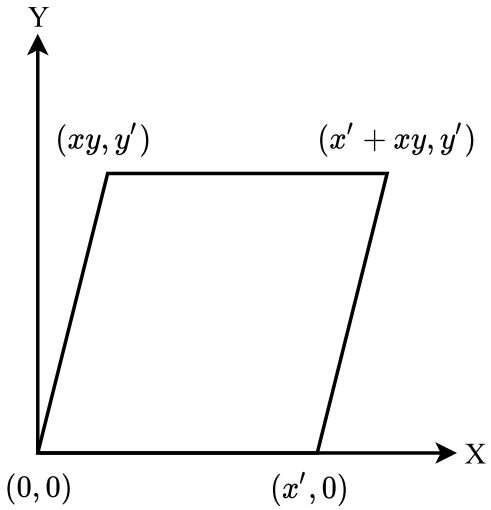


Figure 3.5: Visualization of the tilt factor xy by the coordinates of the vertices of a two dimensional box. The full length of the box are marked by a prime.

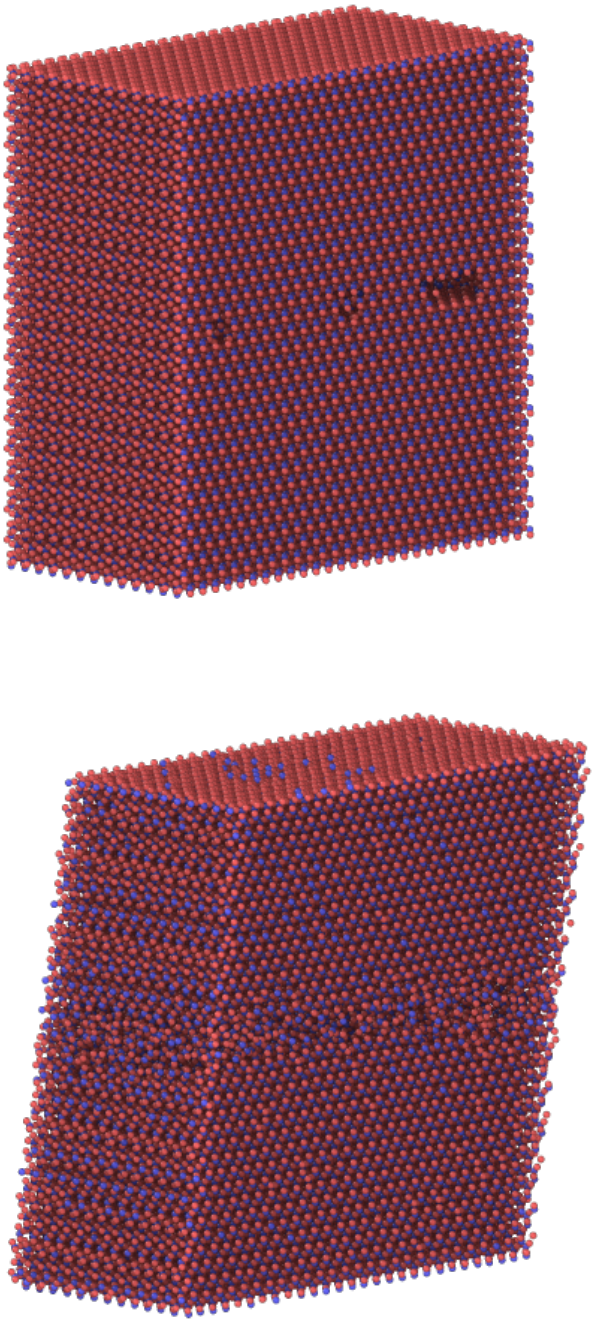


Figure 3.6: α-quartz subjected to deformation by changing the xy tilt factor. Size of the system is 14.07nm $\times$ 14.36nm $\times$ 7.17nm. The region shown in 3.3 is part of this system. We see that the lower figure shows significant failure. During simulation the top half is sliding across the bottom half. Figures made with Ovito [50].

Chapter 4

Creating Geometries

We aim to create material structures which can be represented by an two dimensional image. The image representing the material structure will serve as the regressor in our machine learning model. To create different structures we are removing particles from the system in a controlled manner. For the main part of the study we are focusing on α-quartz, and we consider removing particles from a subspace located at the bottom of the top half of the system, as shown in Figure 4.1, showing the α-quartz system. This subspace will hereby be referred to as the cut space. We can consider the system as two systems that have been fused together, the bottom half with a smooth surface and top half with a rough surface, resulting in a porous system. We will refer to the different ways we have removed particles in the cut space as geometries.

By removing regions of the system we are increasing the porosity. We are introducing local weaknesses, as we know that the strength of a system is often related to the largest flaw, due to stress intensities arising as the system is subjected to deformation.

We have divided the cut space into a 2D grid, in each grid cell particles can either be present or removed. We consider methods for choosing which grid cells we remove particles from. For the Graphene system we considered a 3×5 grid as Hanakata et al. [15]. For the SiO_2 system we chose two grids, 10×10 and 100×50. The 10×10 grid was represented by a 40×20 array.

The cut space spans the system in the x, y dimension, the height being $1/10$ of the total width of the system. The size of the cut space is $140.77\text{Å} \times 143.68\text{Å} \times 7.17\text{Å}$. The cutoff distance for Vashishta 1994 and 1997 is 5.5Å, which is smaller than the height of the cut space. Having a height that is larger than the cutoff distance prevents the voids from contracting due to intermolecular forces. The specific choice for height of the cut space was done to match the width of a grid cell.

4.1 Graphene

The graphene system is divided into a 3×5 grid, yielding 15 grid cells. For each grid cell we draw a random number which decides whether we

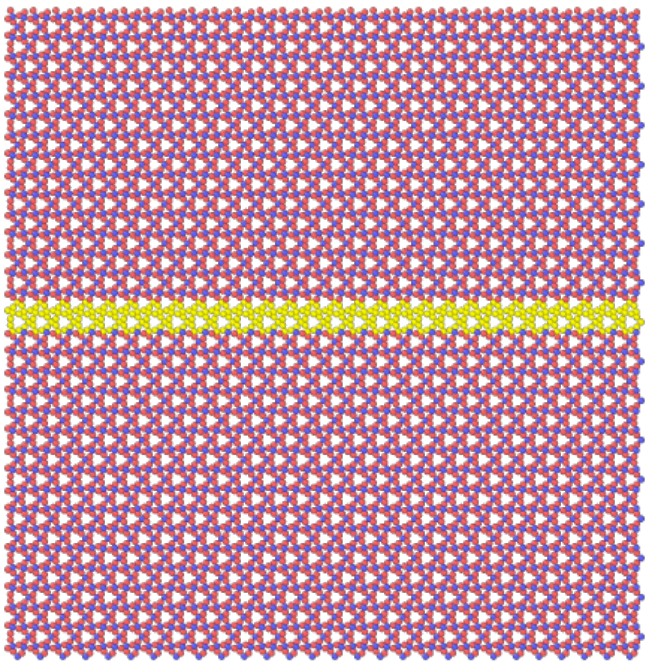


Figure 4.1: Orthogonal view of α-quartz of size 14.07nm $\times$ 14.36nm $\times$ 7.17nm. Cut space shown colored in yellow. Figure made with Ovito [50].

remove particles from that grid cell or not. In total there are 2^{15} different combinations. Due to symmetry some of these combinations are identical, so in practice we do not have to consider all the combinations. Rather than removing particles from the whole grid cell, which would result in many configurations where one part is detached from the rest, we remove a region. In Figure 4.2 we can see an example of a system where particles have been removed in five grid cells, we have also added an overlay which illustrates the 15 grid cells. The data set we created to train a machine learning model to predict strong geometries consisted of 99 randomized geometries. As this was only an introductory task before applying similar methods to SiO_2 we did not investigate a more thorough sampling method.

4.2 SiO_2

In the silica systems we split the cut space into 10×10 and 100×50 grids. The size of the grid cells are 14.07Å $\times$ 7.17Å $\times$ 7.17Å and 1.40Å $\times$ 7.17Å $\times$ 1.43Å, respectively. We consider removing particles by sampling the grid space by random choice, creating geometries manually and by applying an algorithm which creates pseudo-random noise.

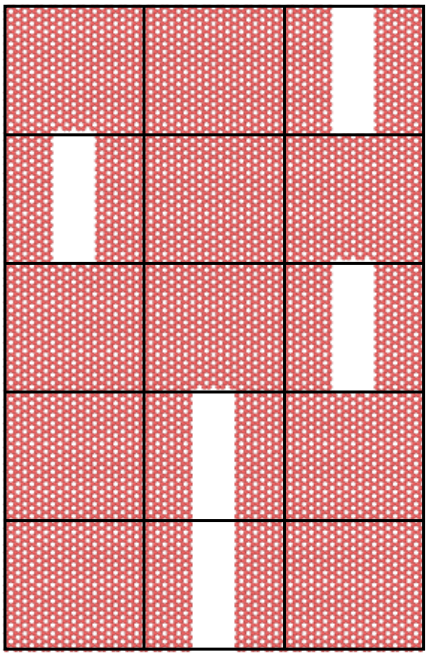


Figure 4.2: Example which shows particles removed from 5 grid cells, for the graphene system, with an overlay illustrating the grid cells. The long void shows two connected voids. Figure made with Ovito [50].

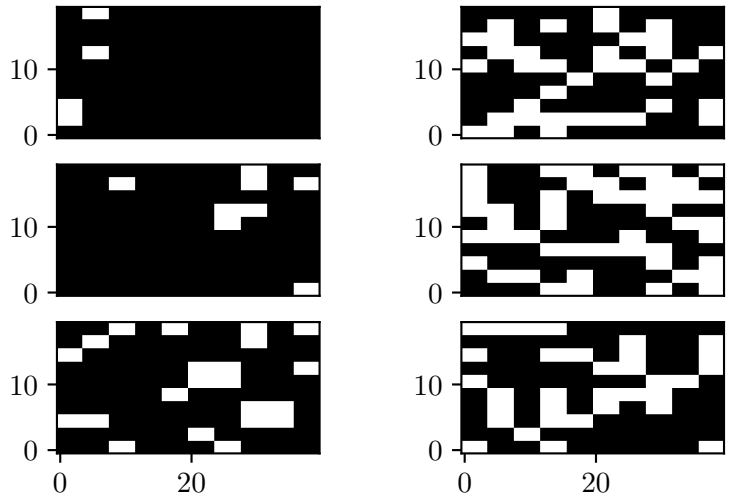


Figure 4.3: Random geometries for the 10×10 grid. Corresponding MD cut space is shown in Figure 4.4.

4.2.1 Random selection

For the 10×10 grid we considered a random choice for which grid cells we removed particles from. We first sampled a probability p_r for particles to be removed, which followed a uniform distribution. For each grid cell we then did a binary random choice with probability p_r, which was done by Numpys random choice:

```
numpy.random.choice(a=[0, 1],
                    size=100,
                    p=[p_r, 1 - p_r])
```

which creates an array with 100 elements, where each element is a random choice of either 0 or 1, with corresponding probabilities p_r and $1 - p_r$. Each entry in a has the probabilities stated in p, respectively. Figure 4.3 shows some examples of randomized geometries, the voids are shown in white, as they will be throughout this study. By using random choice with defined probabilities of the entries we obtain a non-uniform sampling of the porosity. Had we instead sampled a uniform distribution we would statistically have sampled mostly systems where the porosity of the cut space had been 0.5.

With our model made with higher resolution grid (100×50) we wanted to test randomized geometries on the trained model, to see how it would perform on geometries made with different methods. By sampling as discussed above on a 100×50 grid, we will in most cases create a granular surface with very small voids. By instead making the grid smaller we can scale the geometry up to 100×50, which leaves a geometry with larger voids. We initiate randomized boolean arrays of size 10×5, 20×10 and 50×25 which are all a multiple of 100×50, which is important for the

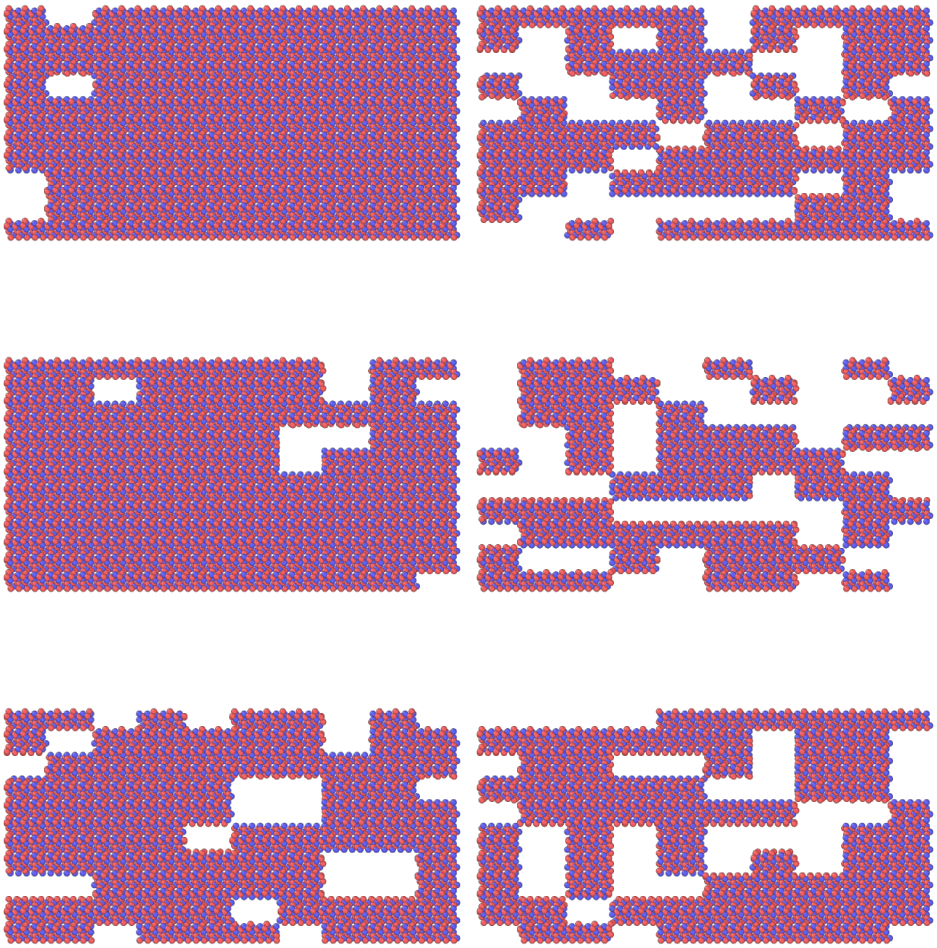


Figure 4.4: Cut space of the α-quartz system showing random geometries for the 10×10 grid. Size of the cut space is 14.07nm $\times$ 0.71nm $\times$ 7.17nm. MD implementation of arrays shown in Figure 4.3. Figures made with Ovito [50].

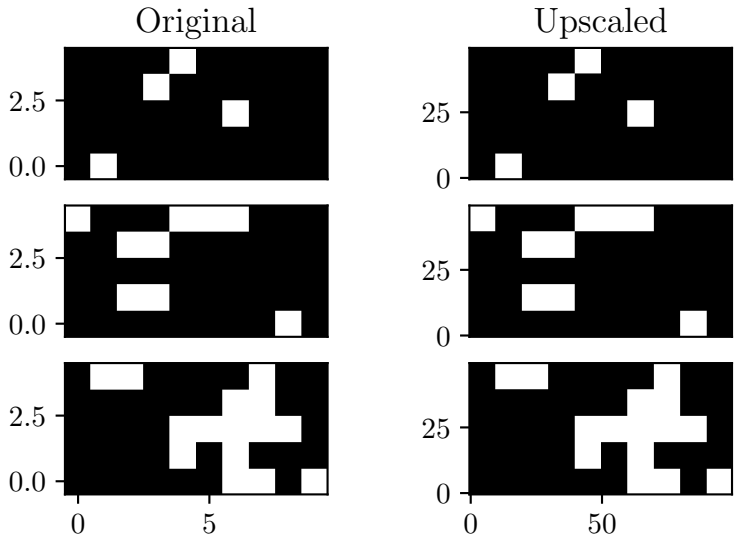


Figure 4.5: Example of three 10×5 randomized noise which are scaled up to 100×50 using Kronecker product.

method we used to scale. To scale up the initial array we use the *Kronecker product*. Consider two matrices $\mathbf{A}$ and $\mathbf{B}$ with dimension $i \times j$ and $k \times l$ respectively. The Kronecker product multiplies each element a_{ij} in $\mathbf{A}$ with the matrix $\mathbf{B}$, resulting in a matrix of size $i \cdot k \times j \cdot l$. Let $\mathbf{A}$ to be a 10×5 initial randomized array as discussed above, we can then let $\mathbf{B}$ be an array of ones with a size that scales the array to a given dimension. We want the output of Kronecker product to be of size 100×50, and by letting the size of $\mathbf{B}$ be a multiple of the size of $\mathbf{A}$ we achieve a scaled up version of $\mathbf{A}$. E.g. for the case of $\mathbf{A}$ being 10×5, the size of $\mathbf{B}$ must be 10×10 if we want the resulting matrix to be 100×50. An example of upscaling is shown in Figure 4.5.

We also considered passing circular voids through the trained network. We decided on a range of radii from $4 \rightarrow 25$. We wanted a maximum radius that was at most half the system size, and a minimum radius which was large enough to where we could have the circle not represented as a box, due to low resolution. The lowest radius was found by trial and error. For a given radius we create multiple geometries with different number of disks, ranging from 1 disk to at max 55 disks. We did not allow for the disks to overlap, so if there were no possible ways to add a new disk to the existing structure, given some finite number of attempts, we terminated the sequence. This allowed us to generate multiple geometries for a given radius, which was important, as there is a finite amount of radii we can use to represent different disks. If two radii are close to equal, we will in practice get identical disks, due to the resolution of our grid. In Figure 4.6 we can see some samples of geometries with circular voids.

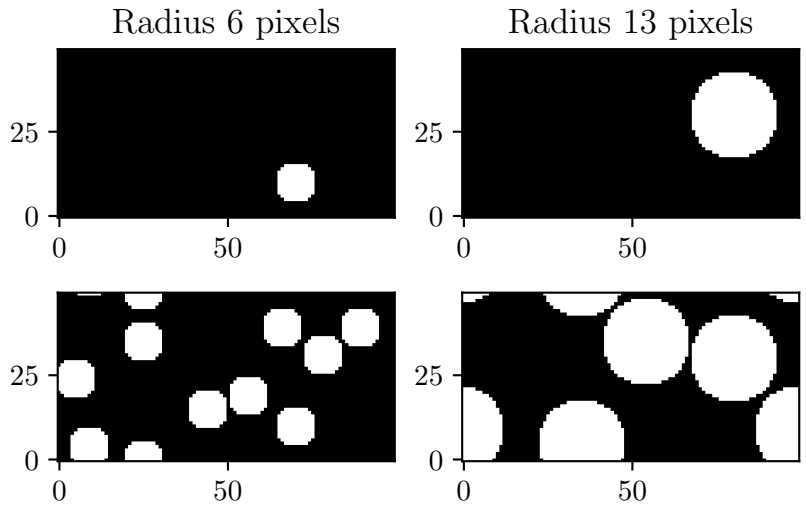


Figure 4.6: Example of two circular noise geometries. Two initial circles with radius 6 and 13 pixels. Below we the respective radii we can see an example of 10 and 4 repeated circles for 6 and 13 radius, respectively.

4.2.2 Manual selection

With a 10×10 grid we have 2^{100} possibilities of sampling the grid, i.e. 2^{100} possible surfaces. When sampling such a large space as discussed in 4.2.1, the grid cells indicating removal of particles will in most cases be evenly spread out, and we are creating white noise, yielding granular surfaces. When we are increasing the sample size we expect the system to behave similarly for all the random cut configurations, since they statistically have the same properties for a given porosity. The probability of sampling a cut that consists of four grid cells next to one another, forming e.g. a 2×2 cut is very low. To see how the system behaves with configurations that seemingly have a low probability of appearing when sampling randomly, we decided to manually create cut configurations. The procedure was to decide on a type of structure (e.g. square or a line with a specific thickness) which we placed such that we could periodically move it across the cut space to get multiple samples from a single structure. In Figure 4.7 we can see an example of such a structure.

4.2.3 Simplex Noise

To further create cut configurations we chose to look at methods for creating noise. Simplex noise is a pseudo-random noise function that is often used in game design to create surfaces in 2D and 3D. Looking at noise generated by the Simplex algorithm the naked eye can easily see that it can, given the correct inputs, resemble a landscape. See Figure 4.8 for an example of Simplex surface. Since we can create real world-like surfaces

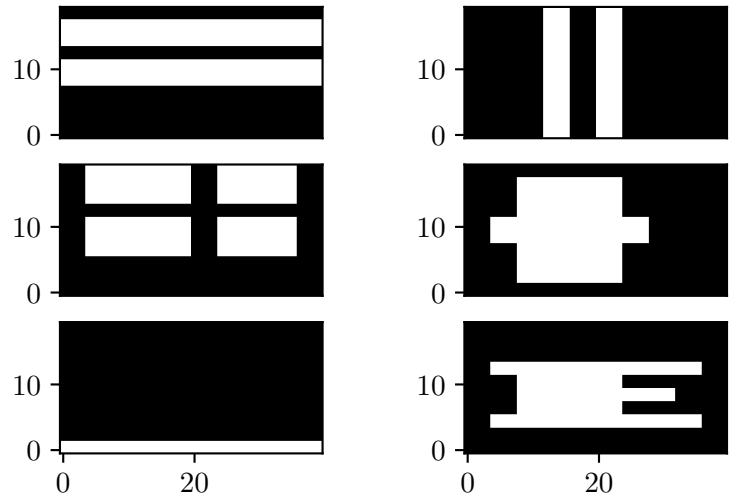


Figure 4.7: Manual structure for the 10×10 grid.

with Simplex noise we expect this to be a good method to create new cut configurations.

We decided to increase the resolution of the cut configuration when using Simplex noise to better represent the geometries created by the algorithm. Ideally we would have 1 pixel for each particle (given a 2D cut space), but since we are to pass images as inputs to a machine learning algorithm we must consider the computational cost. We chose a 100×50 grid, which leaves us with 5000 nodes for the first convolutional layer. We are again operating with a boolean array and there are 2^{5000} different possible cut configurations.

We still consider the cut space to be 2D, therefore we use a 2D noise function when creating the cut configurations. The Simplex algorithm returns values $\in [-1, 1]$ which we map onto $[0, 1]$, for practical reasons discussed below.

Parameters

For Simplex noise we have parameters scale, octaves, base and threshold. The first two parameters mainly decide the shape of the noise, while base applies a displacement to the noise coordinates and threshold sets lower bound for the noise values. Scale is a parameter that decides the size of the structure of the Simplex noise, it scales the noise. By varying the scale we can control the size of the cuts, where a large scales yields larger voids and small scales yields to smaller voids. Octaves is an integer parameter that decides how many passes of noise we perform. E.g. 2 octaves means we perform two passes of noise, which could create more complex structures. We observed little to no difference above 3 octaves. Threshold is a parameter which decides the lowest noise value we accept

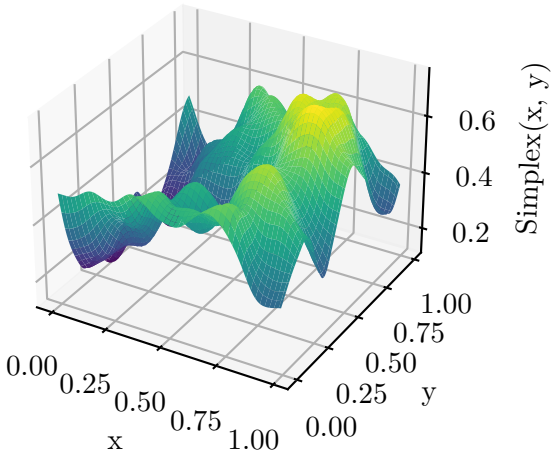


Figure 4.8: Example of a simplex surface.

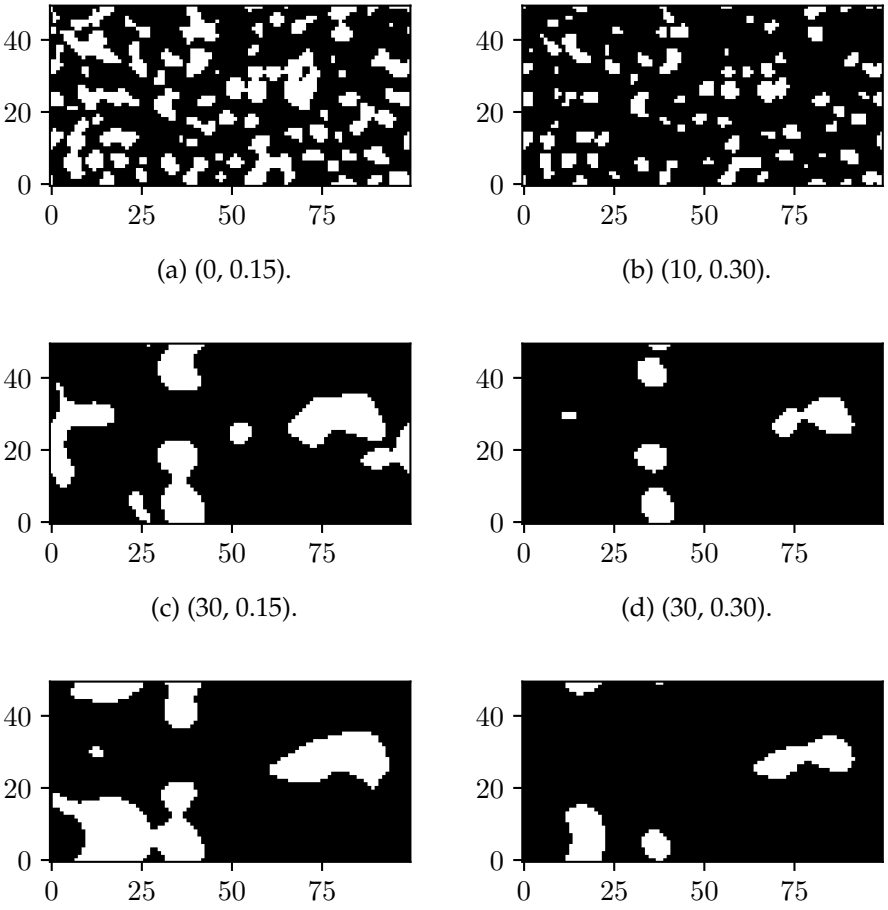
when building geometries. Varying the threshold will allow us to sample different porosities with the same scale. We can see the effects of scale and threshold in Figure 4.9. We define a threshold $\in [0, 1]$ for the noise values, i.e. if the Simplex noise value $<$ threshold we set the value to 0, and if it is $\geq$ we set the value to 1, i.e.:

$$\text{noise value} = \begin{cases} 0 & \text{Simplex noise value < threshold} \\ 1 & \text{else} \end{cases} \tag{4.1}$$

To make the Simplex noise periodic we use the system size as input for creating tileable noise. It is important that we make the noise periodic, or we could get strange boundary effects from voids which are split across the boundary. If we consider a circular shaped void which is split in half across a boundary, the MD simulation will interpret this as a void shaped as a D. From this sample case we might accept the void as split, but we have no control over how the voids look at the boundary, and it is safer to choose periodic noise to avoid unwanted boundary effects. We can change the initial position of the noise coordinates by a factor, which will be called base. Shifting the noise coordinates is useful when creating tileable noise, as you avoid making similar noise for different seeds.

4.3 Penny Shaped Cracks for Griffiths Theory

When studying Griffiths theory of brittle fracture on α-quartz we chose cylinders as the initial crack. The cylinders were chosen to have the a depth corresponding to the thickness of the cut space, $L_z/10 \approx 7.15$Å along the z-axis. The radius of the cylinder, being the half crack length, is kept well below the system sizes in the xz-plane, i.e. $a \ll L_x$, since L_x is the smallest



(a) (0, 0.15).

(b) (10, 0.30).

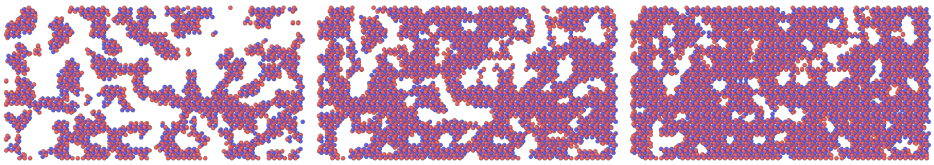
(c) (30, 0.15).

(d) (30, 0.30).

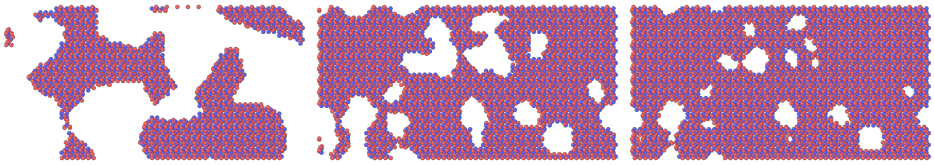
(e) (40, 0.15).

(f) (30, 0.30).

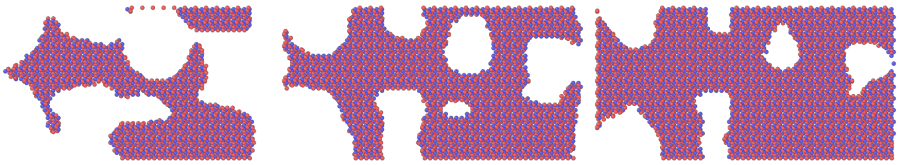
Figure 4.9: Examples of Simplex noise with varying scale and threshold, noted for each subfigure as (scale, threshold). For all samples the octaves are set to 1 and base= 1. We see that increasing the scale makes the voids smaller, as we accept fewer noise values.



(a) (40, 0). (b) (10, 0.15). (c) (10, 0.30).



(d) (30, 0) (e) (30, 0.15). (f) (30, 0.30).



(g) (40, 0). (h) (40, 0.15). (i) (30, 0.30).

Figure 4.10: Examples of Simplex noise with varying scale and threshold, noted for each subfigure as (scale, threshold). For all samples the octaves are set to 1 and base 1. We see that increasing the scale makes the voids smaller, as we accept fewer noise values. The system displayed is the α-quartz cut space. Size of samples is 14.07nm $\times$ 0.71nm $\times$ 7.17nm.

length in the xy-plane. The shearing was done in the xz-plane. We chose the xz-plane due to $L_x \sim L_z$. The radii were chosen as 16 evenly distributed points $\in [L_x/10, L_y/5]$, where $L_x < L_y$ are the system lengths of the cut xy-plane.

4.4 Methods for Removing Particles

In the above sections we have gone through the methods for choosing the grids which represents different geometries in the cut space. To remove the particles we have used a modified version of the Python package molecular-builders method for creating surfaces based on Simplex noise, and an extension which simply maps values from an input array to the particles. Molecular-builder maps values to each particle, while we are working with grids which are not one-to-one with the particles. Instead each grid cell represents a subset of particles, and we therefore have to map the particles to each grid cell. This is done by evaluating the distance from the particle positions to the elements on the grid. More specifically we extract the x and z coordinates of the particles and compare these to a low resolution representation of the system. The correct index for each particle is mapped by

$$x_{index} = \min|x_{\text{particle}} - x_{\text{grid}}|\,, \tag{4.2}$$

where x_{particle} contains the x coordinates for all the particles, and x_{grid} contains the low resolution x coordinates. In other words, we are mapping the particles to the grid by finding which grid cell they are closest to. x_{index} then contains the indices corresponding to the grid for each particle. Same procedure follows to map the z coordinates.

For the case of Simplex noise we use a slightly modified version of molecular-builders geometry ProceduralSurfaceGeometry, that only removes particles inside the cut space, rather than all the particles above a given surface. Another option could be to add the particles above the cut space back (e.g. by ASE) after removing particles from the cut space.

Chapter 5

Machine Learning

In recent years machine learning has gained in popularity due to the availability of data and increasing computational power. This has spiked an interest exploring applications for regression analysis, classification and clustering, and even reducing the dimensionality of data. For image classification machine learning has even surpassed human-level performance, as shown in a study by He et al. [18].

Machine learning is a method for building models based on a set of data. The method learns by automation under a set of rules decided by the user. Machine learning methods can learn complex relationships in the data, more so than traditional predictive models, such as polynomial regression. In natural sciences we are often working with data that has some underlying hidden trends, which can vary in complexity. In some cases we might be able to explain an outcome through a simple predictive model, in other cases

The goal of machine learning is to create a model that optimizes a set of parameters iteratively under the constraint of minimizing the error of the prediction made by the model. By learning, in the context of regression analysis or classification, we mean an iterative method of self optimizations following a set of instructions which are not explicit. We distinguish between unsupervised and supervised learning. In supervised learning we have some target value for each input that the machine learning algorithm learns to optimize towards, much like regression you have a set of data points you fit a model to. In unsupervised learning we allow the algorithm to self-optimize without a target value. The name unsupervised comes from the fact that do not explicitly tell the algorithm what to optimize towards, simply optimize under a given constraint. Autoencoder (AE) is an example of unsupervised learning. The goal of AEs is to find the optimal way to encode and decode an input, under the constraint of minimizing reproduction error. AEs and reproduction error will be discussed below. In this chapter we will present relevant machine learning methods used in this study, and a high level discussion of neural networks and how they operate.

The following books were used to gather information about the relevant methods. The parts regarding component analysis was gathered from

Hastie [17], while for neural networks we used Aggarwal [2], Zhang et al. [58] and Géron [14].

5.1 Neural Network

Neural networks (NN) are are a branch of machine learning which is loosely inspired by the human brain in how it learns to recognizes patterns or subsets of data. A NN contains of a set of interconnected nodes with parameters that change as the model learns patterns. NNs that are made for object recognition serves as a good example in how a network learns in a similar way as the brain does. If we expose a NN to a set of images containing an object, the network will search for patterns in the images, ultimately starting to recognize patterns in the object, which leads to it recognizing the object. One can say that nodes in the network are activated upon recognizing subsets of the input, much like neurons in the brain.

A NN consists of an input layer, one or more hidden layers and an output layer. The number of hidden layers are determined by the user, and increasing the number of hidden layers increases the complexity of the model. The hidden layers consists of so-called nodes, with corresponding weights and biases. In the literature nodes are also often referred to as neurons or units. The output from a hidden layer (also called an activation when considering a specific node) is found by

$$\mathbf{a}^l = f((\mathbf{W}^\mathbf{l})^T\mathbf{a}^{l-1} + \mathbf{b}^l) = f(\mathbf{z}^l), \quad l = 1, 2, 3, ... \tag{5.1}$$

where l denotes the layer, $(\mathbf{W}^l)^T$ is the transposed of the matrix containing the weights, $\mathbf{b}^l$ is a vector containing the biases and $f(\cdot)$ is some activation function, which will be discussed in section 5.1.1. Each value in $\mathbf{a}^l$ corresponds to the output from a given node in a hidden layer. For the first hidden layer $\mathbf{a}^{l-1}$ is just the input $\mathbf{x}$ to the NN. We note that $\mathbf{a}^0 := \mathbf{x}$. A common option is to initialize the weights to be small random numbers uniformly distributed about zero, a widely used method for initializing the weights was presented by He et al. [18], known as Xavier-He initialization. Xavier weight initialization, proposed by Glorot and Bengio [12], initializes the weights as a uniform distribution with standard deviation $1/\sqrt{n}$, where n is the number of nodes in the previous layer, starting at the first hidden layer, the previous layer then being the input. He et al. [18] proposed that for ReLU type activation functions a weight initialization of $\sqrt{2/n}$ performed better. The initialization of bias is a subject of discussion. He et al. [18] suggests initializing the bias to zero, allowing the distribution of the weights to ensure that $\mathbf{z}_l$ has zero mean and is symmetrically distributed about zero. In our case the input can potentially consist of mostly zeros, which would result in most nodes being dead (activations equal to zero), as we can see from Eq. (5.1). Non-zero initial bias makes sure we avoid the issue of dead nodes due to the input being zero. Using Xavier-He initialization for bias is common in machine learning. A study performed by Li and Nia [31] on the effects of randomized initialization of bias on binary input suggests that it performs well.

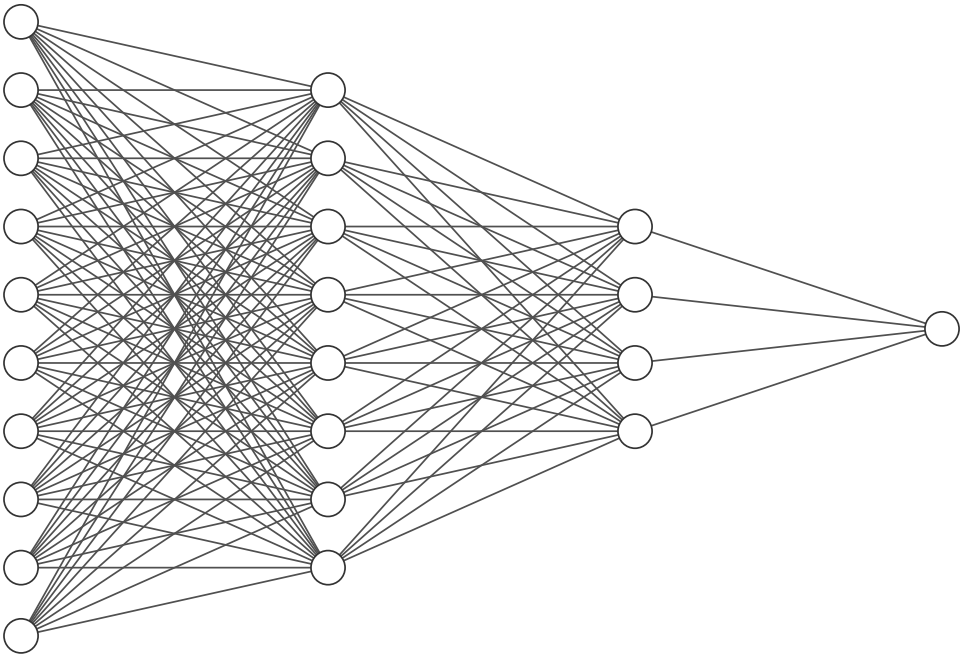


Figure 5.1: Visualization a neural network with $10 \to 8 \to 4 \to 1$ nodes, from the input layer to the output layer. We can see that there is a connection between each node. Figure made using software by LeNail [30].

In figure 5.1 we can see a visualization of a fully connected neural network. Fully connected neural networks are called dense neural networks (DNN). We can see that all the nodes are connected, hence the term fully connected. Each node in layer l has contributions from all nodes in layer $l-1$. For a given layer a specific node has a set of corresponding weights for each node in the next layer, and a single value for the bias. E.g. the weight matrix connecting the input layer to the first hidden layer has the shape $n_{input} \times n_{hidden}$. As an example, we can find the activation for node j in layer l by

$$a_j^l = f\left(\sum_i^{n^{l-1}} W_{ij}^l a_i^{l-1} + b_j^l\right), \tag{5.2}$$

where n^{l-1} is the number of nodes in layer $l-1$. As we can see in the equation above the sum runs over all nodes in the previous layer.

5.1.1 Activation Functions

To decide what information passes to next layer, or the strength of the signal, we use an activation function. Activation functions are important because they add non-linearity to the neural network. If we were to use the identity function as our activation function, we see from (5.2) that we would pass a linear combination of weights and biases to the next layer. Using the identity function we would reduce a multi-layer neural network to a single-layer neural network perming linear regression [2, p.

30]. By wisely choosing an activation function we introduce non-linearity to our neural network, which is important if we want the model to learn complex non-linear properties of the input. Some activation functions force values between $(0, 1)$ or $(-1, 1)$, while others force the values to be ≥ 0, or penalize values that are ≤ 0 for practical reasons that will be mentioned below. In early machine learning Sigmoid functions was widely used, but in time other alternatives surfaced, such as rectified linear unit (ReLU) and tanh which showed promising results, Sigmoid still has some use today, notably in classification and autoencoders. The most widely used activation function today is ReLU.

Sigmoid

Sigmoid returns values that are between 0 and 1. We can interpret the output from a Sigmoid as a probability, which makes it good for certain classification tasks, as the output can be interpreted as the probability for a given classification. Sigmoid is defined as

$$\sigma(x) = \frac{1}{1 + \exp(-x)}\,. \tag{5.3}$$

From a standpoint of computational cost Sigmoid is a good choice due it having a simple derivative

$$\frac{d\sigma}{dx} = \sigma(1 - \sigma)\,. \tag{5.4}$$

Rectified Linear Unit

There exists multiple variants of rectified linear unit (ReLU) that are aimed at fixing issues related to vanishing gradient by adding a term to ReLU output if the input is below 0. Other methods tries to set a threshold for the maximum value to solve the problem with exploding gradient. ReLU is the most used activation function in recent years. ReLU is defined as

$$ReLU(x) = \begin{cases} 0 & x < 0 \\ x & x \geq 0 \end{cases}. \tag{5.5}$$

The derivative of ReLU is simply 0 if $x < 0$ and 1 if $x > 0$. For $x = 0$ the derivative of ReLU is not defined mathematically, but the convention is to set the value of the derivative to 0 for $x = 0$ as well.

A study by Lu et al. [33] notes that initialization of bias can be critical to avoid dying gradients when using ReLU, but that it is mostly a concern for very deep DNNs.

Scaled Exponential Linear Unit

Scaled exponential linear unit (SELU) was proposed by Klambauer et al. [28] as a part of their paper on self-normalizing neural networks. SELU is

defined as

$$SELU(x) = \lambda \begin{cases} x & x > 0 \\ \alpha(e^x - 1) & x \leq 0 \end{cases}, \tag{5.6}$$

where α and λ are constants which both are larger than 1. SELU is based on ELU, which takes the same form as SELU when $\lambda = 1$. This activation functions was a part of exploring self-normalizing convolutional neural network, which was not used in this study.

5.1.2 Backpropagation

As stated in the previously NNs are essentially tuning parameters in such a way that the loss is minimized. By looking at the gradient of the loss function we can determine the direction which decreases the loss, with respect to some parameter. This method is called *gradient descent* (GD), and it is an important tool in optimization of NNs. We update the weights and biases by subtracting the gradient of the loss function with respect to the weight or bias, i.e. we look at a parameter that perturbs the output in some way, and we update it such that it minimizes the loss. The updates of the weight and bias follows

$$w_{ij}^l = w_{ij}^l - \lambda \frac{\partial L}{\partial w_{ij}^l}, \tag{5.7}$$

$$b_j^l = b_j^l - \lambda \frac{\partial L}{\partial b_j^l}, \tag{5.8}$$

where λ is some is some step length, and L is the loss function evaluated for all samples. The gradient with respect to the weights and biases are

$$\frac{\partial L}{\partial w_{ij}^l} = \sum_k \frac{\partial L_k}{\partial w_{ij}^l}, \tag{5.9}$$

$$\frac{\partial L}{\partial b_j^l} = \sum_k \frac{\partial L_k}{\partial b_j^l}. \tag{5.10}$$

The step length defines the length of the step in the direction that minimizes the loss. In machine learning the step length is typically called the learning rate, as it is related to the learning of the model. A large step length has faster convergence, but might not converge as well as a small step length. A typical worst case is when the step length is so large that you cannot converge to an actual minima, and you might end up oscillating about the minima. On the other hand a step length that is too small can lead to slow convergence, therefore a very long learning process. Learning rates will be further discussed in section 5.1.4.

The name backpropagation comes from backward propagation of errors. During training the network makes a prediction based on the input, the prediction is then compared with the target value using the loss function. The backpropagation algorithm iterates backwards through the layers, for each node in layer l it calculates the error contribution from

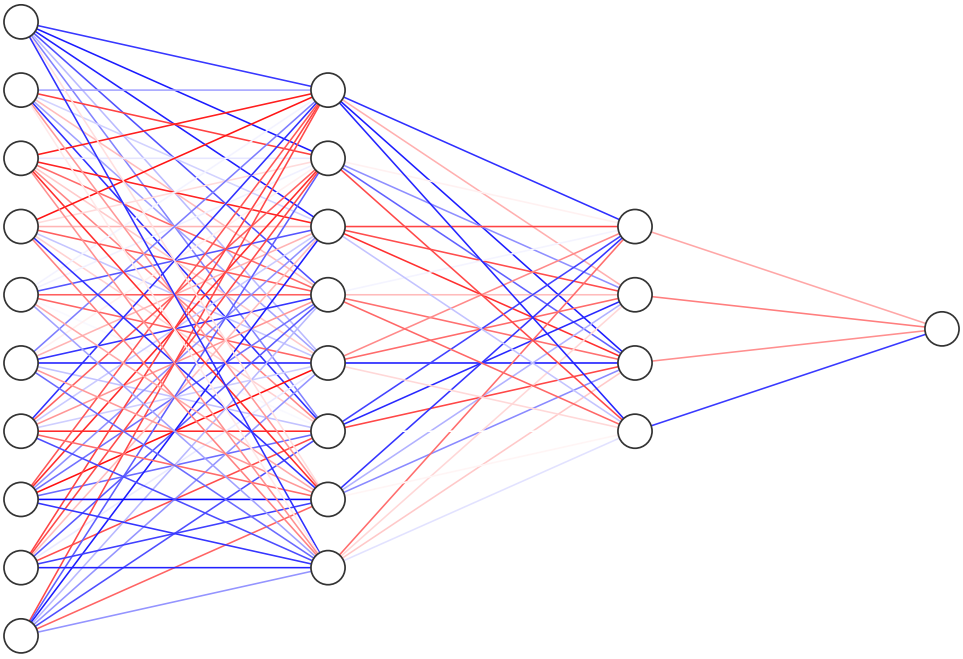


Figure 5.2: Illustration of DNN with updated weights. Red correspond to negative values, while blue corresponds to positive values. The intensity of the color correspond to the magnitude of the weight. The strength of the connection indicates the importance of the feature. Figure made using software by LeNail [30].

connecting nodes in layer $l-1$. As previously discussed each node in layer l has connections to all nodes in layer $l-1$. The weights and biases are then updated for each node in each layer, according to Eqs. (5.7) and (5.8). We can see an illustration of the nodes with updates weights in Figure 5.2.

5.1.3 Loss Function

During learning the network is self optimizing under the criterion of minimizing a loss function, i.e. tune the parameters in such a way that the loss i minimized. A common choice of loss function for NNs for regressional purposes is the mean squared error (MSE) of the target and the prediction. The MSE is defined as

$$MSE = \frac{1}{n}\sum_{i=0}^{n}(y_i - \tilde{y}_i)^2 \,, \tag{5.11}$$

where y_i is the target, $\tilde{y}_i$ is the prediction and the sum runs over all samples.

5.1.4 Optimizers

In sec 5.1.2 we discussed updating the weights and biases to minimize loss, this is known as optimizing. We briefly discussed gradient descent

as a method for finding minima of the loss function with respect to some parameter we wish to optimize. We also discussed how the step length, hereby referred to as the learning rate, is important for the convergence of gradient descent. A bigger problem with GD is that it might not converge to a global minima, it might get stuck at a local minima. GD is the foundation of many of the optimization algorithms used in NN, all aimed at improving convergence.

Adaptive learning rate is a method for adjusting the learning rate as the during training. It is a common method for improving both convergence and computational cost. The idea behind adaptive learning rate is to only use small learning rates where the changes in the loss function are greatest. In Figure 5.3a we can see an example of convergence to a minima for a large and small learning rate, if we return to the term step length, we can clearly see that performing small steps is beneficial for accuracy, but it takes many more computations. The large learning rate completely oversteps the minima, and we would expect it to oscillate above the minima until training is complete. In Figure 5.3b we see an example of adaptive learning rate. In the interval where there is little change to the loss (interval shaded red), we perform less calculations. In the non-shaded area there are more changes to the loss and we reduce the learning rate to increase the resolution of our calculations.

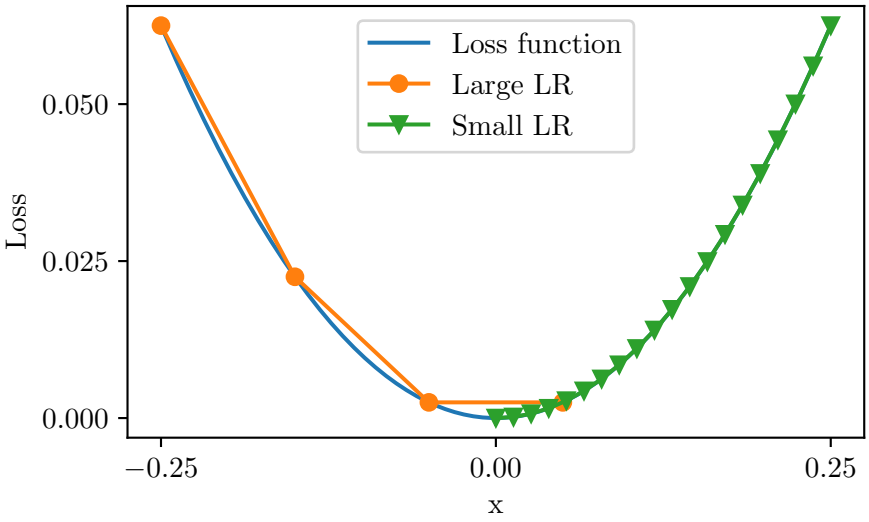
Momentum-based techniques aims at accelerating the loss function in a direction by considering the gradient at more than one step. Instead of updating the parameter, e.g. weights, by subtracting the gradient multiplied by the learning rate, we also subtract the gradient multiplied by some constant. A simple example to illustrate momentum-based learning is momentum-based GD. Consider the update of weights in Eq. (5.7), as stated above we subtract the gradient multiplied by some constant:

$$w_{ij}^{l} = w_{ij}^{l} - \beta \left(\gamma \frac{\partial L}{\partial w_{ij}^{l}} \right) - \gamma \frac{\partial L}{\partial w_{ij}^{l}} \,. \tag{5.12}$$

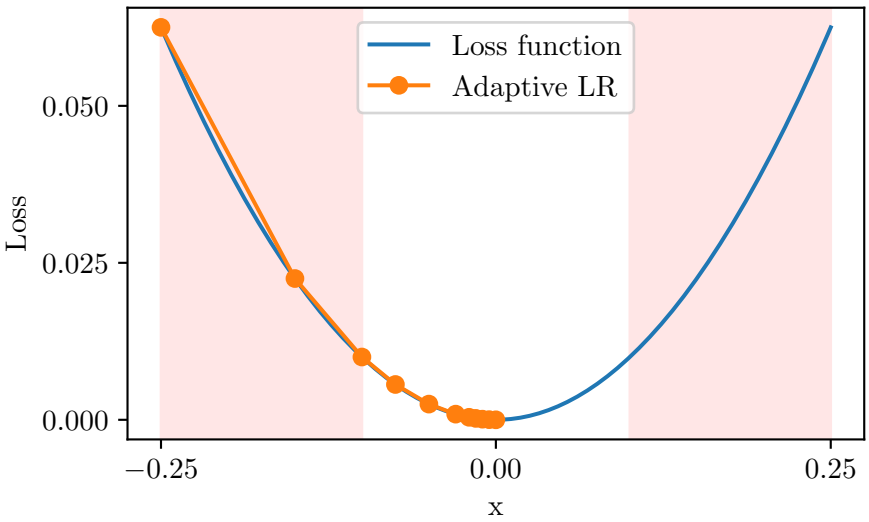
$\beta \in [0, 1]$ is a parameter that decides the momentum. If $\beta = 0$ we get standard GD, if $\beta = 1$ it corresponds to doubling the learning rate, resulting in large steps towards the minima. For momentum-based GD it is easy to see that β controls how quickly the method converges. We can consider adding the momentum-term as adding accelerating the search of a minima of the loss function.

Stochastic Gradient Descent

When it comes to converging to the global minima, standard gradient descent (GD) suffers from two issues, the first being computational cost for large data sets, the second is that it runs the risk of getting stuck in a local minima, which is different from the global minima. GD calculates the gradient based on every sample, it is obvious that this can take a lot of time for large data sets. When evaluating the whole data set we might find a path to a minima that minimizes the loss, it could be that the most obvious



(a) A comparison of convergence to a minima for large and small learning rates (LR).



(b) Example of adaptive LR. The areas colored red shows an interval of the loss function where there is little change.

Figure 5.3: x^2 used as an example of a loss function. The point $(0, 0)$ illustrates a minima of the loss function. The dots and triangles illustrate points where we evaluate the gradient of the loss function, which we wish to minimize.

descending path leads to a local minima, with the global minima located at some other point. *Stochastic gradient descent* (SGD) solves the computational cost by evaluating the gradient for a single sample, rather than the whole data set. SGD can be very unstable to at the begging of training, when fitting the data, one sample may have a significantly different gradient than another (e.g. we evaluate at an outlying sample). The added benefit of evaluating one sample at a time is that we are searching through a broader set of paths to a minima, increasing the probability of stumbling upon a path to the global minima.

Instead of a single sample, we could also implement so-called *mini-batches*, which are subsets of the data set. Evaluating the gradient of the loss for a subset, rather than a single sample will in most cases lead to model that better fits the data, as a subset is likely to represent the full data set better than a given sample. The overall equation takes on a similar form as the updates to the weights and biases in Eqs. (5.8) and (5.7), but the loss (in our case the MSE) is only calculated for one sample. The aforementioned equation then becomes

$$w_{ij}^l = w_{ij}^l - \lambda \frac{\partial L_k}{\partial w_{ij}^l} \tag{5.13}$$

$$b_j^l = b_j^l - \lambda \frac{\partial L_k}{\partial b_j^l}\,, \tag{5.14}$$

for some sample k. If we use mini-batches, the loss is summed over all samples k in the mini-batch M:

$$\frac{\partial L}{\partial b_j^l} = \sum_{k \in M} \frac{\partial L_k}{\partial b_j^l}\,. \tag{5.15}$$

The same follows for the weights.

Adam

The most widely used optimizer today is Adam, short for Adaptive Moment Estimation, a moment based gradient descent algorithm. Adam was proposed by Kingma and Ba [27]. Adam updates exponentially weighted averages of the first and second moment of the gradient, weighted with a bias correction. At iteration i Adam is updated as

$$\mathbf{g}_i = \nabla_\theta L \tag{5.16}$$

$$\mathbf{m}_i = \beta_1 \mathbf{m}_{i-1} + (1-\beta_1)\mathbf{g}_i \tag{5.17}$$

$$\mathbf{v}_i = \beta_2 \mathbf{v}_{i-1} + (1-\beta_2)\mathbf{g}_i^2 \tag{5.18}$$

$$\hat{\mathbf{m}}_i = \frac{\mathbf{m}_i}{1-\beta_1^i} \tag{5.19}$$

$$\hat{\mathbf{v}}_i = \frac{\mathbf{v}_i}{1-\beta_2^i} \tag{5.20}$$

$$\boldsymbol{\theta}_{i+1} = \boldsymbol{\theta}_i - \lambda \frac{\hat{\mathbf{m}}_i}{\sqrt{\hat{\mathbf{v}}_i} + \epsilon} \tag{5.21}$$

where $\boldsymbol{\theta}$ is some parameter (e.g. weights or biases), ∇_θ is the gradient with respect to θ, $\mathbf{m}$ and $\mathbf{v}$ are the first and second moment of the gradient, respectively. ϵ is a small constant > 0 which improves numerical stability. $\beta_1, \beta_2 \in [0, 1)$ are hyperparameters which controls the exponential decay rates of the weighted averages. They are typically initialized as $\beta_1 = 0.9$ and $\beta_2 = 0.999$. $\mathbf{m}_0$ and $\mathbf{v}_0$ are initialized to zero. We see that for the second iteration of the first and second moment in Eq. (5.17)

$$\mathbf{m}_1 = \beta_1 \mathbf{m}_0 + (1 - \beta_1)\mathbf{g}_1 = (1 - \beta_1)\mathbf{g}_1 \tag{5.22}$$

$$\mathbf{v}_1 = \beta_2 \mathbf{v}_0 + (1 - \beta_2)\mathbf{g}_i^2 = (1 - \beta_2)\mathbf{g}_i^2\,, \tag{5.23}$$

and initially the method is biased towards 0 due to our choice of $\mathbf{m}_0 = 0$. The same argument follows for the second moment $\mathbf{v}_1$. To correct this bias we divide Eqs. (5.20) and (5.21) by $1 - \beta_1^i$ and $1 - \beta_2^i$, respectively, this is called the bias correction. This means the first and second moment for the first step becomes

$$\mathbf{m}_1 = \frac{(1 - \beta_1)\mathbf{g}_1}{1 - \beta_1} = \mathbf{g}_1 \tag{5.24}$$

$$\mathbf{v}_1 = \frac{(1 - \beta_2)\mathbf{g}_1^2}{1 - \beta_2} = \mathbf{g}_1^2\,. \tag{5.25}$$

Note that as the number of iterations increase, the bias correction converges to 1. $1 - \beta^i \to 1$, as $\beta^i \to 0$, where we have omitted the subscripts 1 and 2.

5.1.5 Regularization Techniques

While training a neural network we run the risk of overfitting our model to the training data. Overfitting is normally caused by overtraining the model, or due to high complexity (many free parameters) in the model, it can also be a result of the model adapting to noise in the data. A model with many free parameters is prone to learning complex relationships in the input data. An overfitted model generalizes poorly, and the performance on unseen data suffers, while the model performs well on training data. DNNs have many free parameters, and they are prone to overfitting, especially on small data sets. When trying to fit a complex model to a small data set, we could risk having fewer data points than free parameters in the model. For comparison we can consider an n'th degree polynomial, where $n \gg 1$, we know that such a polynomial can fit a data set well, as it can explain complex properties and fit to noise in the training data. A model that is generalized should be able to make good predictions on data that is outside the training set, given that the data are of the same type. Regularization techniques aim to reduce the chance of the model overfitting by penalizing the loss function or removing free parameters. In this study we have applied dropout and weight decay to our model, while also testing alpha-dropout, batch normalization and early stopping.

Dropout

As the name implies dropout is method for dropping nodes during during training, and it was proposed by Hinton et al. [21]. We decide on some probability p, which is typically ≤ 0.5, for dropping a given node. During each iteration of training we sample the nodes in the input and hidden layers with the probability p, deciding whether or not to temporarily drop the node. When a node is dropped, all the connections to that node are dropped as well.

When dropping nodes we are restricting the parameter space of the model, which makes it harder for the model to fit the training data. We can imagine that a single node learns an important feature, which leads to a decrease in the loss, if we remove this node the model will become worse. By dropping nodes during training the remaining nodes are forced to adapt to a varying set of input nodes, as the connections to the features (input values) changes each iteration during training. This will make the model more generalized, as the nodes cannot rely on a subset of features. Dropout also prevents the hidden nodes from learning from the same feature, resulting in shared weights, also known as *co-adaptation*, which typically happens with neighboring nodes. Co-adaptation is a waste of computational power as the co-adapted nodes have learned the same, and they do not contribute to the overall model. Preferably we want the nodes to learn from different features, such that we cover the whole input domain.

Alpha-Dropout

Similar to regular dropout alpha-dropout is a method for dropping nodes during training. Alpha-dropout was introduced by Klambauer et al. [28] as a part of their paper on self-normalizing neural networks. The goal of alpha-dropout is to keep the mean and standard deviation of the input after having dropped nodes. Alpha-dropout works well with SeLU activation function. This method was a part of exploring self-normalizing convolutional neural network, which was not used in this study.

Batch Normalization

When training neural networks it is often beneficial to pre-process the data by scaling the input, such that no features in the input will have inflated importance during training (large values lead to gradients with greater magnitude). Batch normalization, a method proposed by Ioffe and Szegedy [25], is a method for normalizing the linear combination of weights and biases $\mathbf{z}^l$, which is the input to the activation function, by standardization. Standardization is done by subtracting the mean, and then dividing by the standard deviation. Batch normalization is done on each mini-batch separately. The benefits of this method for regularization is that we reduce the problems of vanishing and exploding gradients, leading to a safer learning process. By standardizing during each layer we are minimizing the effects that changing the weights and biases in early layer has on the

nodes in later layers. This can increase convergence speed, as the training data for the later layers are more stable.

Weight Decay

Another way to avoid overfitting is to ensure that the weights do not become very large. If the weights are kept small, the model will not be sensitive to variations in the input, it will not be as sensitive to noise. On the other hand, if some weights in the network are large, we expect that a small perturbation in an input node can lead to a large perturbation in the output. Keeping the weights relatively small suggests a generalized model, i.e. a model that is less sensitive to perturbations in the input.

Weight decay is a method for penalizing the loss function with the square of the Frobenius norm[1] of the weight matrix. This regularization technique is often referred to as L_2 regularization. For a given loss function our new loss then becomes

$$L = \text{loss function} + \frac{\alpha}{2}||\mathbf{W}||_F^2, \tag{5.26}$$

where $\alpha > 0$ is a hyperparameter deciding the strength of the weight decay. By adding the norm of the weights to the loss we are forcing the network to minimize the weights as well as the loss functions, avoiding overfitting.

Early Stopping

During training we expect there to be a point where the loss, evaluated at unseen data (test loss), has a minimum value. After the test loss reaches a minimum value it may stabilize, or increase due to overfitting. Early stopping is a method for aborting training when the loss is at a minimum. The assumption is that our model will not benefit from further training if the test loss starts to increase. By comparing a given number of previous test losses with the current we can decide if our model is overfitting. Early stopping works well if the model is prone to overfitting, but if the loss stabilizes it may be beneficial to continue training, if the loss does not increase. By ending the training early we are effectively restricting the parameter space.

5.2 Convolutional Neural Network

Convolutional neural networks (CNN) is a type of artificial neural network that takes an input structured as a grid, typically an image. CNNs are often used when there are spatial dependencies, making it well suited for image classification and object detection. CNNs are spatially invariant; it does not matter if an object is located at the bottom left or top right. Shifting the input to the CNN should yield the same output, unless the object is

[1] Also called the Euclidean norm or the matrix norm. The Frobenius norm of an $m \times n$ matrix $\mathbf{A}$ is $||\mathbf{A}||_F = \sqrt{\sum_i^m \sum_j^n |a_{ij}|^2}$.

split across boundaries, as CNNs are not periodic. Unlike DNNs CNNs do not have a single weight for each node, but a smaller set of weights which are used in an operation called *convolution*, which will be discussed below. If we were to send an image as an input to a DNN we would have 12288 nodes for an input of dimensions $64 \times 64 \times 3$, which could represent a relatively small RGB image. On top of this we know that there are corresponding weights and biases for each node in a DNN. We see that this quickly adds up to many parameters, especially if we want a deep NN or larger input images.

If we consider a 10×10 RGB image, we know that the image can be represented as a $10 \times 10 \times 3$ array. If this image is passed as an input to a CNN, each element in the input array (pixels) is represented by a node, and it follows that the input layer has 300 nodes. Unlike DNNs, where all the nodes are connected across neighboring layers, the nodes in a CNN in layer l are each connected to a subset of nodes in layer $l-1$. The size of the subset is determined by the *kernel size* in the convolutional layer, and the amount of channels (e.g. 3 for an RGB image). Figure 5.4 shows an outline how the input changes across the layers. We can see the input to a specific node in the next layer, this shows how a single node in layer l gets input from a subset of nodes in the previous layer. We see that as the kernel size increases, we cover a larger area in the previous layer, gathering more information through an increase of parameters, which naturally comes at a computational cost.

CNNs consists of an input layer, one or more hidden layers and an output layer. As for NNs, the output from a hidden layer is passed through an activation function. The output from a hidden layer is typically called an *activation map*. Hidden layers consist of convolutional layers, and in practice they are often followed by a pooling layer, which will be discussed below. The output layer is a fully connected layer, where the input is the flattened output from the last hidden layer, and it outputs one or more values, depending on the task at hand. In our case we have one output value, as the target is the yield stress. The order of operations are as follows

$$\text{input} \rightarrow \text{convolution} \rightarrow \text{activation}(\rightarrow \text{pooling}) \rightarrow ... \rightarrow \text{fully connected}$$

where ... indicates possible more hidden layers.

In the top performing CNNs we often see more than one fully connected layer after the convolution, such as AlexNet by Krizhevsky et al. [29], VGG-16 by Simonyan and Zisserman [46] and ResNet by He et al. [19] to name a few. For a single fully connected layer the output from the last hidden layer is flattened to a vector, which is then passed to a fully connected layer that maps all the features to one or more outputs. This means that the output is a linear combination of all the features from the convolutional layer, which is spatially invariant. By adding another fully connected layer, the nodes in that layer are a linear combination of the output from the previous fully connected layer, which themselves are a linear combination of the output from the convolutional layer, which breaks the spatial invariance. To summarize, by adding more

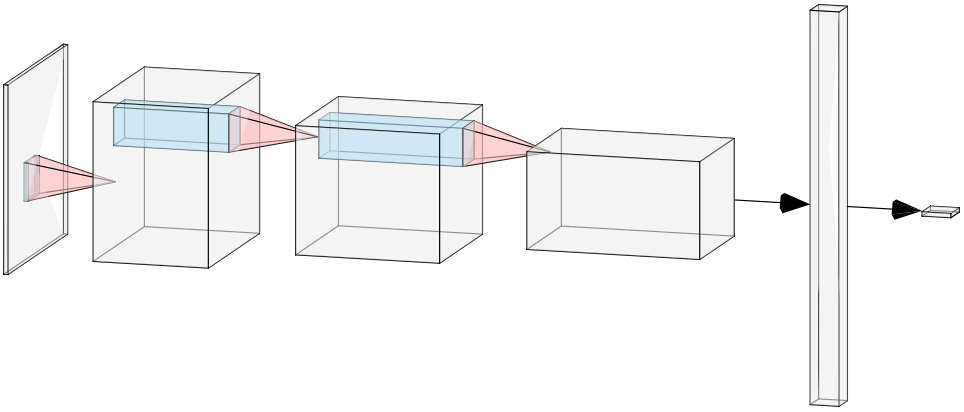


Figure 5.4: Visualization of how the dimensions of the image changes as it is passed through the CNN. Input image is on the left. The blue shaded part shows which features are used in the convolution, which corresponds to the size of the kernel, containing the weights. The red shaded pyramid shows the convolution, taking the blue as input, yielding a single value output. Two columns on the right shows how the input changes through the output layer. The far right is the predicted yield stress, given an input image. Figure made using software by LeNail [30].

fully connected layers after to convolutional layers, we are breaking the spatial invariance of a CNN. CNNs as those mentioned above are trained with the goal of object detection or classification, and it matters little where the the objects are in the image, and where they are relative to one another. In our case it could matter a great deal if a void is located next to another, or if it is far away. Two voids that are very close to one another could act much the same as a single void. In many cases we have voids that are split across boundaries, and we want to avoid that the network learns this as two, three or four separate voids without any knowledge of where they are placed in the image. If a large void is split across the boundaries, the network will most likely learn that two voids half the size has some correlation to large voids.

5.2.1 Convolution – Cross-Correlation

Convolutional neural network is a misleading name, as we are measuring the cross-correlation, not the convolution. However, after this section we will call the operation convolution and not cross-correlation, as this is the practice. The cross-correlation is a mathematical operation that measures the overlap of two functions, when one functions is shifted over the other. The cross-correlation of functions f and k is given by

$$(k * f)(\mathbf{x}) = \int k(\mathbf{z}) f(\mathbf{x} + \mathbf{z}) d\mathbf{z}\,, \tag{5.27}$$

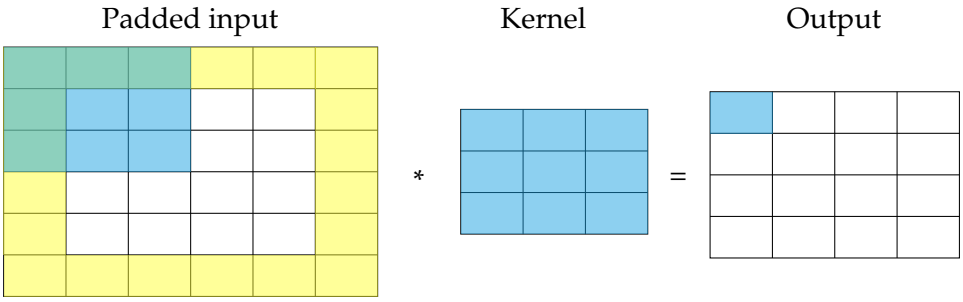


Figure 5.5: Illustrating the cross-correlation between input with zero padding of thickness 0 (shaded yellow) and the kernel, which results in a single value for the output.

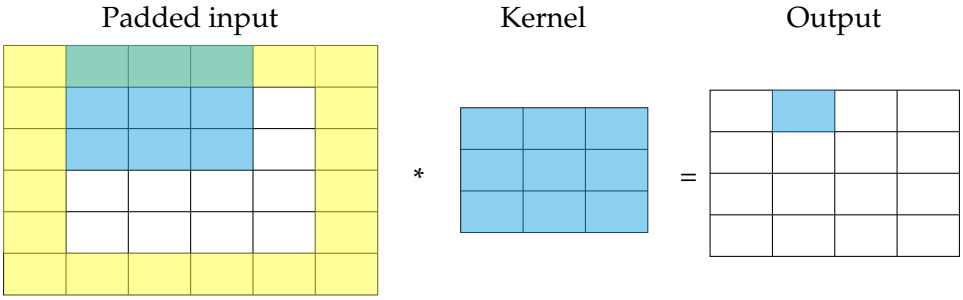


Figure 5.6: Illustrating the cross-correlation between input with zero padding of thickness 0 and the kernel, which results in a single value for the output.

if we assume k has no imaginary part, such that $k^*(\mathbf{z}) = k(\mathbf{z})$. Convolution is similar to cross-correlation, the difference being the sign in the last parenthesis, if we assume both functions are real. Had the function k had an imaginary part, we would have taken the complex conjugate. In the case of discrete values the integral takes the form of a sum. For CNNs we can consider the left hand side in Eq. (5.27) as the output from the convolutional layer, which is fed to the activation function, the function $f(\cdot)$ is the input to the convolutional layer, while $k(\cdot)$ is the kernel. While DNNs have shared weights, CNNs have locally shared weights. The kernel is a matrix containing the learnable weights. For a two dimensional input the output from the convolutional layer is then found by

$$z[i,j,c] = \sum_{r=0}^{K-1}\sum_{s=0}^{K-1}\sum_{d=0}^{C_{in}-1} K[r,s,d]\cdot x[i+r,j+s,d] + b[d]\,, \tag{5.28}$$

for kernel of size $K \times K$, C_{in}, C_{out} is the number of channels in the input and output, respectively, $c \in 0, ..., C_{out} - 1$ and b is the bias. Note that the bias is a constant for each channel. In Figure 5.5 we can see a visualization of $z[0,0,0]$. Note that the kernel is moved across the image to produce an output, the cross-correlation for $z[0,1,0]$ is shown in Figure 5.6.

The convention is to use an odd kernel size and pad the input to the cross-correlation with zeros, in such a way that the output is of the same size as the input. The thickness of the padding can be decided by $(K-1)/2$,

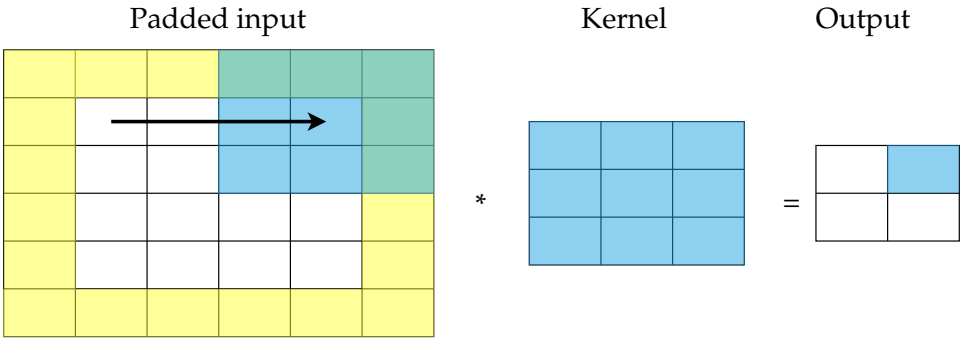


Figure 5.7: Illustration of stride equal to 3. We observe that the output is decreased in size.

for a kernel of size K. In Figure 5.5 we can see how padding is applied for a 3×3 kernel. *Padding* is typically added to preserve the height and width of the image through cross-correlation. We see that if we increase the filter size from 3 to e.g. 5, we increase the padding from 1 to 2, respectively, if we want to keep the shape of the image.

Stride

In some cases we might be working with large images that contain little information, e.g. an image where only a fraction is in focus, while the remaining image is blurry. In such a case it would be natural to consider searching the input in a more coarse fashion. *Stride* is a parameter that decides the step length during cross-correlation. Figure 5.7 shows an example of stride 3, where the arrow shows us how many pixels we move the kernel across the input. We observe that the output becomes smaller when we increase the stride. Stride can be useful for down sampling or to reduce the computational cost. In our case we did not increase the stride above the default value of 1.

5.2.2 Receptive Field

We know that each node in a hidden layer is a linear combination of previous nodes, which themselves may be linear combinations of previous nodes as well. As we increase the depth of a NN we also increase the number of nodes that contribute to the value of a given node, which corresponds to increasing the spatial region that the node gets input from. *The receptive field* refers to the number of input nodes a given node has. Let us consider an example of a CNN with a 3×3 convolutional kernel, N_{in} inputs and 2 hidden layers. If we consider a node in the first hidden layer, it has input from 3 nodes in the input layer. A given node in the second hidden layer has input from three nodes in the first hidden layer, which all have contributions from three nodes each, where 3 nodes overlap, bringing it to a total of 5 nodes. A visualization of the receptive field is shown in Figure 5.8.

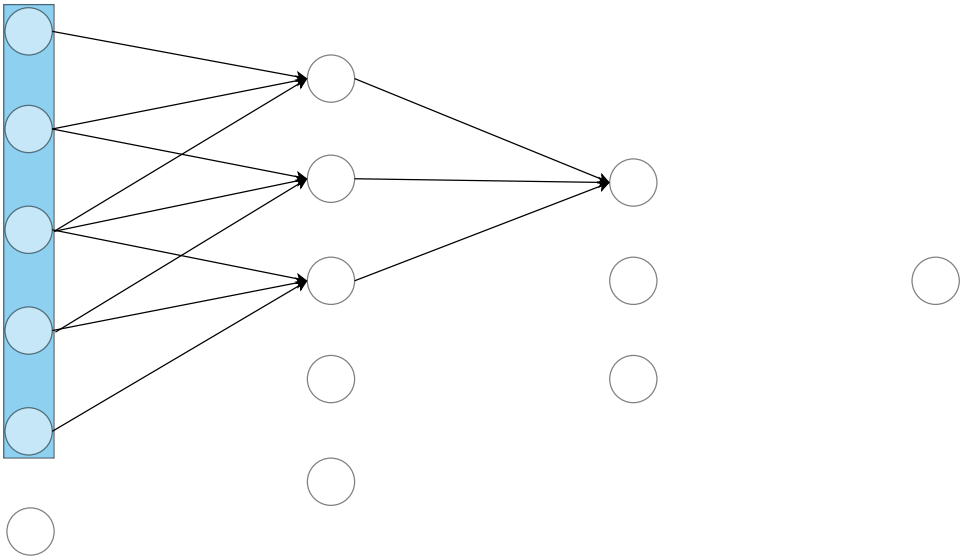


Figure 5.8: Illustration of receptive field for a given node in a CNN with kernel of size 3. The receptive field is shaded in blue. The arrows shows connections between nodes.

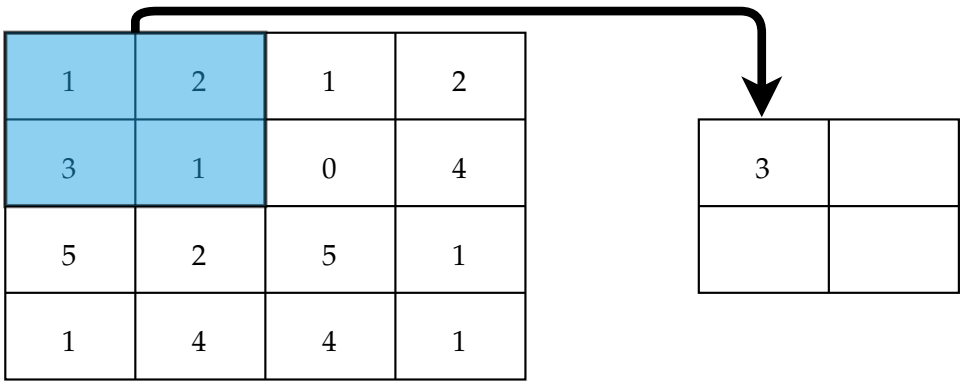
5.2.3 Pooling

By the convolutional layers we store more and more information in channels, or the depth of the image, while keeping the height and width of the image constant, given a proper choice of stride. Our goal is to store features that are important considering the target value. In our case we want to find features in the cut space for a given structure, which are significant for the strength of the material. Pooling is an operation that acts on a small subset of the activation map, typically a 2×2 subset, which decreases the spatial size of the input. The most used pooling operation is *max pooling*, which is an operation that finds the maximum value of the subset on the activation map, which is then cast to a new input. The idea with max pooling is that you locate the important features within a subset, eliminating the less important ones. By important features, we mean the features that has the largest activation. In Figure 5.9 we can see an example of max pooling with stride 2 for the first 5.9a and second pooling 5.9b. For pooling stride works the same way as for cross-correlation, but the typical choice is 2 for 2×2 pooling. If we choose stride 1 for 2×2 pooling, we could run the risk of choosing a feature that was ignored by the former pooling, or we could choose the same feature twice. The safest choice is to use the same stride as the size of the pooling. Another pooling option is to use average pooling, which maps the average value of a subset of the activation map to a new activation map.

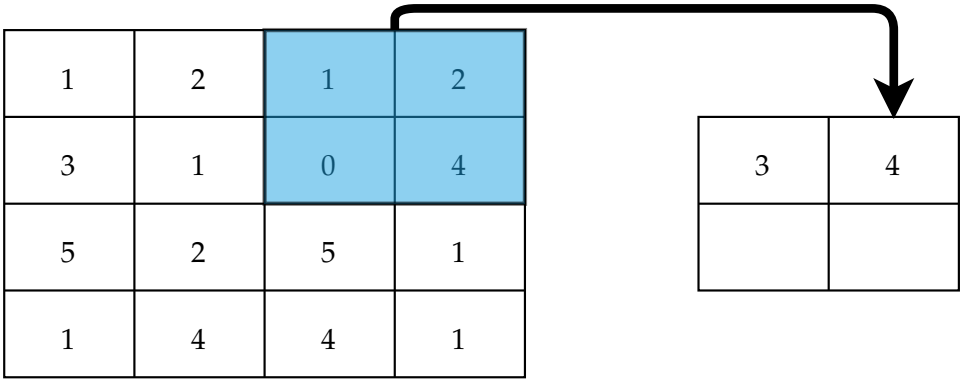
Pooling is an operation that does not break the translational invariance. In Figure 5.10 we can see max pooling applied to an image three times. We can consider max pooling as a compression which keeps the most important features, or the largest activations. By reducing the activation

map through pooling we are effectively increasing the receptive field, as some nodes form new connection with other nodes.

In our case we have applied max pooling, which is the source of the decreased height and width seen in Figure 5.4



(a) First step.



(b) Second step.

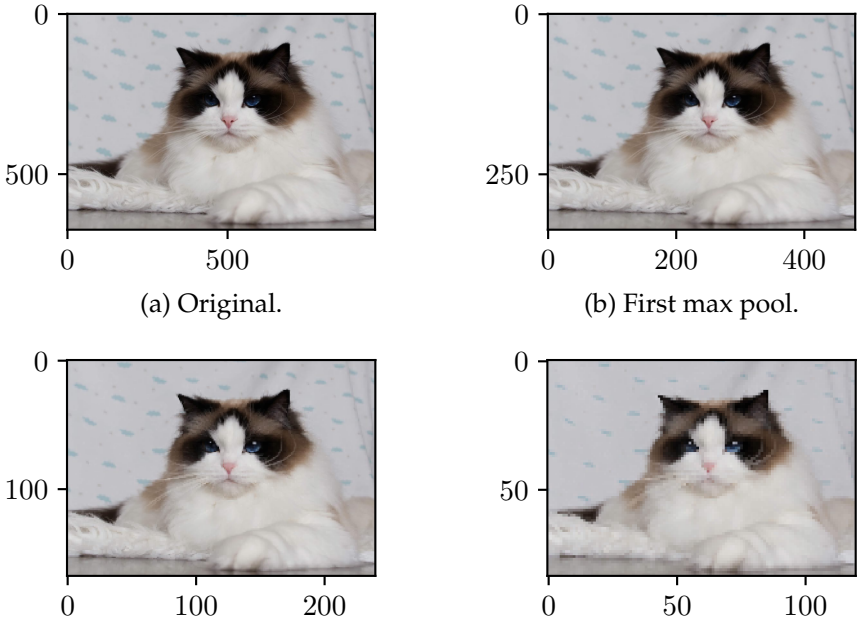
Figure 5.9: Visualization of 2×2 max pooling, with stride 2. We see that the largest value is chosen as the output.

5.2.4 Self-normalizing Convolutional Neural Network

Self-normalizing neural networks (SNN) were proposed by Klambauer et al. [28], we consider applying the self-normalizing methods on a convolutional neural network (SCNN). SNN consists of fully connected layers which are followed by an alpha-dropout layer, which are then passed through SELU activation function. We applied alpha-dropout and SELU on a CNN to test if self-normalization had any impact on the performance of our model. The hidden layer were as follows

$$\text{convolution} \rightarrow \text{alpha-dropout} \rightarrow \text{SELU} \rightarrow \text{max pooling} \rightarrow \dots,$$

which was then passed to a fully connected layer with a single output value.



(a) Original.

(b) First max pool.

(c) Second max pool.

(d) Third max pool.

Figure 5.10: Max pooling applied three times on an image of the authors cat, Sally. Top left image is the original. The max pooling is applied to each color channel separately. We see that the amount of pixels in the image decrease as we apply max pooling. For third max pooling we see that some details have disappeared, but important features such as whiskers, snout and the overall shape of the cat is present.

5.3 Dimensionality and Noise Reduction

To reduce the dimensionality or noise of the samples we can apply unsupervised machine learning methods to create new representations of our data. The methods we have considered learns effective representations of the data in a lower dimensional space. What separates the methods is how they aim to reduce the dimensionality.

We have considered supervised and unsupervised principal component analysis (PCA), autoencoder and convolutional autoencoder. For the case of convolutional autoencoder we also added a linear layer at the end, proposing a convolutional autoencoder neural network, which acts as a supervised convolutional autoencoder.

By reducing the dimensionality of our data we hope to increase the computational time, while maintaining good results when applying CNN on our data.

5.4 Principal Component Analysis

The idea of principal component analysis is to reduce the amount of variables per sample, known as dimensionality reduction. It can also be used for compressing images, by inverse transformation using the principal components. PCA extracts features that are correlated, which typically have an explanatory value. By eliminating features that have little explanatory value we decrease the possibility of overfitting. The idea of PCA is to express the variance of the data in so-called principal components. There are as many principal components as there are features. The first principal component is found by fitting a line that maximizes the variance to the data. The second component is found in the same way, under the criterion that it is perpendicular to the first component, i.e. that it is uncorrelated with the first principal component. The first component contains most of the variance in the data, and the second contains the second most, and so on. By using multiple principal components we can increase the proportion of variance. If the first component explains 0.2 of the variance, and the second 0.1, we can build a model with two principal components which explains 0.3 of the total variance. In practice we use as many principal components as necessary to get a specific proportion of variance.

To find the principal components we first and foremost center the data by subtracting the mean, and then divide by the standard deviation of each sample:

$$\mathbf{Z} = \frac{\mathbf{X} - \bar{\mathbf{X}}}{\text{std}(\mathbf{X})}, \tag{5.29}$$

where $\mathbf{X}$ is a matrix containing data, structured in rows. We then calculate the covariance matrix of $\mathbf{Z}$ by $\text{cov}(\mathbf{Z}) = \mathbf{Z}^T\mathbf{Z}$. Then we find the eigenvectors $\mathbf{V}$ and eigenvalues $\mathbf{d}$. The principal components are then found by

$$\mathbf{C} = \mathbf{Z}\mathbf{V}. \tag{5.30}$$

We can then transform our data by multiplying the input by the transpose of the principal components

$$\mathbf{X}' = \mathbf{X}\mathbf{C}^{\mathbf{T}}, \tag{5.31}$$

to produce a feature reduced data set with dimensions $(n_{\text{samples}}, n_{\text{components}})$. If we choose enough principal components, we should cover most of the variance of the data set. In fact, we could cover all the variance of the data set by using a fraction of total principal components. PCA is a popular method which is often used in data science, its simplicity and ability to explain data by reducing the features makes it well suited for many tasks.

5.4.1 Supervised Principal Component Analysis

Standard PCA is an unsupervised machine learning algorithm, as discussed previously this means the algorithm does not consider the corresponding target of an input. Ideally we would want the dimensionality reduction to happen such that it eliminates features that are unimportant when considering the target, or in our case the yield stress. We therefore consider supervised PCA (SPCA). SPCA adds one step to PCA that eliminates features based on univariate regression coefficients for each sample. After this step SPCA does the same as PCA. The steps are as follows

1. Standardize data
2. Calculate regression coefficients
3. Eliminate features that has a coefficient which is lower than some threshold
4. Perform PCA on the feature reduced input

We find the regression coefficients by linear regression and set a threshold of 1. The feature reduced images were then treated by standard PCA as discussed above. SPCA can be used to estimate which features are important for maximizing the output. The principal components in SPCA have high variance as for standard PCA, but they also have correlations with the target value.

5.5 Autoencoder

The goal of autoencoders is to find an effective lower dimensional representation of the input. This is done by creating efficient coding of the input, which is then decoded. The decoded data is transformed to an image of the same size as the input. Autoencoders are capable of learning complex, nonlinear, relationships of the input features, unlike PCA which tries to represent the data in a space which is one dimension lower, also known as a hyperplane. If there are correlations in the input features an autoencoder can be used to learn these features. Autoencoders falls under the branch of unsupervised learning, as there is no target value

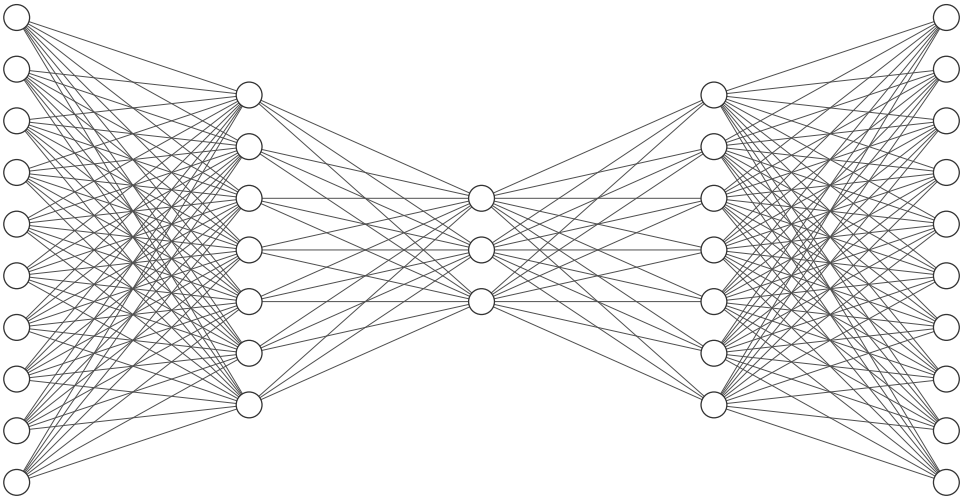


Figure 5.11: Visualization of an autoencoder. Input is on the left and output on the right. The input is encoded towards the middle layer, and then decoded towards the output layer, so to match the input. Figure made using software by LeNail [30].

which the algorithm optimizes towards. They are often used for noise and dimensionality reduction. Autoencoders consist of an encoder and a decoder, which both consists of one or more hidden layers, each layer is followed by an activation function. Each layer in the encoder reduces the amount of nodes, in the same way that a DNN does. The encoded input is then sent through the decoder, where each layer increase the number of nodes (opposite of a DNN). The loss function for autoencoders is the MSE of the input and decoded output (learned representation), this is also called the reproduction loss. The motivation of the network is to reproduce the input by optimizing the reproduction loss.

In Figure 5.11 we can see a visualization of the nodes in an autoencoder. We observe that the nodes are reduced and then increased to the same amount of nodes as the input.

5.5.1 Convolutional Autoencoder

We also looked at a different autoencoder that uses convolution to store information about the image in the channels. We also apply max pooling, which reduce the amount of nodes, and forces the network to choose important/strong features. As with the autoencoder discussed above we motivate the network by the reconstruction error.

The encoder consists of two or more convolutional layer. The first convolutional layer reduces the width and height of the input, while increasing the channels. There can follow more hidden layers which expand the channels, and further reduce the height and width of the input. The last convolutional layer reduces the channels to the same amount as the input, this is the output of the encoder. Each convolutional layer is followed ReLU activations and a max pooling.

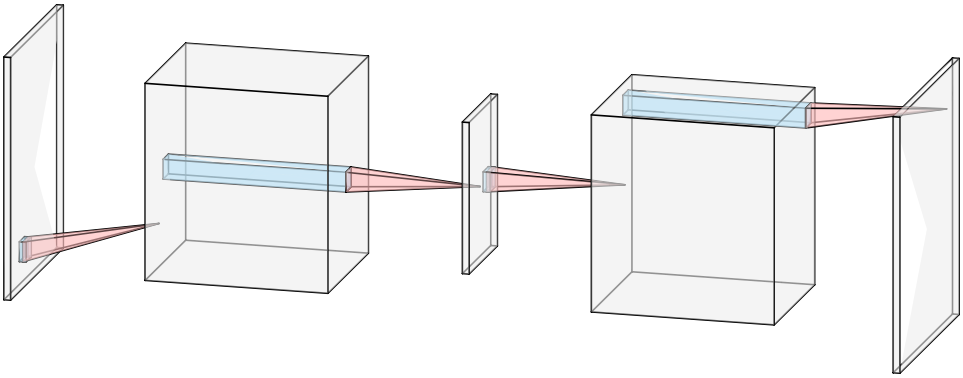


Figure 5.12: Visualization of a convolutional autoencoder. Input is on the left and output on the right. The input is encoded towards the middle layer, and then decoded towards the output layer, so to match the input. Figure made using software by LeNail [30].

The decoder is set to do the opposite of the encoder. The convolutional and max pooling layers are swapped with transposed convolutional layers, where the first is followed by ReLU activation, while the last hidden layer is followed by a Sigmoid activation to force the values between 0 and 1. A visualization of the layers in a convolutional autoencoder is shown in Figure 5.12.

Transposed convolution is an operation where the kernel is multiplied by a single feature from the input, which makes up a part of the output from the transposed convolutional layer. In Figure 5.13a and 5.13b we can see an example the first and second step for transposed convolution, respectively. Note that the overlapping features in the output are summed together.

As with the standard autoencoder we can make a new dataset consisting of the reduced images from the convolotional autoencoder, by extracting the output from the encoder. Convolutional autoencoders gives us an opportunity to compress the input to our CNN, which will save computational cost. Unlike standard autoencoders, convolutional autoencoders are spatially invariant.

5.5.2 Convolutional Autoencoder Neural Network

We propose a NN topology that adds a fully connected layer at the end of a convolutional autoencoder. We aim to make the autoencoder semi-supervised, such that the encoding and decoding is motivated by the target value. If this method succeeds we could be able to use this method to locate what parts of the structure contribute to maximizing the output, i.e. locate what voids are important for maximizing the strength. Initially we considered the loss to be the MSE of the target and predicted yield shear stress. The weakness of this model is that the convolutional autoencoder part is not really motivated by any reproduction error, so we are not certain what it actually does. We also considered a different approach to

motivate the network by considering the reproduction loss as well as the loss of the prediction and the target value, by simply adding them together. For each training iteration we extract the output from the decoded layer, which is used to calculate the reproduction loss. We also calculate the loss with respect to the prediction and target yield shear stress. These two losses added up serves as the total loss of the network. In short we ask the network to optimize both the convolutional autoencoder and a fully connected layer which takes the decoded image as input. Since the method is semi-supervised it should remove features that are less important for the output. In Figure 5.14 we can see a visualization of the convolutional autoencoder neural network.

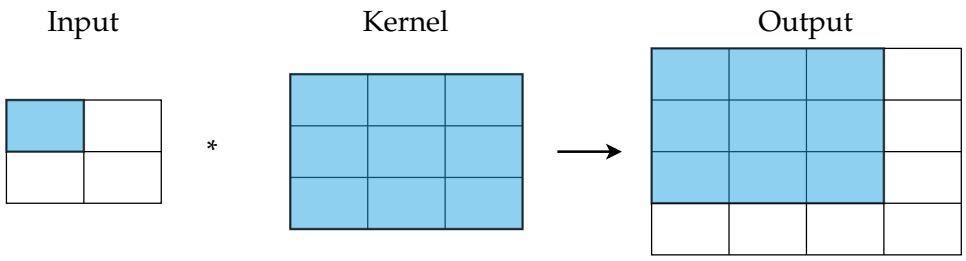
5.6 Explainability

Explainable artificial intelligence (XAI) is a large field within researching how to explain what and how AI is learning [1]. The non-linearity of neural networks makes it difficult to pinpoint what is being learned. For certain applications, e.g. medical, it might be unacceptable to apply a model that is not fully transparent. Beyond the input and output, neural networks are essentially like a black box. We have little control over what happens during learning, we can describe how the nodes are updating through optimizations, but what happens from node to node is a mystery. The why and the how a NN arrives at a specific decision is hard to explain. There is often a large number of parameters, which only increases the difficulty of trying to explain the decisions the model has taken. We can visualize the magnitude of the activations and follow the path from the output to the input, which gives us some insight into what the network deemed as an important feature to when trying to explain the output.

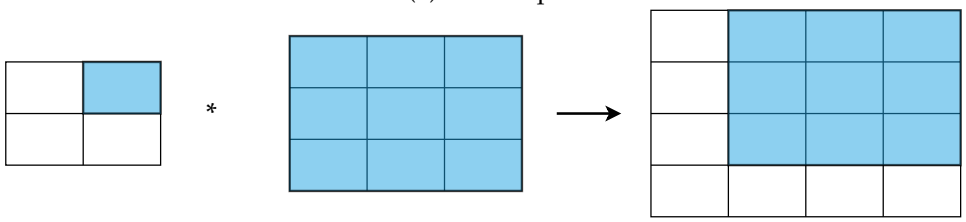
5.6.1 Saliency Maps

Saliency maps gives insight into which features of the input image is the most important for the yield stress. It shows what features effects the output the most, if they are subjected to a small perturbation in its value. If a feature has a high correlation with the output, we know that the weights have large values. This means that if perturb that feature by slightly changing the value, we are likely to see a large change in the output. Let us consider a classification case where we want to classify cats as an example. It is natural to consider that the whiskers, shape of the ears and tail are important for a classifier when trying to classify a cat. If we change the pixels of the input image, such that the cat lacks an ear by modifying the color to match the background, we can imagine that the model would be less certain that that there is a cat in the input image.

Saliency is a method for propagating the gradient of an input backwards through the model. Saliency maps was first proposed by Simonyan et al. [47] as a method for visualizing the rank of the pixels in the input image, based on their influence on the score of a classifying CNN. The output



(a) First step.



(b) Second step.

Figure 5.13: Visualization of transposed convolution. In the output we see that there are overlapping values, this is handled by summation.

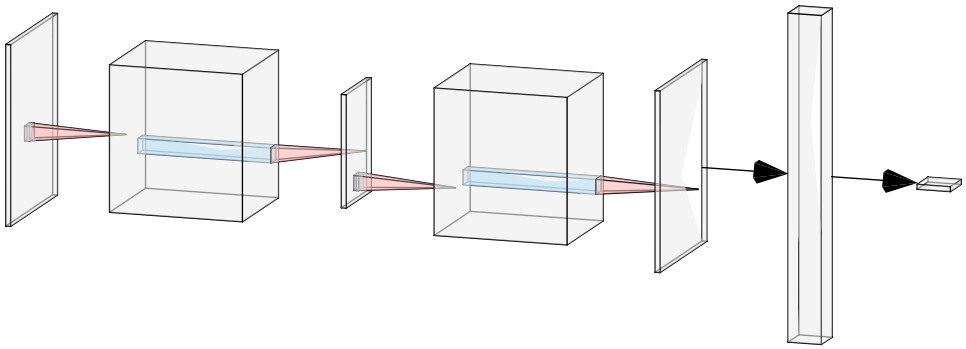


Figure 5.14: Visualization of a convolutional autoencoder neural network. Input is on the left and output on the right. The input is encoded towards the second hidden layer, and then decoded towards the fourth hidden layer, so to match the input. The output from the decoder is then flattened and passed through a fully connected layer. Figure made using software by LeNail [30].

of a deep network, given a flattened input image $\mathbf{i}$, can be written

$$O(\mathbf{i}) \approx \mathbf{W}^T\mathbf{i} + \mathbf{b}\,. \tag{5.32}$$

For a one layered network the equation above is not an approximation, as there is a direct path from the output node to the input. As the network becomes deeper, there is an increasing amount of weights and biases that contribute to each node, making the $O(\mathbf{i})$ a non-linear function, as stated by Simonyan et al. [47]. Simonyan et al. [47] therefore proposed to use a first-order Taylor approximation of the output $O(\mathbf{i})$, where $\mathbf{w}$ is found by the partial derivative of the output with respect to the input

$$\mathbf{W} = \frac{\partial O}{\partial \mathbf{i}}\,. \tag{5.33}$$

Following is the code for creating a saliency map for a given image:

```
model.eval()                    # Evaluate the model
image.requires_grad_()          # Record autograd operations

prediction = model(image)       # Run the image through the CNN
prediction.backward()           # Backpropagate the image

# Extract the maximum values across channels
saliency = torch.max(image.grad.data.abs(), dim=1)[0, 0]
```

Saliency maps were compared with the stress field and displacement of the particles to see if the network had learned any physics.

5.7 Searching for New Material Structures

Due to the computational cost of performing molecular dynamics simulations we cannot build a solid dataset for our model in reasonable time. Hanakata et al. [15] proposed a search algorithm where they started with a small dataset, on which they trained a CNN, creating a model on the base data set. The model was then exposed to all of the cut configurations, and made predictions about the yield stress. They then chose the configurations (in our study referred to as geometries) that were predicted as strongest and perform MD simulations to get the realized yield stress, these 100 samples constitute the first generation. The next step was to add these new 100 samples to the full data set and retrain the CNN, and sample the next generation as was with the first generation. This type of algorithm is sometimes referred to as a genetic algorithm, inspired by natural selection. You start with some initial population on which you perform an evaluation for some property, this evaluation is used to perform a selection of new individuals for the population. The goal is to create a population which has for each evaluation and selection strengthens the property. For this study we apply a similar generative search algorithm as Hanakata et al. [15], although our goal is not the same, as we cannot perform MD simulations on all the 2^{5000} combinations, so we cannot check if our model actually finds the geometries with the highest yield. Instead we build a base data set of 3366

samples, which is used to train a CNN. We then expose the trained network to close to 3 million unseen geometries of simplex noise. We choose the 100 strongest samples which has a porosity in the interval $p = 0.15 \pm 0.01$ which are predicted as strongest. We realize the 100 samples and retrain the network. This is done 10 times, where each generation sees a new set of seeds for creating simplex geometries, meaning we have exposed the network to a total of 30 million geometries during the generations. Having increasing our data set to 4366 samples, we try to retrain with the 30 million samples, generating five new generations. A final attempt to find a significantly stronger structure is done by exposing the network to close to 75 million unseen geometries. We will generally refer to this method as the search algorithm.

5.8 Evaluating the Performance of Machine Learning Methods

During our introduction to autoencoders we mentioned reproduction error as a means to motivate the learning during self optimization. We can easily see that since we are comparing images, the closer to 0 the error is, the better. For other machine learning methods we also measure the MSE, but we also look at other ways to evaluate how our methods perform.

I section 3.11 we discussed the MD precision. This will serve a comparison for our results. We cannot expect our model to have a MSE that is lower than the overall precision of our calculations. When we are choosing samples for a specific yield stress we will use the MD precision to compare against the standard deviation of the distribution.

5.8.1 Train/test splitting

When evaluating the performance of a model it is important to evaluate it on data it has not been trained on. We cannot determine if a model is overfitted if we evaluate it on the same data that it has been trained on. Instead of allowing the model to pat itself on the back by evaluating the performance on the same data it has been trained on, we challenge it by exposing it to data it has not seen. The normal approach is to split the data set into subsets, where each subset is dedicated to training and testing the model. In some cases there is also a third subset which is used for validation. The splitting is usually 80% and 20% (or 10% and 10% if validation is used), respectively. The training subset is naturally the largest. The test set is used during training to evaluate the model on-the-go, while the validation set is used to evaluate the model at the end. In our case we have only used a test set.

5.8.2 Coefficient of Determination – R^2

To determine how well the model model fits to the data, one typically looks at the coefficient of determination, popularly called R^2 score. R^2 tells us

how well the model explains the variance in the data. It is is defined as

$$R^2 = \frac{\text{TSS} - \text{RSS}}{\text{TSS}}, \tag{5.34}$$

where TSS - total sum of squares is the squared distance of the targets y_i and the mean value $\bar{y}$, and it is a measure of the variance of the data, and is given by

$$\text{TSS} = \sum_i^N (y_i - \bar{y})^2 . \tag{5.35}$$

RSS - residual sum of squares is the squared distance of the data and the predictions $\hat{y}_i$, and it measures the unexplained variance after fitting the model, and is defined as

$$\text{RSS} = \sum_i^N (y_i - \hat{y}_i)^2 , \tag{5.36}$$

where both the sums above run of the number of samples N. TSS $-$ RSS, being the unexplained variance due to fitting subtracted from the variance in the data, measures how much of the variance is explained by our model. Ideally our model explains none of the variance and the RSS is zero, which means our model explains the data perfectly, yielding an R^2 score of 1. On the other end the R^2 score can be close to 0, which means our model explain much of the variance, and $RSS \approx TSS \Rightarrow R^2 \approx 0$.

Chapter 6

Processing and Analyses on Simulation Data

We will be executing thousands of simulations, which all produce data that needs to be analyzed with different methods. All of our MD simulations were performed using LAMMPS, in the following sections we will discuss how we measured relevant quantities using LAMMPS and how we obtained the data. We will also discuss pre-processing of the data prior to fitting our model with NNs, as well as post-processing.

For this chapter we gathered information about Savitzky-Golay filtering from Dombi and Dineva [9, p. 447-449] and Press et al. [43, p. 766-772].

6.1 Padding Input for Convolutional Neural Networks

In section 5.2 we discussed how CNNs are not periodic, the kernel does not consider any pixels that are outside, and there is no wrapping functionality for kernels in the most popular machine learning libraries for Python. The periodic noise creates some issues for the machine learning model. We know that our voids can be split across boundaries, as we made the noise periodic, which the CNN will interpret as two or more separate voids, depending on the placement of the void. Ideally we would want the CNN to consider the whole void and learn how they are shaped and how it effects the strength. Griffith's theory of brittle fracture states that the strength of a system is proportional to the largest crack, or in our case void, so it could be important that we allow the network to see the whole size of the void. To allow the CNN to see the full voids, we pad the data by $1/4$ in each direction, using Numpy's padding function with wrapping. In Figure 6.1 we can see an example of a padded and unpadded input. A different approach was to pad the input by $(K-1)/2$, where K is the kernel size, assuming odd K, and set the padding in the CNN to be 0 for the first layer, and then $(K-1)/2$ as usual for each successive layer. By padding the input this way, instead of allowing for padding by zeros in the first convolutional layer, we expose the first convolution to information instead of zeros.

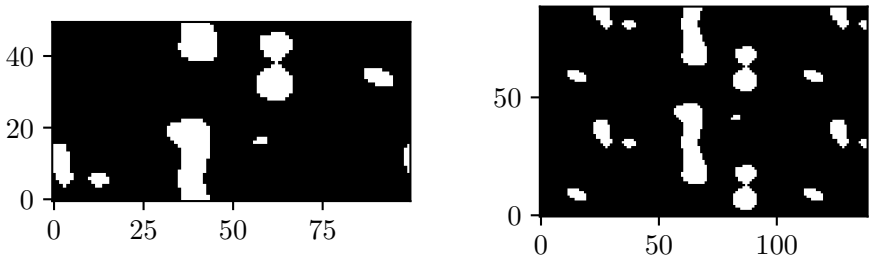
6.2 Shuffling Voids

If the strength of the system is decided by other factors than the porosity, we expect that it matters how the voids are placed in the cut space, especially how they are placed relative to one another, as discussed in the end of section 5.2. If we choose one structure, which has a corresponding yield stress, we expect that when shuffling the voids we should observe a change in the yield stress. The methodology we have chosen is to extract each void from a single structure and save them in a three dimensional array, where the first dimension correspond to the number of voids, and the second and third contains a single void.

Since our geometries are binary arrays, we can apply methods to label each subset that represents a void (which are blobs of value 1). Scipy has a method for applying labels to matrices. We have to show care when labeling, due to periodic boundaries. An example of labeling on an 8×8 matrix is shown below.

$$\begin{bmatrix} 0 & 0 & 0 & 0 & 1 & 1 & 1 & 0 \\ 0 & 1 & 1 & 0 & 0 & 1 & 0 & 0 \\ 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 & 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 \end{bmatrix} \rightarrow \begin{bmatrix} 0 & 0 & 0 & 0 & 2 & 2 & 2 & 0 \\ 0 & 1 & 1 & 0 & 0 & 2 & 0 & 0 \\ 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 3 & 3 & 0 & 0 & 0 & 0 \\ 0 & 3 & 3 & 3 & 0 & 0 & 0 & 0 \\ 0 & 0 & 3 & 3 & 0 & 4 & 4 & 0 \\ 0 & 0 & 0 & 0 & 0 & 4 & 4 & 0 \end{bmatrix}.$$

We see that the one void which is split across the y-boundary has been labeled as 2 and 4. To solve this we iterate along the boundary at $y = 0$ for x and $x = 0$ for y, if there are values at the at $y = 0$ and $y = 7$ for a given x, we swap the values for the labels at $y = 7$, i.e. we interchange the values of



(a) Original Simplex noise.

(b) Padded Simplex noise.

Figure 6.1: Comparison of padded and unpadded Simplex noise.

4 to 2, such that the labeled matrix becomes

$$\begin{bmatrix} 0 & 0 & 0 & 0 & 2 & 2 & 2 & 0 \\ 0 & 1 & 1 & 0 & 0 & 2 & 0 & 0 \\ 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 3 & 3 & 0 & 0 & 0 & 0 \\ 0 & 3 & 3 & 3 & 0 & 0 & 0 & 0 \\ 0 & 0 & 3 & 3 & 0 & 2 & 2 & 0 \\ 0 & 0 & 0 & 0 & 0 & 2 & 2 & 0 \end{bmatrix}.$$

Since we are extracting each void, we made a choice to move the void if it was split across boundaries (hereby called a split void), this was done for practical reasons. Each time we register a swap of values, as discussed above, we also register that we have to roll the array containing the split void. By rolling we mean shifting the values and indices of the matrix in a controlled manner, e.g. $[0,1,0] \rightarrow [0,0,1]$. The code for both operations is as follows

```
# Lw is the labeled image, num is the number of labels
lw, num = scpi.measurements.label(image)

# List to keep track of which labels to roll
roll_label = [[], []]

# For y = 0
for i in range(lw[:, 0].shape[0]):
    # If boundary value is non-zero
    if lw[i, 0] != 0 and lw[i, -1] != 0:
        # then we change the values for the lower void
        lw[lw == lw[i, -1]] = lw[i, 0]
        if lw[i, 0] not in roll_label[0]:
            # Register the need to roll, unless it has already
            # been registered
            roll_label[0].append(lw[i, 0])
# For x = 0
for i in range(lw[0, :].shape[0]):
    if lw[0, i] != 0 and lw[-1, i] != 0:
        lw[lw == lw[0, i]] = lw[-1, i]
        if lw[-1, i] not in roll_label[1]:
            roll_label[1].append(lw[-1, i])
```

Listing 6.1: Python code to correct the labeling of voids due to periodic boundary conditions.

Following this the idea is to save the voids to an array as stated. For the cases where the void is split, we roll the array until the void is not split. Below is a snippet of the code for rolling

```
if n in roll_label[0]:  # Roll if registered rolling for x
    c = 0               # Counter for rolling
    while (scpi.measurements.label(img)[1] != 1
            and c < img.shape[0] // 2):
        # Roll when there are more than 1 label, or if we
        # have rolled over half the image
        img = np.roll(img, shift=1, axis=1)
```

```
        c += 1
        if n in roll_label[1]:  # Roll if registered rolling for y
            c2 = 0
            while (scpi.measurements.label(img)[1] != 1
                    and c2 < img.shape[1] // 2):
                img = np.roll(img, shift=1, axis=0)
                c2 += 1
```

Listing 6.2: Python code to roll voids if they are split across boundaries.

The code in Listing 6.2 is executed for every label, and it covers rolling in the x-direction and y-direction (if necessary). There is also a section which covers y-direction. Having extracted each void we have a three dimensional matrix with each void. To shuffle the asperities we simply roll each void a random number of pixels and sum along the first dimension (number of voids), making sure there is no overlap.

6.3 Void Analysis

To study the voids as a function of the yield stress we look at the number of voids, the relative area of the largest void. To get the total amount of voids we apply the same methods as discussed in the previous section and shown in Listing 6.1 to label the voids, then we count the number of voids by their corrected labels. The area of the largest void is found by summing up the amount of pixels for each labeled void. The area which is largest corresponds to the void which covers the most pixels. The relative area is then found by dividing by the total amount of pixels in the image.

6.4 Smoothing

The log output data from LAMMPS is averaged across all particles for each time step to create a single volume for the whole system, e.g. temperature, pressure. The raw data can be noisy, making it difficult to locate the point of yield. The result we are extracting might also be effected by noise, making the data point an anomaly. Instead of using the raw data points, we can represent it by some local averaged value, based on surrounding data points. By smoothing the data we are effectively averaging over a given number of data points, which should yield a better representation of the data. We have chosen Savitzky-Golay filtering as our method for smoothing.

6.4.1 Savitzky-Golay Filter

Savitzky-Golay is a method that fits a low level polynomial to a subset (window) of data. The minimization is done by least squares error. A moving window averaging finds the smoothed value g_i by

$$g_i = \sum_{n=-n_L}^{n_R} c_n f_{i+n}\,, \tag{6.1}$$

where n_L and n_R are the number of data points to the left and right of data point i, respectively, and f_{i+n} is the raw data at point $i+n$. Using a moving average window with a constant $c_n = 1/(n_L + n_R + 1)$ performs well on linear functions. If the functions are non-linear, moving window average always decreases the amplitudes and width of maxima, which means our smoothing has become biased. For our case the amplitude has physical importance. Savitzky-Golay filtering tries to fit a polynomial, rather than a constant, for the moving window. A polynomial of degree p can be written as

$$P = a_0 + a_1(x - x_i) + a_2(x - x_i)^2 + ... + a_p(x - x_i)^p \,. \tag{6.2}$$

Savitzky-Golay aims at finding the coefficients by minimizing the least squares error. The matrix equation we wish to solve is

$$\mathbf{J} \cdot \mathbf{a} = \mathbf{y} \tag{6.3}$$

where

$$\mathbf{J} = \begin{bmatrix} 1 & (x_{i-n_L} - x_i) & \cdots & (x_{i-n_L} - x_i)^p \\ \vdots & \vdots & \vdots & \vdots \\ 1 & 0 & \cdots & 0 \\ \vdots & \vdots & \vdots & \vdots \\ 1 & (x_{i+n_R} - x_i) & \cdots & (x_{i+n_R} - x_i)^p \end{bmatrix} , \tag{6.4}$$

and $\mathbf{a} = (a_0, ..., a_p)$ and $\mathbf{y} = (y_{n_L}, ...y_i, ..., y_{n_R})$ is the raw data. Left multiplying Eq. (6.3) by $\mathbf{J}^T$, and then multiplying by $(\mathbf{J}^T J)^{-1}$ to solve for the coefficient vector $\mathbf{a}$, we find the coefficients by the least squares estimate

$$\mathbf{a} = (\mathbf{J}^T\mathbf{J})^{-1}(\mathbf{J}^T\mathbf{y}) \,. \tag{6.5}$$

The value of the polynomial fit at g_i is then used as the smoothed value for the raw data point i. The method requires the window to be an odd number long, as the smoothed value is center value of the window. The method performs well when it comes to preserving the amplitude, given that we show care when choosing the window size, as it is sensitive to the size of the window. We chose Savitzky-Golay filter because it performs well at preserving the amplitude, as we want to find the yield stress for our data, which is a maxima. Figure 6.2 shows an example of Savitzky-Golay filtering. We see that the choice of window fits makes a good approximation of the global maxima, while for the local maxima below 1 GPa shear stress the window is too large.

6.5 Measuring Yield Stress From the Load Curve

We saw from the loading curves presented in Figure 2.5 that the stress can have multiple local maxima, this can pose a problem when trying to extract the yield stress. In some cases the system might see some local failures prior to ultimate failure, as we saw in section 2.2 upon discussing yield stress. Assuming there are several local maxima explaining both yield and ultimate stress, we employ a method for finding maxima which

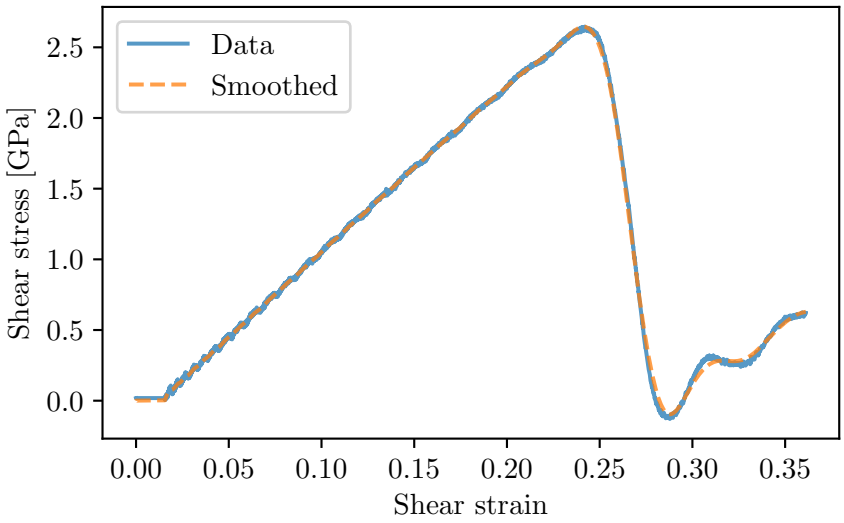


Figure 6.2: Example of Savitzky-Golay filtering of a loading curve. Window is 1001 data points, there are a total of 20000 datapoints. The polynomial used for fitting is of third order.

distinguishes between global and local maxima. The method must only choose significant maxima, ignoring local maxima that caused by small variations in the loading curve. The Python package Scipy's method find_peaks allows for tuning such that it only selects maxima in accordance with user input. First we locate the maximum value of the stress, the half maximum stress will serve as a minimum height of a peak. Further we require the peak to be separated by a distance of 500 data points from any other peak. Lastly we set a required prominence of the peaks to be 0.025. Peak prominence is the vertical distance between a peak and a minima in the interval between two peaks. Following is the function call

```
height = 0.5 * pxy[np.argmax(pxy)]
argmax = find_peaks(x=pxy,
                    distance=500,
                    height=height,
                    prominence=0.025)
```

which is used to find the index for the yield stress. The output log data from the MD simulations were smoothed using Savitzky-Golay filter. In some cases smoothing the data made it easier to locate maxima when the point of yield was not easily found due to the noise in the data.

6.6 Input and Target for Machine Learning

When fitting a model to data we need corresponding target values for each sample. As discussed in section 5 when introducing machine learning, the objective for our model is to predict the yield stress for a given structure. The cut space is a three dimensional slab which we treat as if it were two

dimensional when we are removing particles, which is represented as an image consisting of zeros and ones. The voids have the value 1, while the bulk is represented by 0. Pre-processing the data is often done in machine learning, typically one scales, normalizes or standardizes the data. If the difference in the order of the values in the input are large we run the risk of large values dominating, leading to poor results when training, as it may lead to some gradients being artificially large. By having an input where the values are of the same order of magnitude, or standardized, we ensure that the magnitude of the gradient is not biased. For our case, since the input image consists of zeros and ones, we do not perform any pre-processing. For the units we have chosen in LAMMPS (metal units) the pressure is given in bars, which means the stress is given in bars when we extract the data from LAMMPS log. The yield stress, being the target value, is scaled to GPa, such that the target value typically ranges from 0.5 to about 3, in units GPa.

Part II

Results, Implementation and Discussion

Chapter 7

Reproducing Parts of the Graphene Kirigami Study by Hanakata et al.

Following the study of Hanakata et al. [15] proved a good exercise before studying similar methods on SiO_2. We created 99 random samples which were realized, the term realizations is used to address predictions which have been simulated via MD, i.e. predictions that have been realized. We measured the yield tensile stress during the MD simulation. We used the same machine learning algorithm as presented by [15], a convolutional neural network with three convolutional layers with a 3×3 convolutional kernel, followed by ReLU activation and 2×2 max pooling. We achieved an R^2 score of 0.837 and RMSE 0.141 GPa[1]. We experienced a varying R^2 score, the cause of this could be the random choice of splitting the training and test sets each instance of training. If we were to do a proper analysis we would apply cross validation[2] to find a split of training and test set, such that the test set represents the variance in the data well. Figure 7.2 shows an example of five loading curves, and Figure 7.1 shows the geometry for the highest yield stress.

[1] We are using RMSE when presenting the loss of the machine learning algorithm, because it has physical interpretation, rather than MSE which has units GPa squared.

[2] Cross validation is a method for running multiple instances of the training with different to verify a results, an example is changing samples in the training and test sets.

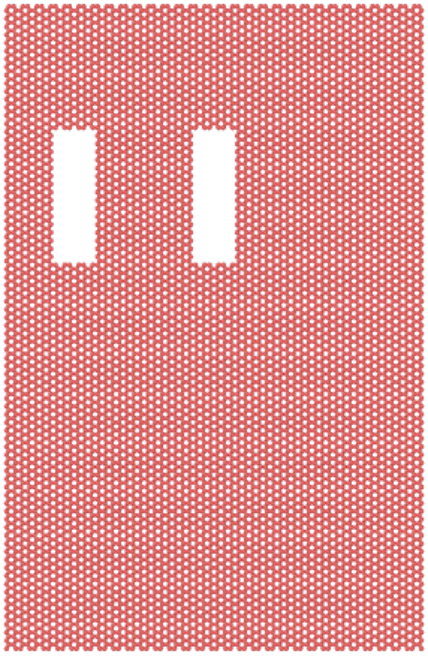


Figure 7.1: The strongest graphene geometry which was found by MD simulations of random geometries.

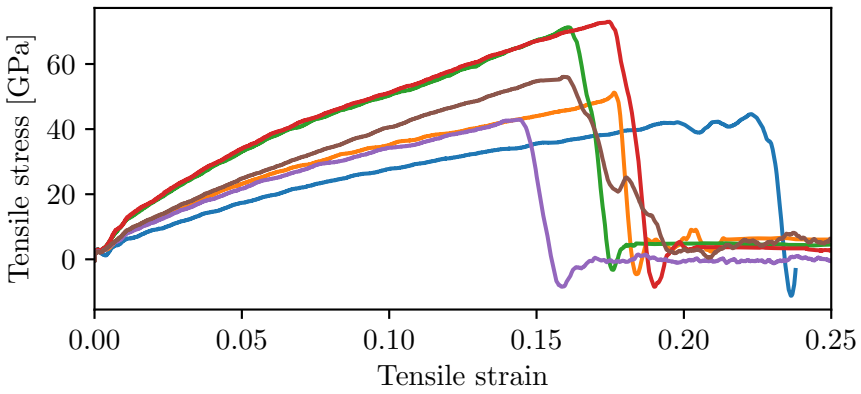


Figure 7.2: Loading curve for graphene subjected to tensile force.

Chapter 8

Building a Data set of Interface Geometries

We will go through the results in such a way that it represents the thought process of how we arrived at using different methods and certain results.

We restricted the porosity $\phi < 0.5$. If the porosity is larger than 0.5 there is more void than there is bulk material in the cut space. We expect that such systems are weak and that they are less sensitive to the structure of the geometry, and therefore of little interest.

For the earlier cases we used 3×3 convolutional kernel for all the convolutional layers, and therefore padding of 1, and no weight decay, as described in section where we presented the graphene results. After having fully expanded our data set we experimented with different CNN topologies and arrived at a different CNN topology[1], which improved the R^2 score. It will be clear at what time, and why, we changed the topology of the CNN.

8.1 Amorphous Silica

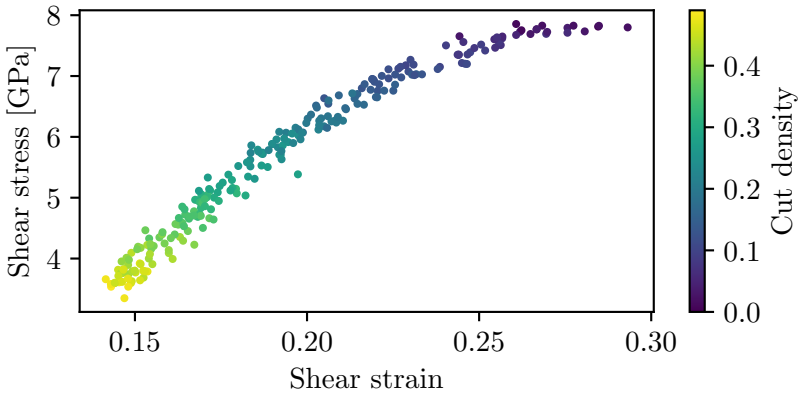
Initially we started out with an amorphous silica created by melting and quenching β-cristobalite, but it turned out that the strength of such a system was mainly decided by the porosity, and the layout of the voids was not as important. We generated a randomized and a manual data set, which consisted of 243 and 303 samples, respectively. Figure 8.1 shows the point of yield – the value for yield shear stress and the corresponding shear strain. We will refer to yield shear stress as yield. We can see the that there is a clear trend showing that the strength increases as the cut density becomes lower. Cut density is the density of cuts in the grid, a cut being a grid cell where particles have been removed. CNN performed well on both the data sets. For the random data set we got an R^2 score 0.979 and RMSE of 0.187 GPa, while for the manual case we achieved an R^2 of 0.940 and RMSE of 0.194 GPa. We attempted different geometries for the CNN by

[1] Topology is often used to describe the layout of the layers for a neural network.

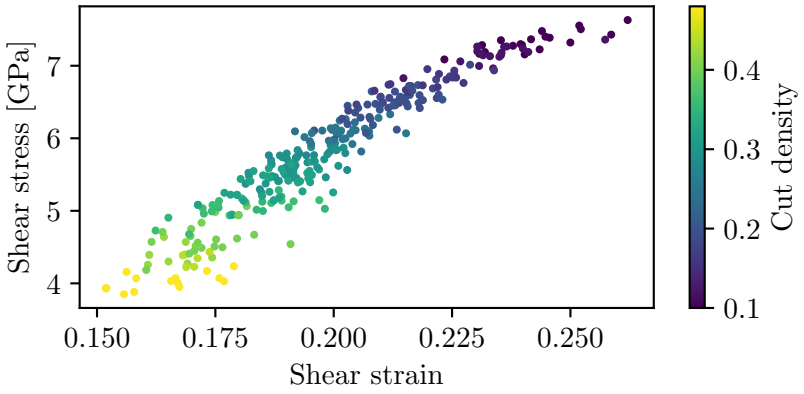
- increasing the convolutional kernel to 5×5
- increasing to 3×3 max pooling
- increasing and decreasing the number of hidden layers.

All of the above performed worse, with $R^2 < 0.85$.

To investigate if there were any effects of the data set being too small we studied the R^2 as we increased the number of samples. We added the manual and the random samples together and retrained the network. In Figure 8.2 we can see that there were no apparent effects of a small data set, due to the R^2 not increasing as we increased the number of samples.



(a) Random geometries.



(b) Manual geometries.

Figure 8.1: Point of yield for random and manual cut configurations for amorph silica. Cut density is an estimate of the porosity.

8.1.1 Polynomial Fitting

When we performed polynomial fitting with yield stress as the regressors and porosity as the predictor we got R^2 scores ≈ 0.98 for polynomials of

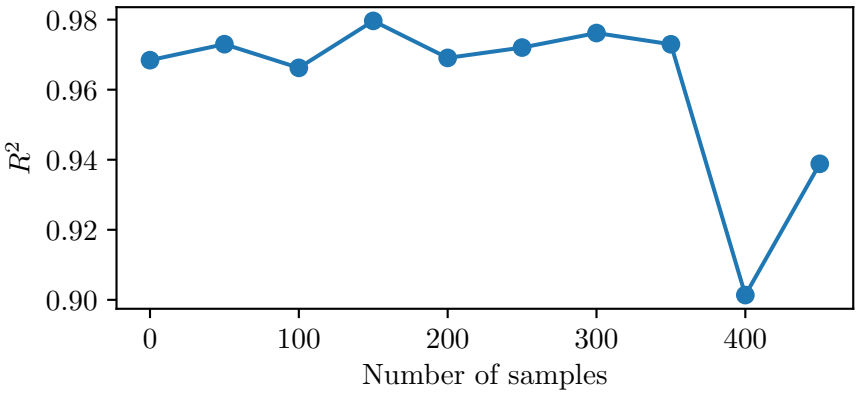


Figure 8.2: R^2 scores as we increase the number of samples. For the case of the amorphous system. Note that the sudden dip in R^2 score is most likely due to a unlucky random choice of samples for the test set.

degree $p \in [1, 4]$ for the manual data set. For the random data we achieved an R^2 score ≈ 0.99 for polynomials of degree $p \in [1, 4]$. Linear fitting of the data can be seen in Figure 8.3. Further we set up a one layer deep fully connected NN with with porosity as target. Since the input is a 10×10 image this means the network has 100 input nodes, and 1 output node for the predicted yield. This type of neural network is similar to polynomial fitting in the sense that there is a regression from a set of nodes to one output, but since the output is passed through an activation function we know that it is capable of capturing non-linear relationships. This method gave an $R^2 = 0.961$ for the manual geometries. We assume that the neural network picked up on some complex relationship in the images, which caused the R^2 score to become lower than for the case of polynomial fitting.

8.2 α-quartz

Since we can achieve better results by polynomial regression we did not pursue further studies with amorphous silica, but moved on to α-quartz. We started off with manual and random geometries for the 10×10 grid, and then moved on to Simplex noise using a 100×50 grid.

8.2.1 Manual and Random Geometries

The manual and random data set consisted of 263 and 100 samples, respectively. In Figure 8.5 we can see the point of yield for each data set. We observe that for the case of manual geometries the yield seems to be decided by something else than just the porosity. For the random case we are still seeing a clear trend for the porosity being the main factor of the strength. For the case of random noise we are essentially sampling

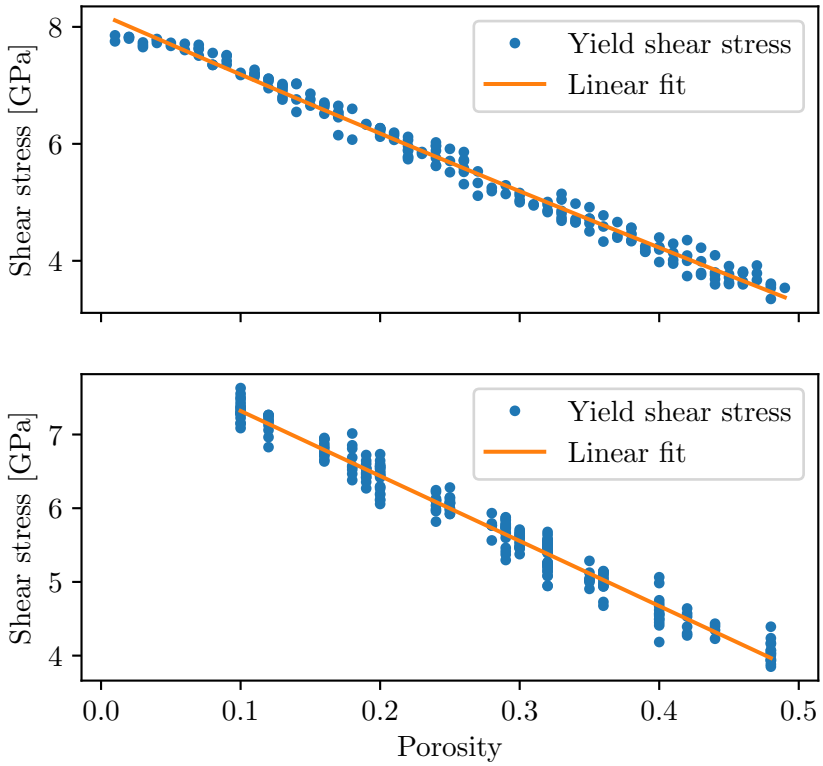


Figure 8.3: Linear fit of the random data (top figure) and manual data (bottom) for amorphous silica.

granular noise, so would expect the samples to behave similarly, as they are statistically similar.

For α-quartz, the CNN achieved an $R^2 = 0.921$ and RMSE $= 0.044$ GPa for the manual cuts, while for the polynomial regression $R^2 = 0.772$ for a third degree polynomial. Our space for possible randomized geometries is 2^{100}, and we know that most of these cases amount to white noise which most likely has similar mechanical properties. We could create a method for sampling the random space in a specific way, introducing some bias into the sampling. This would probably end up being a very time consuming project if we were to find a balance between bias and a sampling method that samples geometries showing more interesting properties. Creating manual geometries is also not a feasible way to create possibly thousands of samples.

8.2.2 Simplex Noise

Motivated by the results from the manual data set we moved on to Simplex noise to create new geometries. The grid was increased from 10×10 to 100×50. We created a data set with 1190 samples with the following

parameters

1. Scale $\in [10, 60]$[2]
2. Threshold $\in [0, 0.6)$ with steps of 0.01
3. Octaves $= 1$

We observed that for low porosities $\phi < 0.1$, there was little spread in the yield stress, as can be seen in Figure 8.7a. In the regime of low porosities the voids are small, and we do not expect much difference in the yield stress. The physics at larger porosities are more interesting, and we therefore decided to restrict the porosity to be $\phi > 0.085$, while still keeping it below 0.5. The reduced data set consisted of 1136 samples. When training the CNN we achieved $R^2 = 0.909$ and RMSE $= 0.154$ GPa, on the reduced data set we saw a no significant difference in the R^2 and MSE. The polynomial regression achieved $R^2 = 0.870$ for a third degree polynomial.

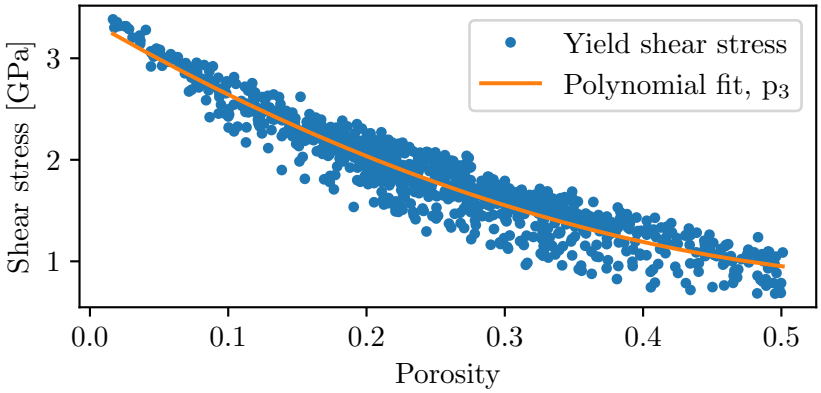


Figure 8.4: Third degree polynomial fit on the data set consisting of 1190 samples made by Simplex noise.

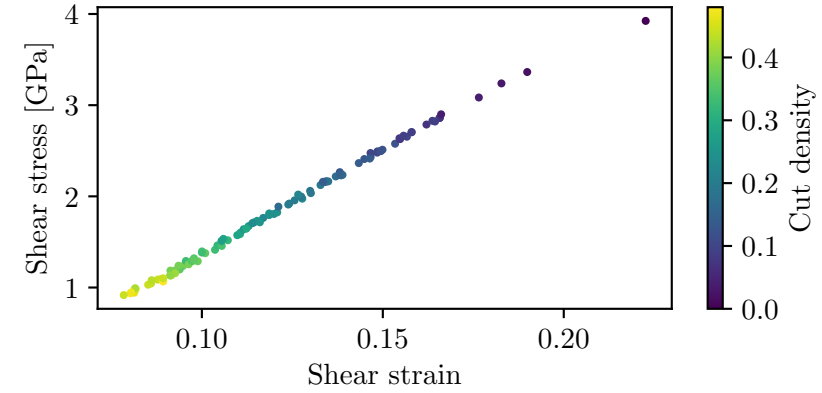
8.2.3 Extending the Data Set

To extend the data set we consider a method for finding geometries that are poorly explained by the model created by polynomial regression. We decided to train a CNN (hereby called poly-CNN for simplicity) where the inputs were the images as previously, but the target being the polynomial fit subtracted from the yield shear stress

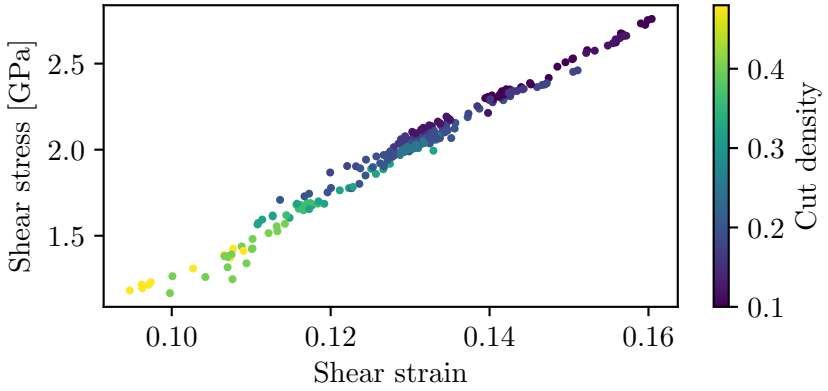
$$\tau_i^p = \tau_i^y - p_3(\phi_i)\,, \tag{8.1}$$

where i indicates a specific sample and p_3 is the third degree polynomial fit. Figure 8.6 shows the targets for the poly-CNN. We assume that the samples which have a target value that is furthest away from 0 are geometries that are poorly explained by porosity, as we know they are poorly explained by

[2]Scale $\in (50, 60]$ was only present in the earlier samples. We reduced it to maximum 50.



(a) Random geometries.



(b) Manual geometries.

Figure 8.5: Point of yield for random and manual cut configurations for α-quartz. Cut density is the estimated porosity based on the arrays, not the counted particles.

the polynomial fit. Further we assume that the geometries that are poorly explained by polynomial fit has some characteristics that are significant for the strength of the material. The goal is to have the CNN pick up on complex features which the model based on polynomial regression cannot explain. After having trained the poly-CNN we ran a series of unseen geometries through the model and let the model predict the yield shear stress τ_i^p. We then chose the samples with largest $|\tau_i^p|$ to expand our data set by realizing geometries that predictively deviate the most from the polynomial fit. When choosing geometries we chose specific intervals of the porosity, to make sure we had an even distribution of porosities. Had we simply extracted the geometries without showing care we would have chosen geometries with a high porosity, as there is more spread in this part of the model. The data set was extended to 1789 samples, samples are shown in Figure 8.7b, we can see that the spread of the data has increased compared to Figure 8.7a. When increasing the data from

1789 samples we noticed that our geometries had some similarities, the cause was found to be the lack of changing the base parameter in the Simplex noise. Since we had not shifted the coordinates by applying a base, the voids were placed at the initial coordinates. This means the data set was biased towards the placements of the voids, but the voids themselves were unique. We then further increased the data set to 3366 samples, shown in Figure 8.7c. For the last addition we introduced varying the octaves $= 1, 2, 3$, and varying base for each sample by a unique integer. The extension to the data set was mainly done by random choice, but it also contains 400 samples from experimentation of predicting largest yield stress at porosity $\phi = 0.3 \pm 0.0125$, these samples can be seen as a concentration of dots in Figure 8.7c. These 400 samples will be discussed in section 11.1. Admittedly these samples should have been left out from the total data set, as they introduce a bias towards porosity of ≈ 0.3. In the following studies we did not focus on porosity 0.3.

By introducing the poly-CNN as a method for choosing geometries, we have achieved a wider spread in the data by adding relatively few samples. If we had continued the random sampling of Simplex noise, we would most likely have had to perform a significant amount of more simulations to achieve the same variance in the data.

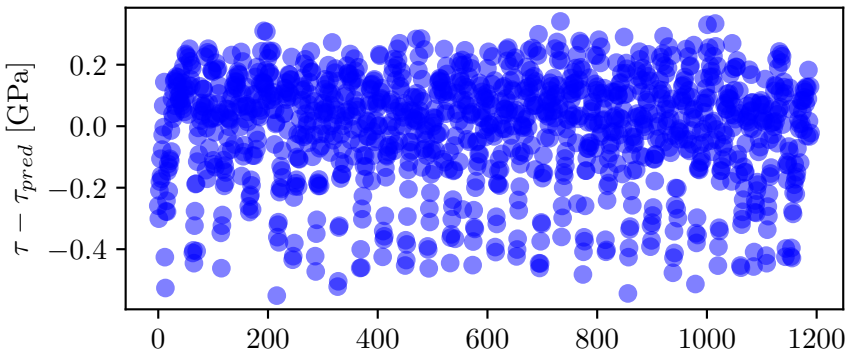
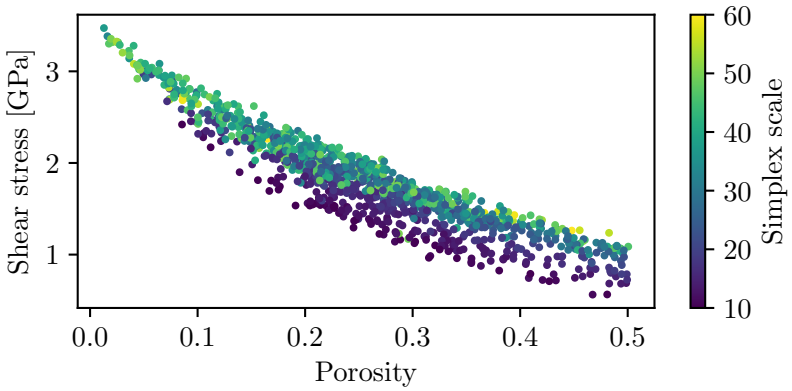


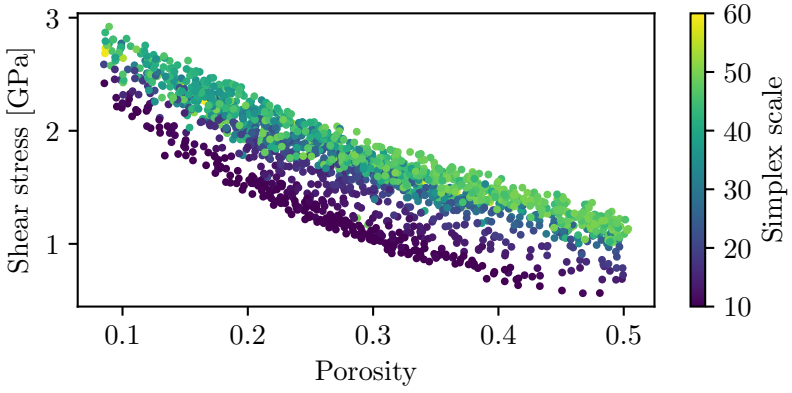
Figure 8.6: Target values for the poly-CNN, used to predict geometries that deviate from the polynomial fit. Predicted yield stress from a polynomial fitting of the yield shear stress vs porosity. The predictions are subtracted from the actual yield stress.

8.2.4 Summary

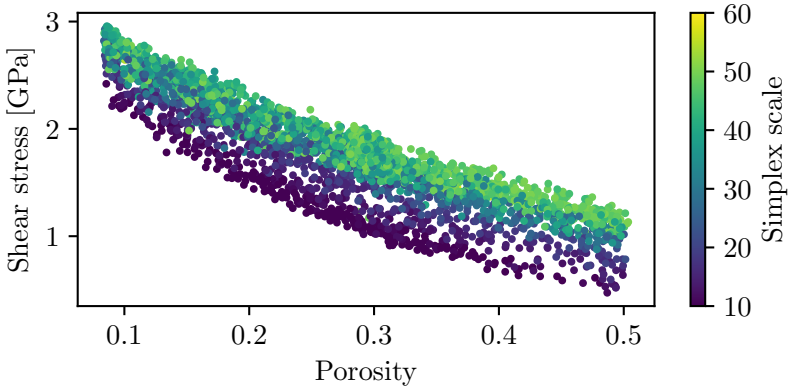
We found that CNN performed well for the manual geometries and Simplex noise for α-quartz. The polynomial fitting did not perform as well as it did for amorphous silica. By careful consideration when sampling we could expand our data set such that it contained samples with larger span in the yield for the same type of geometries.



(a) Base data set consisting of 1190 samples. Was reduced to 1138 samples upon removing samples where porosity $\phi < 0.085$.



(b) Data set expanded by poly-CNN method, consists of 1789 samples. Note the increase in variance in the data set compared to (a).



(c) Final data set consisting of 3366 samples. Expansion from (b) was done by random sampling of Simplex noise.

Figure 8.7: Figures show how the data set increased through the choices we made. Note that the increase from 1789 samples to 3366 did not increase the variance in the data as much as poly-CNN.

Chapter 9

Periodic Padding of the Model Input

The first attempt at padding the input was by periodic wrapping. The spatial size was increased by $\sim 1/8$th in each direction, yielding a 87×137 image, increased from 100×50. We observed a small increase in the R^2 score for both 3×3 and 9×9 kernel. For 3×3 the R^2 increases from 0.959 to 0.966, while for 9×9 R^2 increases from 0.975 to 0.986. A source of the increase in performance could be because we are introducing more nodes to the CNN, which allows for better fitting. An unfortunate effect of padding is that the porosity is not preserved.

The other attempt was to periodically pad by $(K-1)/2$, and set the first convolutional padding to be zero. This was done for the case of kernel size $K = 9$. The R^2 score was 0.956, which was a slight decrease from 0.975. This method would have been interesting to study further, but we arrived at the idea too late. This approach gives some periodicity to the CNN, it does not preserve the porosity either, but the porosity is closer to the actual porosity, as we are padding with fewer pixels. This padding meant the input size increased from 100×50 to 108×58. The results from the two first generations of the search algorithm proved promising, however they will not be presented in this part, but in the appendix in section D.

Chapter 10

Neural Network Topology Exploration

With a data set that shows a wide range of yield stress for a given porosity we investigated different topologies for the CNN that predicts the yield. We also studied what the networks were learning by using saliency maps. The saliency maps were also compared with stress fields from the MD simulations and particle displacements, to see if there were any correlations between what the network had learned and the physical quantities related to fracture that can measure.

When testing different methods we considered the R^2 score as a metric for evaluating the performance of the CNN, and we considered saliency maps to gain some insight into what the model picked up from the input. The R^2 score we had prior to testing was 0.964.

Specifying all the results we got from the various tests we did with neural networks will not be an enjoyable read, and it does not bring any value to the study. We therefore outline the main result, and present some of the results which outlines our thought process and helps us understand the connection to the physical problem, or to understand what the model learns as important features in the images.

We found no improvements when increasing or decreasing the number of layers for the neural network. We also found no improvements when increasing and reducing the channels for the convolutional layers. We found a slight increase in the R^2 score when adding a second fully connected layer, but as we argued in section 5.2 this breaks the spatial invariance of the method, which means the model is not fit for this problem. The final CNN has the same number of channels and convolutional layers as presented by Hanakata et al. [15].

When testing activation functions we found that stochastic gradient descent performed slightly worse compared to Adam. The optimal learning rate for Adam was found to be 0.0001.

10.1 Convolutional Kernel

Increasing the convolutional kernel proved to be the most important factor when connecting what the model characterizes as important features, and what we observe as important, for the yield of the model. We will return to what we observe as important features, and convolutional kernel size in section 17 when discussing explainability. Increasing the kernel means we have more weights to optimize, and the weights are optimized for a larger portion of the input image for each convolution. Figure 10.1 shows saliency maps for increasing kernel size. We see that as we increase the kernel, larger structures seem to show more importance for deciding the yield, as we can see by the intensity. We saw a small increase in the R^2 score when increasing the kernel to 9×9, and as we will see during the discussion of searching for new material structures in section 11, we saw that it performed better at predicting yield shear stress. When increasing the convolutional kernel to 11×11 we saw an almost doubling of computational time. Training the CNN took about 2.5 hours on an Nvidia P100 GPU, so we did not proceed with a larger kernel. The increase in R^2 was about 0.012 when using an $11 \times 11 kernel$, compared to 9×9, and we saw no real improvement in the saliency maps.

10.2 Regularization Techniques

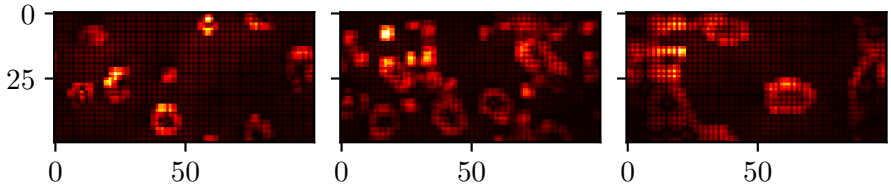
During our testing of regularization techniques we found that adding dropout and weight decay showed some improvements. Early stopping typically terminated the training after less than 100 epochs, giving an $R^2 = 0.826$, well below 0.964.

Batch Normalization

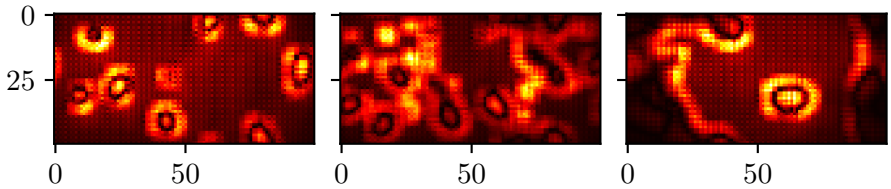
Batch normalization reduced the performance of the model, with an $R^2 = 0.937$, which is not a big drop from 0.964 when comparing the numbers, but the distance to R^2 has almost doubled. Saliency map for a model containing batch normalization is shown in Figure 10.2, we see that it little similarity to the base geometry. This model seems to have picked up on some complex relationship undetectable by the human eye, and we therefore did not continue with this regularization.

Weight Decay

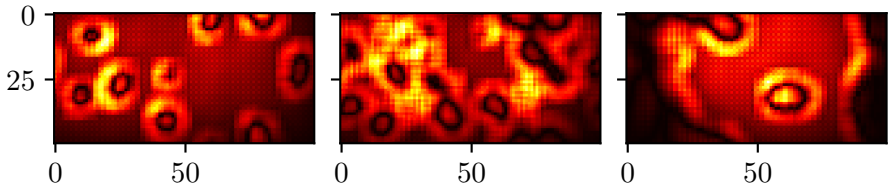
Adding weight decay showed some improvements for $0.0001 < \alpha \leq 0.01$. Small weight decays increases computational time, as the model takes more time to converge. $\alpha = 0.01$ seemed to be a good trade-off, we gained the extra performance with an acceptable performance loss from $R^2 = 0.0.964 \rightarrow 0.963$ for $\alpha = 0.001$ and $\alpha = 0.01$, respectively. For $\alpha = 0.1$ we saw $R^2 = 0.946$.



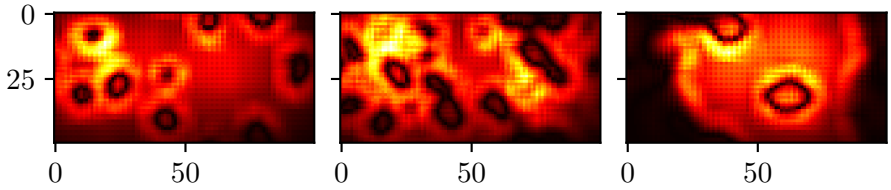
(a) 3×3 kernel.



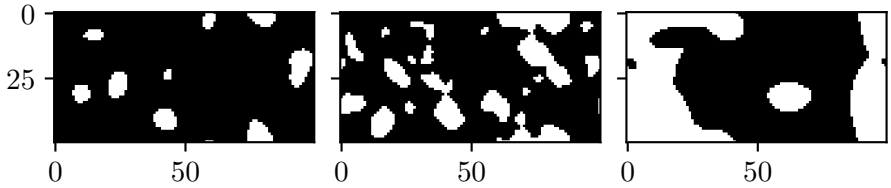
(b) 5×5 kernel.



(c) 7×7 kernel.



(d) 9×9 kernel.



(e) Geometry for comparison.

Figure 10.1: Comparison of saliency maps with varying convolutional kernel. Geometries are made with Simplex noise, for α-quartz system.

Dropout

We achieved best performance from using 0.5 as dropout rate. For the cases where the data set was small, the difference in R^2 for the dropout rates were larger than for our final data set with 3366 samples. For the small data sets we achieved best results for dropout rate smaller than 0.5. Dropout is often used by applying dropout after each convolutional layer. When we add

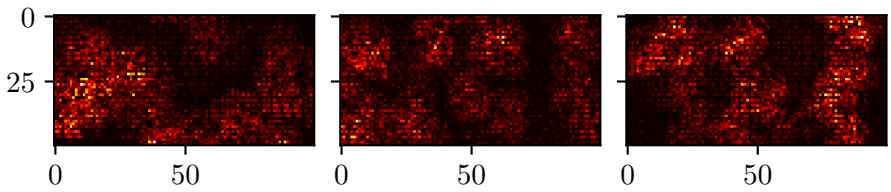


Figure 10.2: Saliency map for batch normalization test. Geometry is the same as for Figure 10.1.

dropout after each convolutional layer the saliency maps looked somewhat promising, outlining the shapes of the voids, but the R^2 score dropped to 0.899, an example of the saliency map is shown in Figure 10.3. We see that there are some brighter areas, the most prominent being a small spot at the left most figure at $\approx [47, 25]$. We observed more than once that there were large values for small voids, we assume that this is due to small voids being present in systems with low porosity, which are typically strong systems. E.g. for geometries with high scale and high threshold we will typically see small circular voids, and these geometries often have small porosity and large yield.

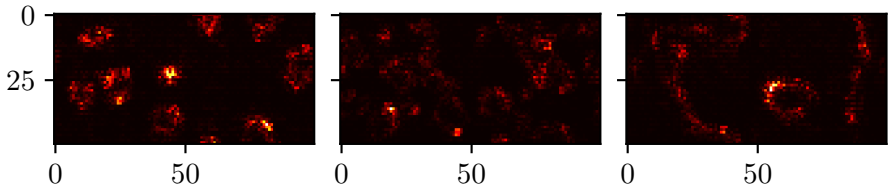


Figure 10.3: Saliency map for the case of adding dropout after each max pooling. Geometry is the same as for Figure 10.1.

10.3 Summary

We saw that increasing the convolutional kernel to 9×9, adding weight decay with decay strength $\alpha = 0.01$ and adding a dropout layer with 0.5 dropout rate before the fully connected layer gave a performance boost to our CNN.

Chapter 11

Searching for New Material Structures

In this chapter we will present methods for searching for new geometries which has specific mechanical properties. We will use the search algorithm to find new geometries. The approach is as follows

1. Train a CNN on the base data set (at first consisting of 3366 Simplex geometries)
2. Used the trained model to perform predictions on a set of unseen geometries
3. Choose a subset of the unseen geometries which matches the target mechanical properties
4. Simulate the subset and add the results to the base data set
5. repeat steps 1 to 4.

We will present variations of searching for new material structures. We include the variations to show the robustness of the method, and some variations which were not as successful, to show the thought process through the study. We performed two variations where we targeted samples with a specific porosity and extracted the geometries with the highest yield. We also targeted geometries with a specific predicted yield.

We will refer to the method of searching through geometries, choosing new geometries and performing MD simulations for the chosen geometries as the search algorithm. The samples made by the the search algorithm through the generations will be referred to as made through generative design.

11.1 Target Porosity $\phi = 0.3 \pm 0.0125$

Our first attempt at a generative design was for a fixed porosity. We train the model on the base data set, we then extract 100 predictions that match the target porosity with the highest predicted yield. We performed four

iterations of the search algorithm. Figure 11.1 shows how the relative area of the largest void changes as the strength of the material increases, the data are all geometries within the target porosity. We see that stronger geometries have fewer and larger voids, while the weaker geometries has many small voids. We observe that the relative area in some cases are close to the target porosity. This indicates that there is either one single void, or a large void in combination with one or more smaller ones. In Figure 11.2 we can see samples from the 4th generation. We see that the geometries are similar and that the voids are large, in some cases there is only one void. We concluded that the choice for target porosity is too large. In retrospect it might be interesting to study this further, due to the fact that the strength might be dominated by one void, considering that the overall strength of a system is in many cases decided by the largest void, as we discussed during Griffith's theory. Later in the study we return to target porosity, but with a value of $\phi = 0.15 \pm 0.01$.

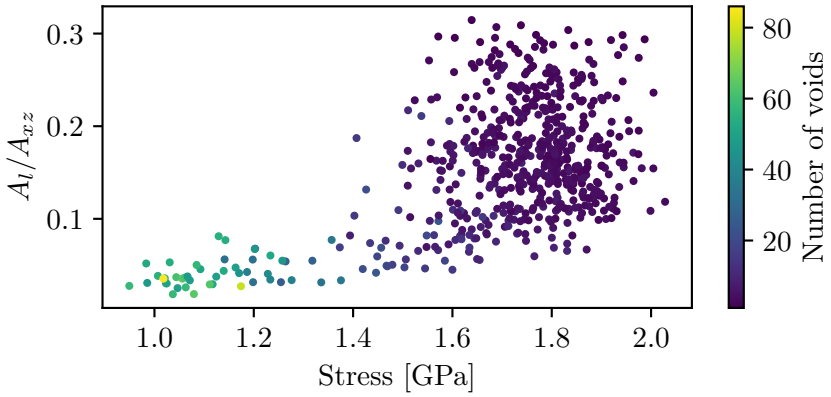
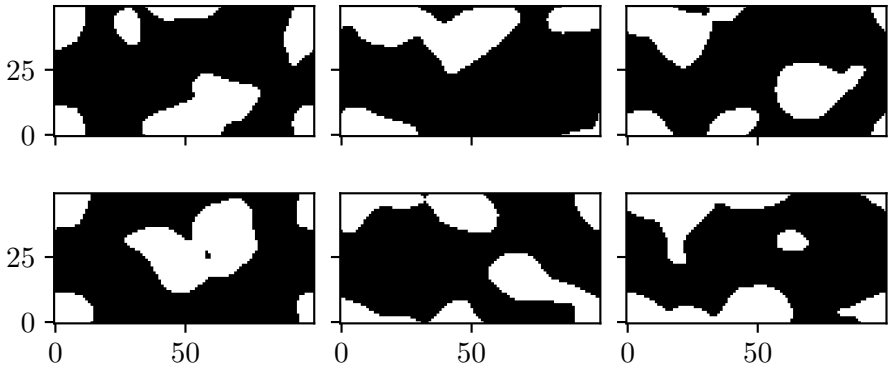


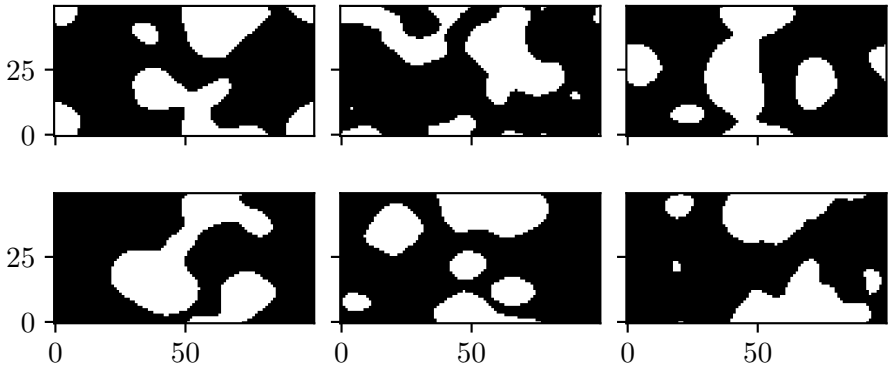
Figure 11.1: Relative area of the largest void as the true yield shear stress increases. A_l is the area of the largest void, while A_{xz} is the cross-sectional area of the cut space.

11.2 Target Yield Shear Stress

For our final data set and CNN consisting of 3366 samples we applied the search algorithm to see if our model could predict a geometry with a given strength. We chose $\tau^y = 2.1 \pm 0.001$ GPa for our target yield shear stress. When we have performed predictions on unseen geometries, we then randomly choose 100 samples out of all the geometries which were predicted to have the target yield. Looking at the data set in Figure 8.7c we see that samples with a true strength of 2.1 GPa shows a range of porosities $\approx [0.12, 0.25]$, which means we will have a broad range of possible geometries and types of voids. The search space was decided to consist of of 1476000 Simplex geometries for each generation. When extracting the geometries which has been predicted to have desired yield we end up with more than 100 geometries. We typically ended up with



(a) Strongest samples with true $\bar{\tau}^y = 1.96$.



(b) Weakest samples with true $\bar{\tau}^y = 1.63$.

Figure 11.2: Comparison of the six strongest and weakest geometries for the realizations of the 4th generation at target porosity $\phi = 0.3 \pm 0.0125$. Note that the top middle image in (a) consists of one void split across the boundaries.

1000 geometries which were predicted to have the desired yield. To narrow the samples down to 100 we performed a non-uniform random choice by the following Python code:

```
# inds are the indices for the samples with desired yield
n = inds.shape[0]
probabilities = np.ones(n, dtype=np.float16) / n
# The probabilities above are assigned to each entry in inds,
# which ensured non-uniform sampling
inds_new = np.random.choice(inds, size=100, p=p, replace=False)
```

The 100 samples were then realized by MD simulations.

For the 1st generation we found that the 100 realizations had $\bar{\tau}^y = 2.104$ with a standard deviation $std = 0.103$ GPa, which is very close to the predicted yield. The distribution of true and predicted yield can be seen in Figure 11.3a. We then applied the search algorithm; adding the realizations and retraining the model. The results from the 5th, and final, generation is shown in Figure 11.3b. We see that the average yield was slightly further away from the average for the 5th generation, while the standard deviation decreased to $std = 0.086$. Generation 2-4 is shown in the appendix section

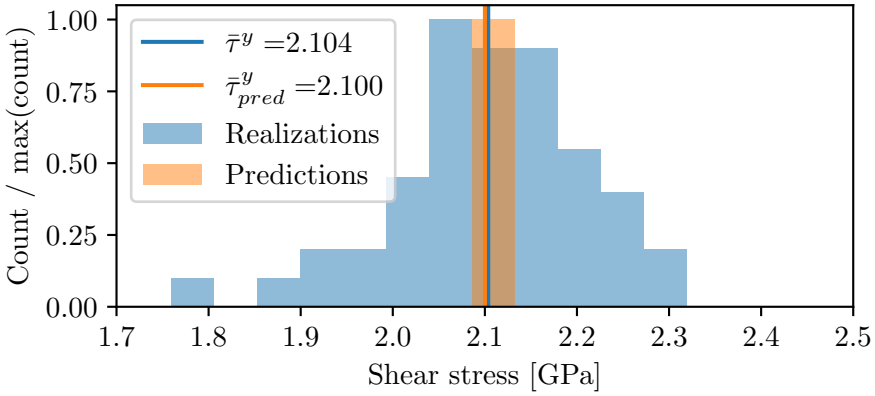
C.1. To summarize, we found that through generations 2-5 the average yield shear stress deviated more from the target yield than the it did for the 1st generation. The standard deviation became slightly smaller for the 5th generation, meaning the spread about the target yield became smaller. In Figure 11.4 we can see a heatmap showing the intensity of the voids for the 1st and 5th generation. The intensity explains how many samples among the 100 has a void for a specific grid cell. If the network has learned that a certain type of geometry is stronger, considering the distribution or shape of the voids, we would expect to observe this as regions with high intensity. For both cases we see that the maximum intensity is somewhere in range of 33 – 35, meaning a third of the samples show voids for the same grid cells. We see that there are more local high intensity areas for the 5th generation, but we see no patterns forming. If we were to continue the search algorithm we might have ended up with more clearly defined local intensities, indicating that the model has settled on some geometry which it has learned to have a higher yield.

For the 1st and 5th generation we also tried sampling through twice as many unseen geometries to see if this would improve the performance. If the performance increases, it would imply that the accuracy being limited by the sampling. We also narrowed the the target yield such that there were exactly 100 predictions to choose from, giving a target $\tau^y = 2.1 \pm 2.5 \cdot 10^{-5}$ and $\tau^y = 2.1 \pm 1.5 \cdot 10^{-4}$, for the 1st and 5th generation respectively. We achieved $\bar{\tau}^y = 2.078$ GPa and $\bar{\tau}^y = 2.056$ GPa, respectively. The distributions are found in the appendix. The average yield and the standard deviation became smaller in both cases. The results are summarized in Table 11.1.

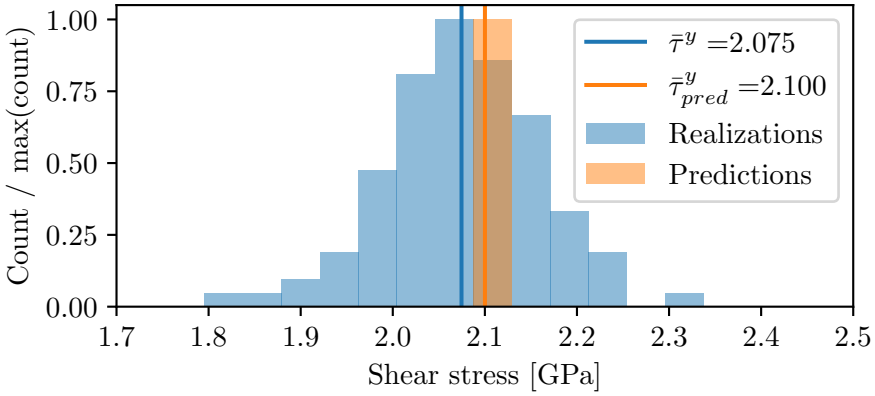
Table 11.1: Results from search algorithm for target $\tau^y = 2.1 \pm 0.001$. Results contain average true yield stress, as well as standard deviation of measurements done by MD simulations. The cases of narrowing the target yield due to increased search space are noted.

Generation	$\bar{\tau}^y$ [GPa]	std [GPa]	Note
1	2.104	0.104	
1	2.078	0.082	Long search, $\tau_t^y = 2.1 \pm 2.5 \cdot 10^{-5}$
2	2.085	0.091	
3	2.044	0.085	
4	2.057	0.093	
5	2.075	0.086	
5	2.056	0.085	Long search, $\tau_t^y = 2.1 \pm 1.5 \cdot 10^{-4}$

We performed one generative design for different target yields to test if the model performs well on other targets. The target predictions were $\tau^y = 1.0 \pm 0.001$ GPa, $\tau^y = 1.5 \pm 0.001$ GPa and $\tau^y = 2.5 \pm 0.001$ GPa. These are shown in Figure 11.5. We see that the average yield is close to the target for the lower targets, while for the target $\tau^y = 2.5 \pm 0.001$ GPa has a true yield of almost 2.6 GPa. There are few samples at $\tau^y > 2.5$, so we



(a) 1st generation. Standard deviation of 0.103 GPa of measurements done by MD simulations.



(b) 5th generation. Standard deviation of 0.086 GPa of measurements done by MD simulations.

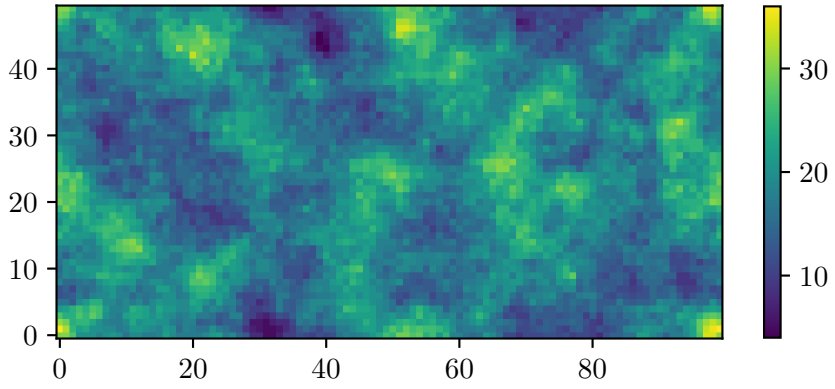
Figure 11.3: Distributions of yield shear stress for the predictions and realization for the search algorithm with target $\tau^y = 2.1 \pm 0.001$ GPa of measurements done by MD simulations.

would expect that the performance suffers in this case, as it is not familiar with the geometries at this yield.

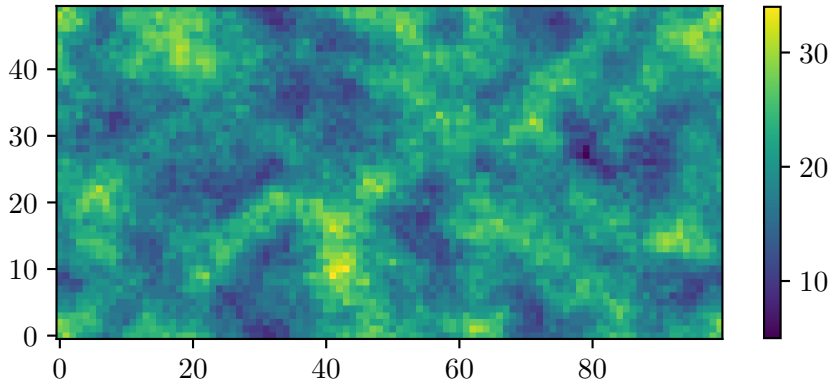
We saw that the standard deviation became smaller during this generative design. The network iteratively learned how to predict geometries for a specific yield.

11.2.1 Increasing the Convolutional Kernel

When increasing the convolutional kernel to 9×9 we saw an decrease in the standard deviation. The 1st generation had average yield stress $\bar{\tau}^y = 2.091$ GPa and standard deviation $std = 0.059$ GPa for target $\tau^y = 2.1 \pm 0.001$ GPa. We saw no improvement in average yield and standard deviation through the 2nd or 3rd generation, as can be seen in Table 11.2.



(a) 1st generation.



(b) 5th generation.

Figure 11.4: Intensity of voids for all 100 geometries for both 1st and 5th generation. From the search algorithm targeting $\tau^y = 2.1 \pm 0.001$ GPa.

11.3 Target Porosity – Strongest Predicted Geometries

In the previous section we showed that we can generate a geometry for a specific yield strength. The next step was to see if we could generate geometries with a specific porosity, and see if the network accurately predicts the yield. The 100 predicted samples we chose to realize were the ones that had the highest predicted yield. Effectively we are then seeing if our model can predict geometries with a given porosity and strength. For this generative design we also explore the possibility of finding geometries that show significantly higher yield by constantly extracting the predictively strongest geometries. For each generation we sampled 2952000 unseen Simplex geometries.

For this part of the study we had moved to a 9×9 convolutional kernel, as we saw an increase in the R^2 score, as discussed in the previous section. When discussing explainability we will also see why we chose 9×9 kernel. The network is the same as will be presented in section 18.2, when we introduce the Python code for the CNN.

Table 11.2: Results from search algorithm for target $\tau^y = 2.1 \pm 0.001$ using a 9×9 convolutional kernel. Results for average true yield stress, as well as standard deviation.

Generation	$\bar{\tau}^y$ [GPa]	std [GPa]
1	2.091	0.059
2	2.116	0.069
3	2.083	0.062

The 1st generation is shown in Figure 11.7a. We see the average true yield is close to the average predicted yield. The standard deviation is 0.038 GPa, which is much lower than what we saw in the last section when we targeted a specific yield. We also performed a search over four times as many geometries for the 1st and 6th generation, i.e. 12 million Simplex geometries. Again we saw no significant improvements from expanding our search space. 10th generation is shown in Figure 11.7b. The continuation of the search algorithm, which will be discussed below, was based on the first 10 generations. We also performed two more generations with this approach, to compare with the approach that will be discussed below. The results are summarized in Table 11.3. We observed that the distribution of porosities for the chosen geometries were leaning towards the lower end of $\phi = 0.15 \pm 0.01$, as we can see in Figure 11.8. This is as we would expect, as porous materials typically become weaker. We saw no notable increase of the average yield during the 10 generations.

Full-Search Approach

When we had trained 10 generations we decided to modify our approach of the search algorithm, more in line with what Hanakata et al. [15] did. The main part to take from their approach is that they retrained the network on all the samples each generation. The design space Hanakata et al. [15] had to work with consisted of 2^{15} possible configurations, which made it possible to perform MD simulations on all configurations.

Generation 11 and 12 discussed in the section above were created simultaneously with the generations for this approach. The 11th generation also serves as a comparison for the 1st generation of this alternate approach. During the 10 generations discussed above we have exposed the model to 30 million geometries in total, let us assume that within these samples there exists some geometries which has higher yield, let us call it an "optimal geometry". It is possible that our model has skipped an optimal geometry during the generative design. During training of this approach we include the 1000 samples which were realized through the ten generative design iterations presented above. We would expect that generative design to behave differently as the model search a known space of geometries, instead of being exposed to new geometries. The prediction of 30 million geometries took about 17 hours on an Intel Xeon E5-2670 CPU with 16 physical cores. Then the new approach is:

Generation	$\bar{\tau}^y$ [GPa]	$\bar{\tau}^y_{pred}$ [GPa]	std [GPa]	Note
1	2.574	2.567	0.038	
1	2.598	2.583	0.042	Long search, 12 mill.
2	2.579	2.549	0.047	
3	2.580	2.605	0.043	
4	2.588	2.548	0.043	
5	2.590	2.561	0.041	
6	2.582	2.573	0.042	
6	2.598	2.592	0.046	Long search, 12 mill.
7	2.583	2.566	0.048	
8	2.582	2.580	0.046	
9	2.589	2.582	0.043	
10	2.591	2.568	0.043	
11	2.593	2.510	0.045	
11	2.596	2.536	0.034	Long search, 24 mill.
12	2.583	2.601	0.040	

Table 11.3: Results from the search algorithm for target porosity. Results for average predicted and true yield stress, as well as standard deviation. Includes intermediary attempts at searching a wider space of unseen geometries, these are noted by "long search" and how many samples is included in the space.

1. Retrain the CNN on the 30 million geometries
2. Choose the 100 strongest within the target porosity $\phi = 0.15 \pm 0.01$
3. Add to the existing data set
4. Repeat step 1-3.

The 1st generation is shown in Figure 11.10a. We see that the network predicted the geometries to be weaker than they actually were. Due to lack of time we performed the generative design for 5 steps. We saw a small increase in the average true yield, which can be seen in Figure 11.12. The standard deviation showed no improvement. The 5th generation is shown in Figure 11.10b. Comparing the distributions for the 1st and 5th generation we see that the model performs better for the latter. In fact, for the second generation we observed that the predicted average yield was $\bar{\tau}^y_{pred} = 2.631$, while the realized value $\bar{\tau}^y = 2.610$, so the model only needed the 100 samples from the first generation to correct itself. Comparing the 1st and 2nd generation to the 11th and 12th from the previous approach we see that the new approach yield better results. In Figure 11.9 we can see a larger increase in the R^2 score from the 1st to the 2nd generation. Table 11.4 summarizes the results. Figure 11.11 shows the void intensity for the 100 strongest and 100 weakest out of the realizations up until 5th gen, which in total is 1500 realizations. We see that there is some pattern in the placement

of the voids for both the weakest and the strongest, indicating that the network has learned a way to place the voids to optimize the strength of the system. Figure 11.13a shows how the R^2 score and MSE evolved during training, we achieved $R^2 = 0.9913$ and RMSE $= 0.052$ GPa. We see that the model quickly stabilizes at a high R^2 and low MSE.

Figure 11.15 shows the six strongest and six weakest geometries among the 1500 designed geometries. We see that the six strongest shows a set of voids with smoother periphery, and they tend to line up along the shearing direction – positive x-direction (right in the plane of this paper). The weakest geometries shows more variation in the shape of the voids, but they also show that they have a tendency to line up along the shearing direction.

We then let the model predict the yield for 74 million unseen geometries. Figure 11.14 shows that it performed well, but unfortunately the increase in average yield is not larger than what we saw during generations 1 through 5.

We decided to make an attempt at doubling the data set by flipping the geometries about the axis which we applied the shearing. Figure 11.16 shows an example of the flipped geometries. Due to periodic boundary conditions the flipped geometries should have the same yield if we perform MD simulations. We saw no change in the R^2 or MSE during training. We found no improvement for the distributions of yield shear stress with $\bar{\tau}^y_{pred} = 2.603$ GPa and $\bar{\tau}^y = 2.610$ with standard deviation 0.042 GPa. The averages are still very similar. We might expect that the standard deviation would decrease.

In section 15 we will discuss shuffling the voids of the geometry which was found to have the highest yield among the 10 generations from the previous generative design approach (not including the geometries found by increasing the search space) and the five generations presented here.

Table 11.4: Results from the alternative search algorithm for target yield shear stress. Results for average predicted and true yield stress, as well as standard deviation.

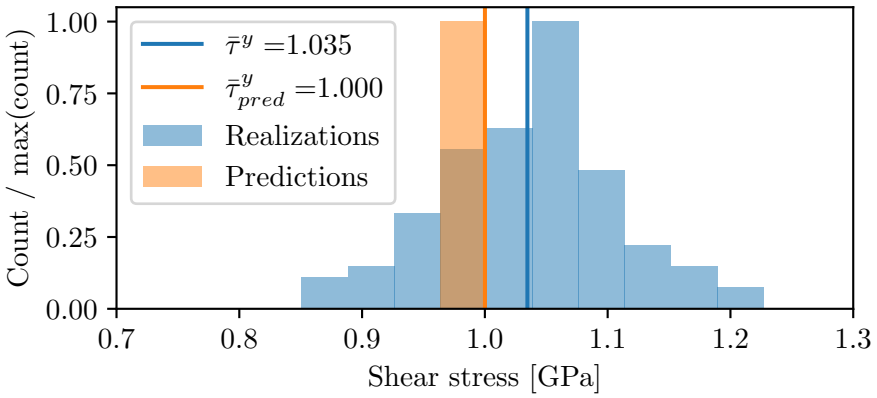
Generation	$\bar{\tau}^y$ [GPa]	$\bar{\tau}^y_{pred}$ [GPa]	std [GPa]
1	2.604	2.535	0.037
2	2.610	2.631	0.041
3	2.614	2.611	0.040
4	2.619	2.606	0.037
5	2.617	2.613	0.040

11.4 Summary

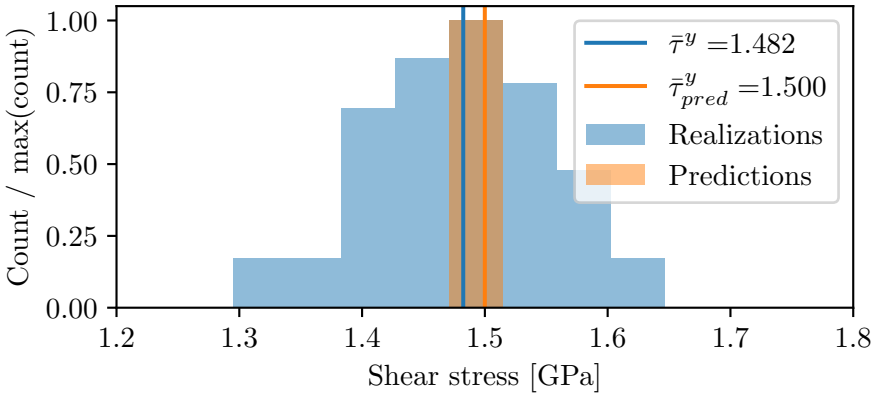
When experimenting with target porosity $\phi = 0.3 \pm 0.0125$ we learned that the voids were often large, in some cases there were one void present. We decided that the search for geometries that showed interesting mechanical

properties could be difficult for less complex geometries. We decided to move to targeting a specific yield to see if we could predict the strength of material structures accurately. We found that we could predict different target yield stresses. When we performed a search algorithm we did not see any improvement in the distribution of the realized yield. We found that training the model with a 9×9 convolutional kernel produced distributions of realized yield that had smaller standard deviation than for 3×3 kernels. When returning to target porosity $\phi = 0.15 \pm 0.01$ we found that we could accurately predict the strength of material structures when putting a constraint on the space of geometries. We saw no improvement in the average yield of the realized generations across 10 generations. When changed our sampling approach to be limited to the space containing all the geometries from the 10 generations we saw that the average yield slowly increased. The CNN performed well with an R^2 score of 0.991.

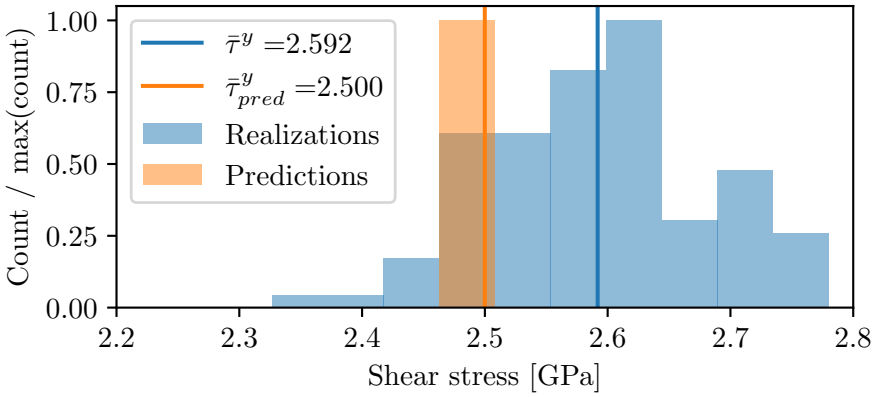
We found no improvement in distribution of the realized yield when increasing the space of unseen geometries.



(a) Target $\tau^y = 1.0 \pm 0.001$ GPa. Standard deviation 0.071 GPa of measurements done by MD simulations.

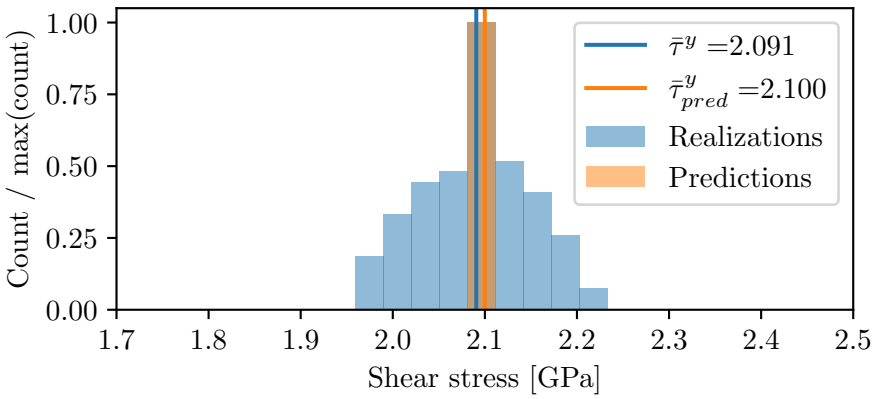


(b) Target $\tau^y = 1.5 \pm 0.001$ GPa. Standard deviation 0.071 GPa of measurements done by MD simulations.

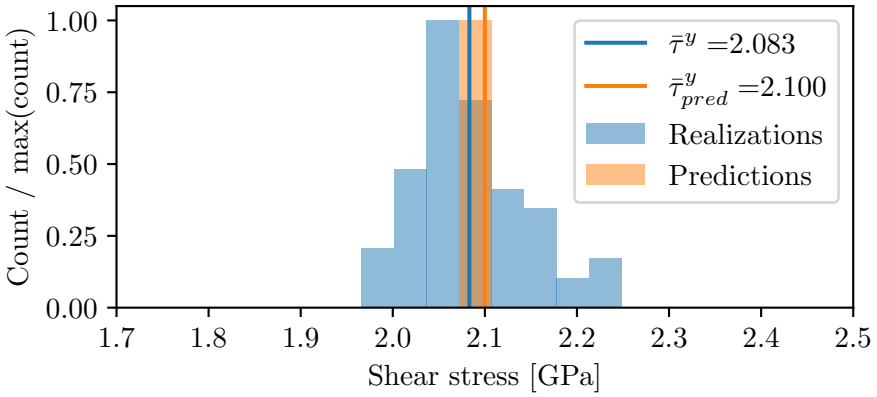


(c) Target $\tau^y = 2.5 \pm 0.001$ GPa. Standard deviation 0.089 GPa of measurements done by MD simulations.

Figure 11.5: Results from various target yield stresses for the generative design. We performed only one iteration of the generative design.

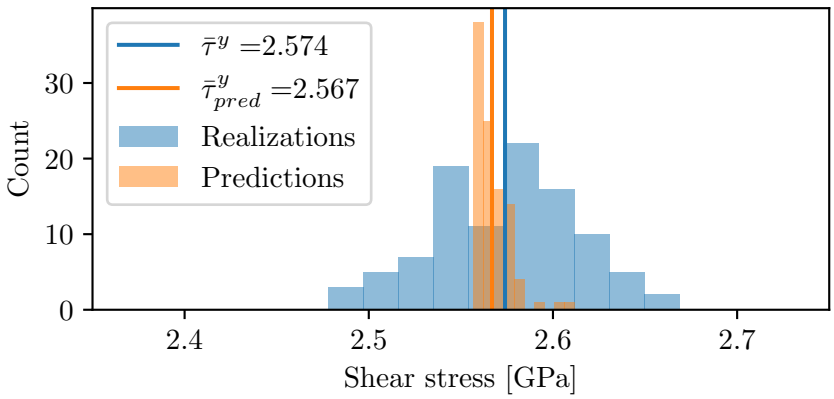


(a) 1st generation. Standard deviation 0.059 GPa of measurements done by MD simulations.

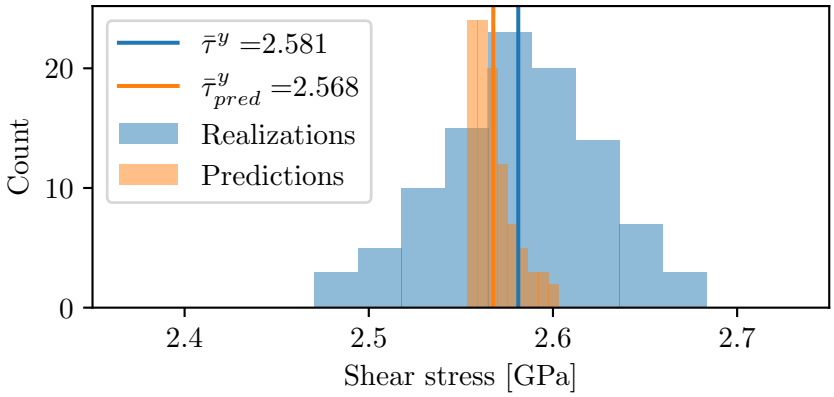


(b) 3rd generation. Standard deviation 0.062 GPa of measurements done by MD simulations.

Figure 11.6: 1st and 3rd generation of the search algorithm for target $\tau^y = 2.1 \pm 0.001$. Model was trained with a 9×9 convolutional kernels.



(a) 1st generation. Standard deviation 0.038 GPa of measurements done by MD simulations.



(b) 10th generation. Standard deviation 0.040 GPa of measurements done by MD simulations.

Figure 11.7: Distributions of true and predicted yield shear stress for the 1st and 10th generation for target porosity $\phi = 0.15 \pm 0.01$.

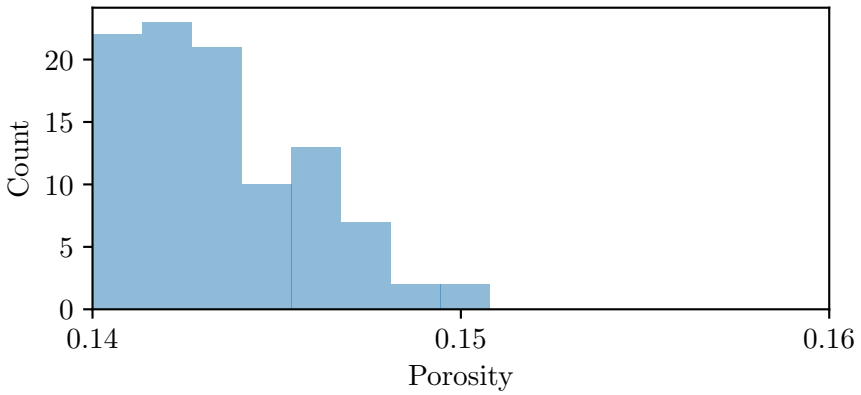


Figure 11.8: Distribution of porosities for 1st generation with targeted porosity $\phi = 0.15 \pm 0.01$.

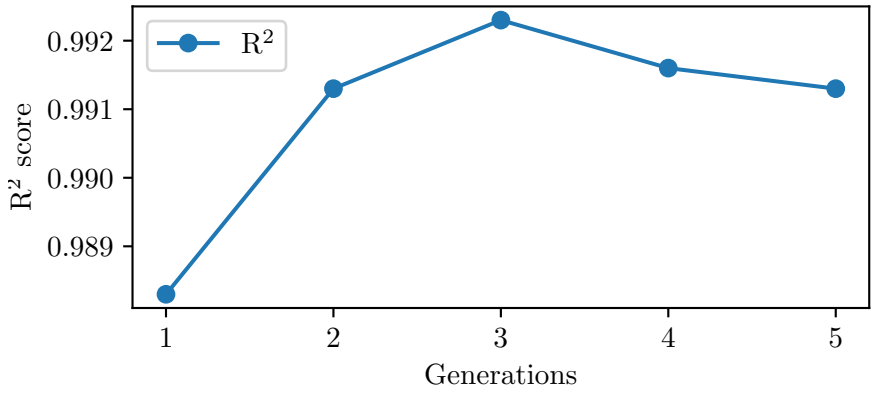
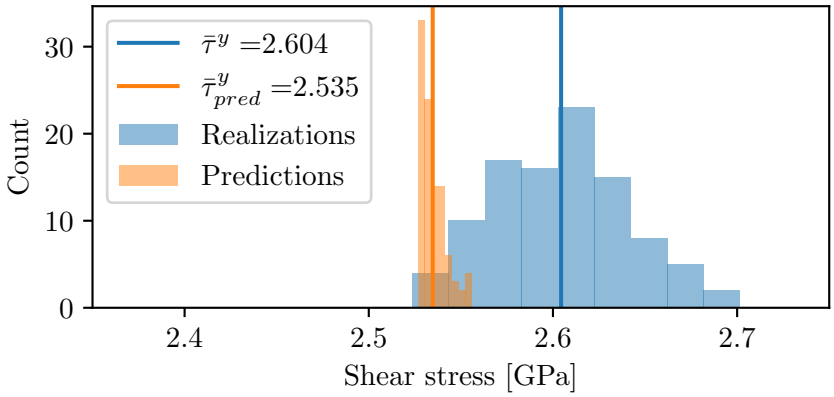
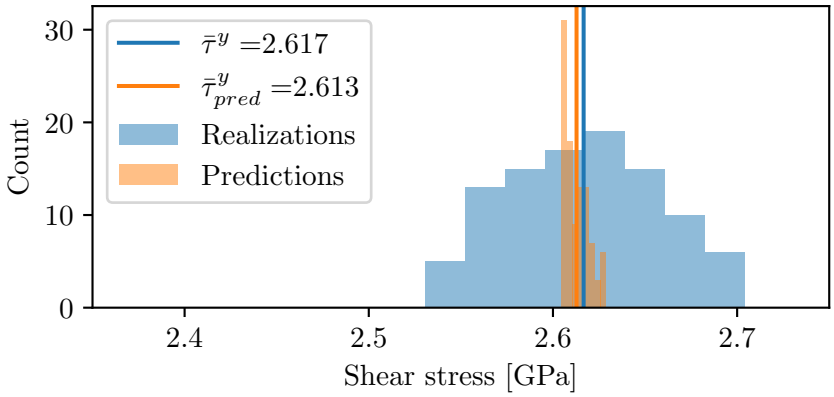


Figure 11.9: R^2 score as we increase generations of the full-search approach of the search algorithm.

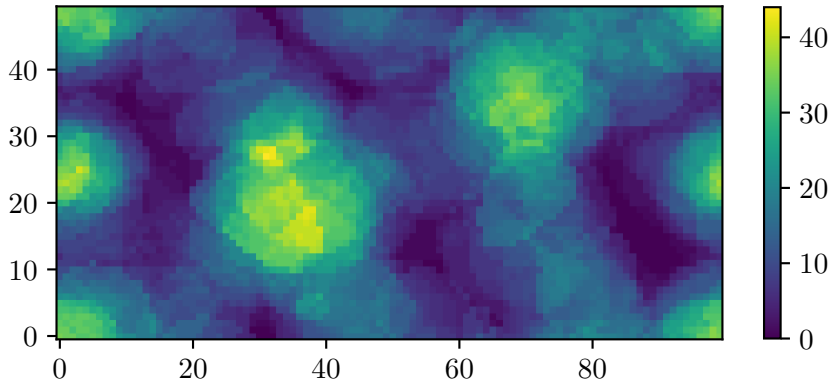


(a) 1st generation. Standard deviation 0.038 GPa of measurements done by MD simulations.

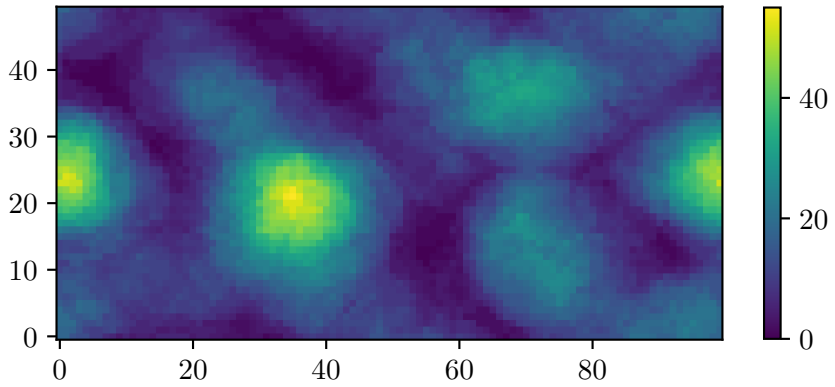


(b) Full distribution of predictions.

Figure 11.10: 1st and 5th generation for target porosity $\phi = 0.15 \pm 0.01$ for the full-search approach to the search algorithm.



(a) 100 strongest. $\bar{\tau}^y = 2.678$ GPa of measurements done by MD simulations.



(b) 100 weakest. $\bar{\tau}^y = 2.502$ GPa of measurements done by MD simulations.

Figure 11.11: Figures shows the intensity of voids. Geometries are chosen 1500 samples from the combined data set of the ten generations from the search algorithm, and the five generations from the full-search approach to the search algorithm.

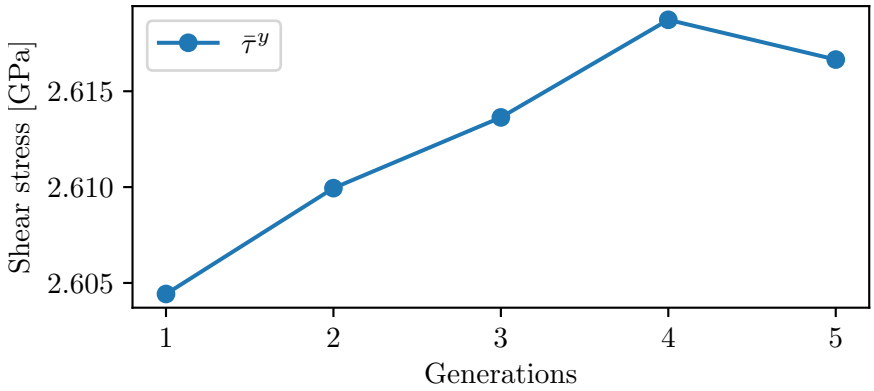
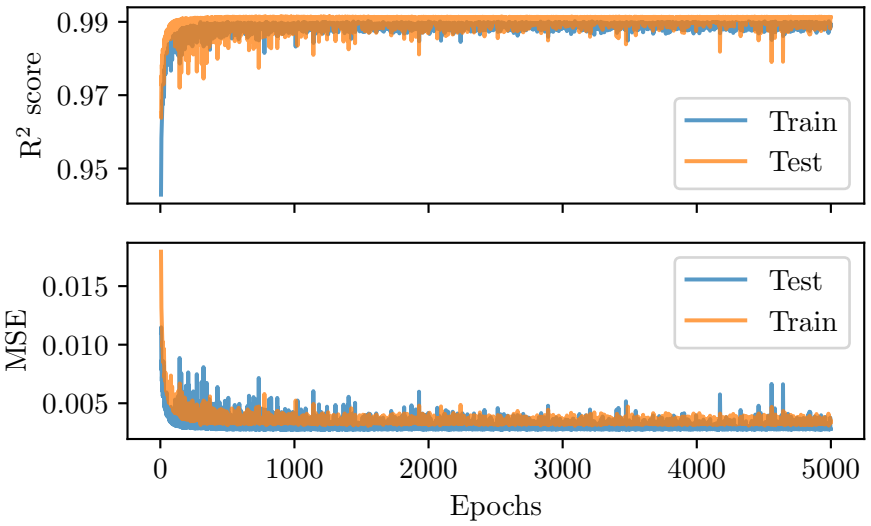
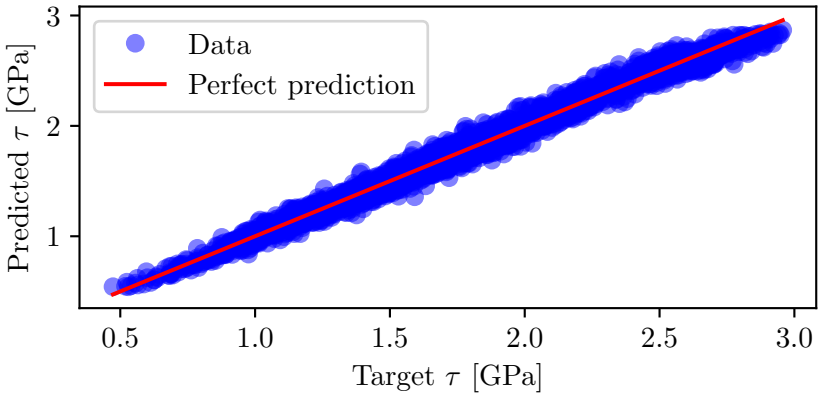


Figure 11.12: Yield shear stress as we increase generations of the full-search approach to the search algorithm.



(a) R^2 and MSE during training.



(b) Predicted yield by model compared to true value.

Figure 11.13: Results from the training for the 5th generation of the full-search approach for search algorithm. Final R^2 = 0.9913 and MSE= 0.0028 (RMSE = 0.052 GPa).

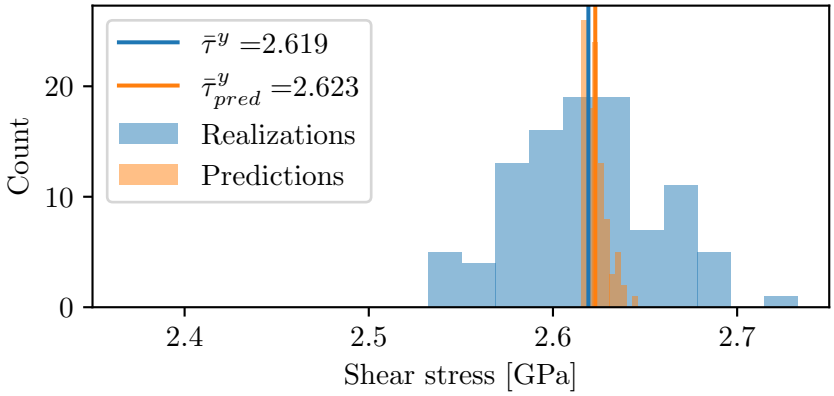
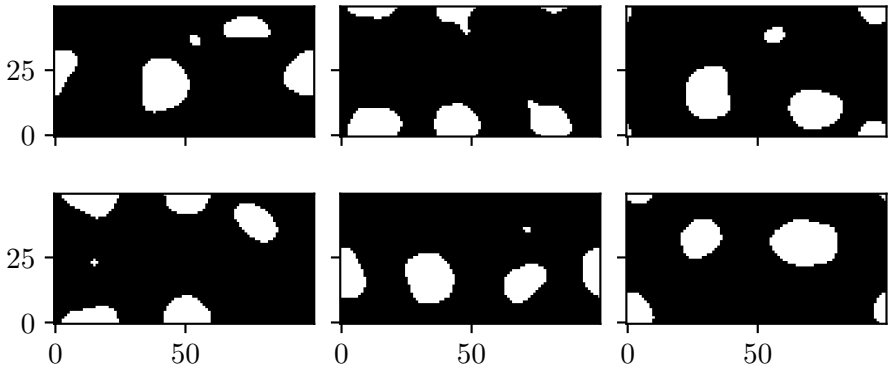
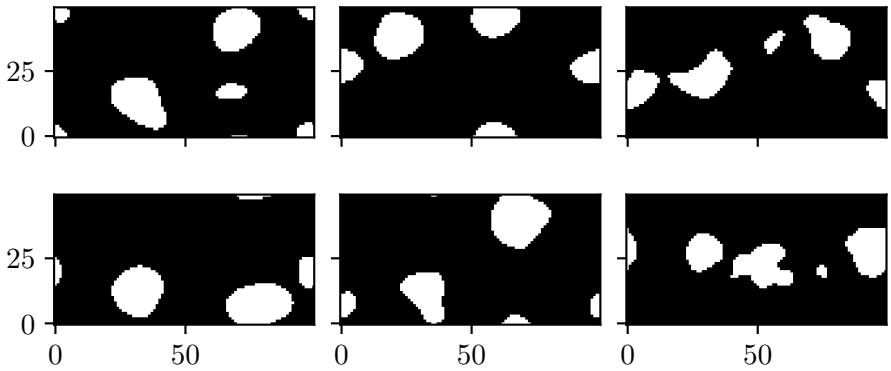


Figure 11.14: Distributions of yield shear stress for predictions and realizations after searching 74 million unseen geometries. The model was trained with Simplex noise, and had undergone 15 generations of the search algorithm, where the last 5 were performed with the full-search approach.



(a) Highest yield shear stress. $\bar{\tau}^=$2.715 GPa.



(b) Lowest yield shear stress.$\bar{\tau}^y$ = 2.424 GPa.

Figure 11.15: Comparison of the 6 strongest and weakest geometries from the 1500 geometries which were created up until 5th generation of the full-search approach to the search algorithm.

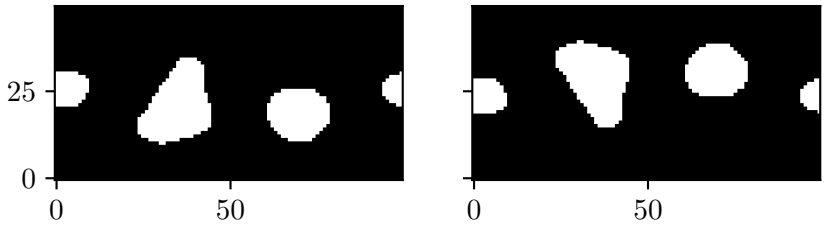


Figure 11.16: Comparison of original (left) and flipped (right) geometry. For this two dimensional case we can consider that the shearing force is applied along the x-axis (to the right in the paper plane).

Chapter 12

Effects of Low and High Resolution

When representing shapes in low resolution we can often see that the pixels as squares which make up the objects in an image. The geometries we are working with are coarse-grained, which means we might represent objects in such a way that circular shapes are represented as squares. We suspected there might be stress concentrations due sharp edges as a result of the coarse-grained geometries we use to create α-quartz material structures, these sharp edges can cause a drop in yield stress. We found no such effects when comparing yield stress between low and high resolution. For the high and low resolution geometries we found no difference in the yield stress as can be seen in Figure 12.1. If there were effects from sharp edges, which caused stress intensities and hence weakened the system, we would expect not expect the yield shear stress to be similar for the case of low and high resolution disks.

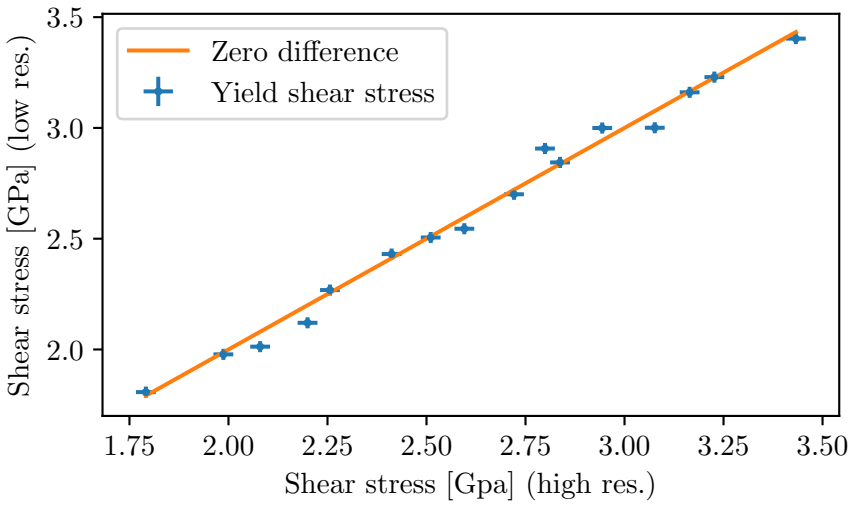


Figure 12.1: Comparison of yield shear stress for disks made from high and low resolution representations of a disk. Data includes MD precision for both x and y.

Chapter 13

Principal Component Analysis and Autoencoders

Principal component analysis and autoencoders can be used to represent data in a lower dimensional space by extracting important information in the input. We will experiment with lower dimensional versions of the images representing geometries as inputs to our CNN.

When training the CNN with the reduced data set we expect the computation time to decrease as there are fewer parameters. We removed one layer from the CNN as the output was reduced to 2 pixels in the third convolutional layer. We also restructured the channels such that it grows from $1 \to 8 \to 16$ (compared to $1 \to 16 \to 32$).

When training CNNs on the encoded images from the autoencoder and convolutional autoencoder we observe no noticeable speed up, in fact the network at times comes to a halt, as it is seemingly getting stuck.

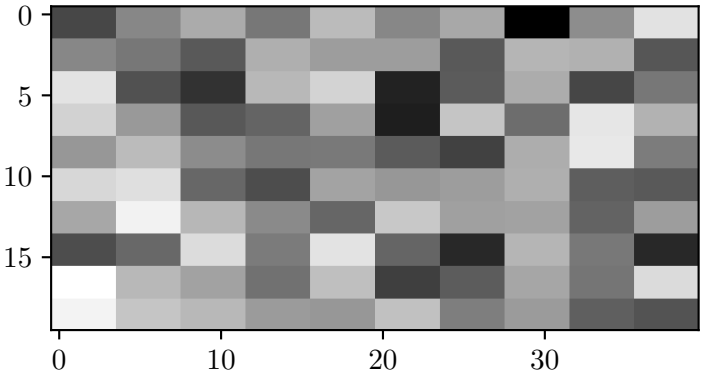
13.1 Principal Component Analysis

For the amorphous silica we studied PCA and autoencoders on randomized and manually made geometries of the 10×10 grid. The images representing the 10×10 grid was 40×20 in terms of pixels, which means we can have in total 800 principal components. The data set consisted of 245 samples for the randomized and 303 samples for the manually made geometries. This part of the study was mainly an exploration to see if we could find something interesting from component analysis.

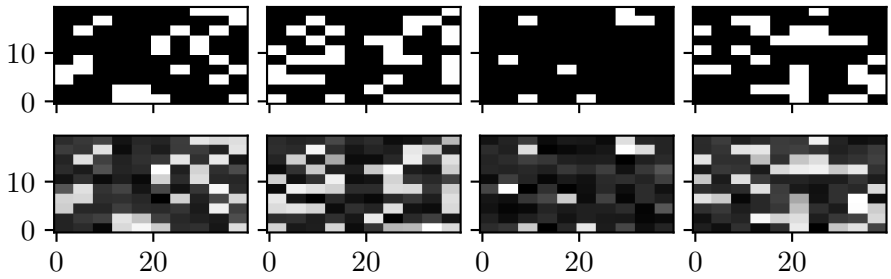
We will first consider the randomized data set. For PCA we found that the first 81 principal components (PCs) contained 96% of the explained variance. By increasing the PCs to 100 we could represent the data 100%, in Figure 13.1b we can see the first principal component, the four geometries and the corresponding inverse transform of the 81 first PCs, while Figure 13.2b shows the same for 100 PCs. For the case of 100 PCs we see that the figures are close to identical, with some minor shades of gray indicating some pixel inequalities with the original image. We then performed a transformation of the input data and naively passed this as inputs to a CNN, as a reshaped 9×9 and 10×10 image, respectively. For 81 PCs

we got an R^2 score of 0.837, and RMSE of 0.518 GPa, while for the 100 PCs we got an R^2 of 0.880 and RMSE of 0.445 GPa. For the manually made geometries saw a 0.999 explained variance with 100 PCs. We passed the PCs as inputs to a one layer deep neural network with yield stress as the target and got an $R^2 = 0.951$ and RMSE $= 0.201$. The principal components themselves look like noise, as can be seen in the first principal component in Figure 13.1a and Figure 13.2a. When applying PCA to the simplex data set containing 3366 samples we could explain 96% of the variance with 1000 PCs. To reach 100% explained variance we had to use all available PCs. Figure 13.3 shows the first PC and the inverse transform. The artifact shown at $\approx [60, 30]$ is most likely a result from the bias that was introduced in the data set, as explained earlier.

We did not study PCA any further. We expect that this method might work for a data set with significant correlations in the input. We could have passed the compressed images (inverse transformed) as inputs to the CNN, but this means we have a poorer representation of the original input, with just as many features. If we were working with a subset of geometries, where we knew each sample had some features which were similar, such as a specific void, we could possibly use PCA to find these features, since it shows significant correlation across the samples.

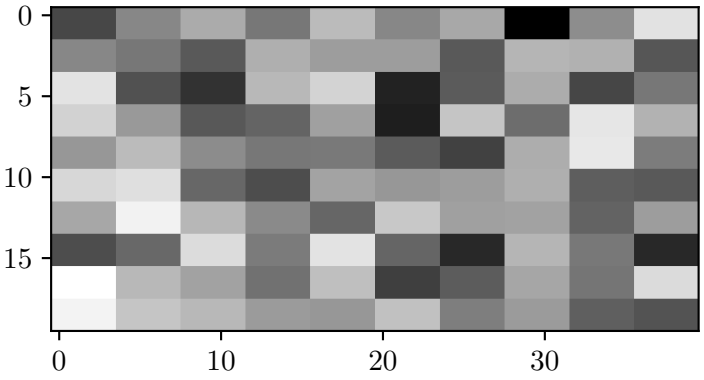


(a) First principal component

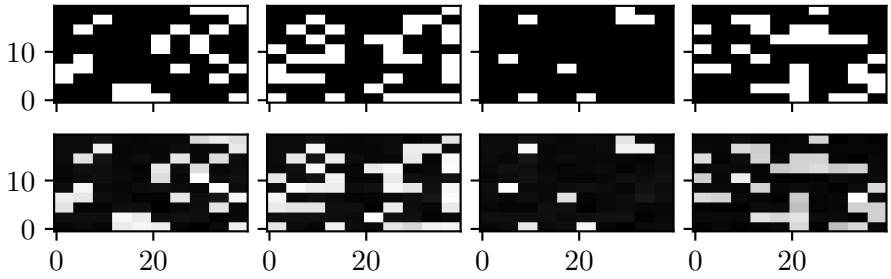


(b) Comparison of the input image (top) and the inverse transformed of the PCA (bottom).

Figure 13.1: Principal component analysis with 81 PCs on the randomized data. 96% explained variance.



(a) First principal component.



(b) Comparison of the input image (top) and the inverse transformed of the PCA (bottom).

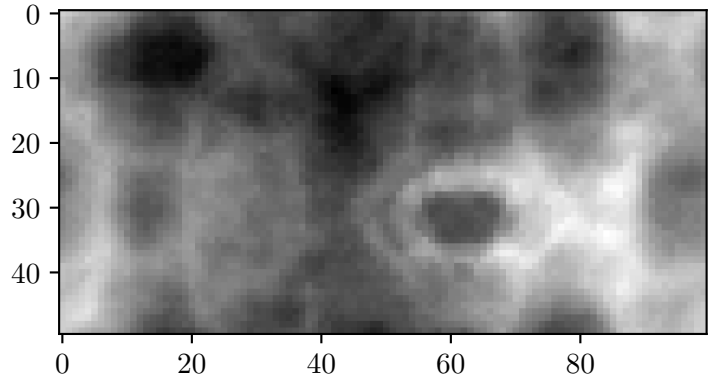
Figure 13.2: Principal component analysis with 100 PCs on the randomized data. 100% explained variance.

13.1.1 Supervised PCA

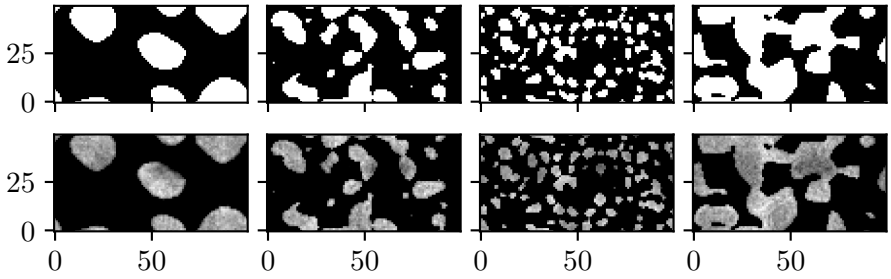
By setting a threshold of 1 for the regression coefficients we reduce the flattened input images to $(245, 376)$ for the case of random data set, and $(303, 520)$ for the manual data set. For the case of randomized data we can explain 95% of the variance with just 40 PCs, while for 47 PCs we explain 100% of the variance. The $R^2 < 0.65$ when we passed the principal components as inputs to a CNN.

13.2 Autoencoder

Performing feature reduction with a standard autoencoder we achieve a reproduction loss of 0.009. In Figure 13.4 we can see the four samples of input vs decoded images. We observe that there were some minor differences in the images, where there are voids that have been reduced in size, or removed. The output from the encoder does not reveal any resemblance to the input images when reshaped. This is as we would expect, since the autoencoder is based on multiple fully connected layers, which we know, by our previous discussion, are not spatially invariant.



(a) First principal component.



(b) Comparison of the input image (top) and the inverse transformed of the PCA (bottom).

Figure 13.3: Principal component analysis with 1000 PCs on the Simplex data set of 3366 samples. 96% explained variance.

The curve for the reproduction loss, and a comparison of the inputs and encoded "images" are found in appendix F. The encoded outputs were passed as inputs to a NN with yield stress as target. The NN consisted of one layer with 20 input features and 1 output. We achieved $R^2 = 0.789$ and MSE$= 0.348$.

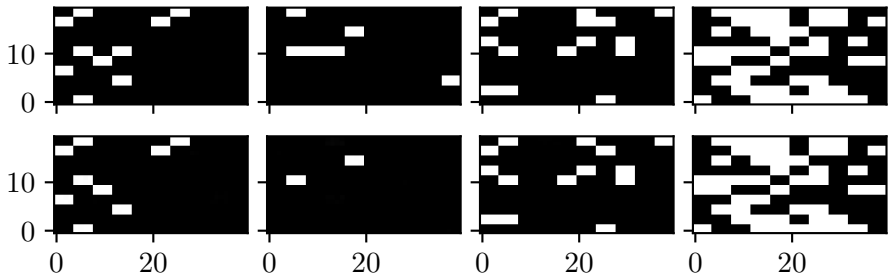


Figure 13.4: Comparison of the original input (top) vs decoded images (bottom) for the autoencoder for random geometries.

13.2.1 Convolutional Autoencoder

The layers of a convolutional autoencoder are spatially invariant, so we would expect the output from the encoder to have some resemblance to the input image. Four samples of geometries and the output from the encoder are shown in Figure 13.5. We see that there are similarities between the encoded and the original images. The model achieved a reproduction loss of 0.087 on the data set consisting of randomized geometries. The output form the decoder is 25×12. By using the encoded images as inputs in a CNN we have greatly reduced the amount of features. Unlike for regular autoencoders we expect the spatial information for the convolutional autoencoder to be saved to some extent. On the randomized data set we got an R^2 score of 0.902 and RMSE of 0.408 GPa with the encoded images as input to the CNN. For a case where the inputs consists of images with little variance such as images of the number 3 as discussed above, this method looks to be effective. For our case we might want to find the extreme cases, previously called "optimal geometry", which has mechanical properties that stand out. By feature reduction we might end up missing out on these features.

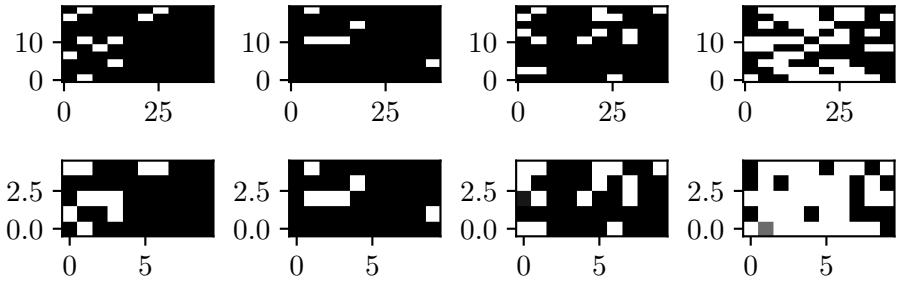


Figure 13.5: Comparison of original input (top) and encoded images (bottom) for the convolutional autoencoder on the randomized data.

13.3 Convolutional Autoencoder Neural Network

For the convolutional autoencoder neural network we got an R^2 score of 0.976 and MSE of 0.293 on the data set consisting of randomized cuts. Figure 13.6 shows the output from the encoder and decoder, we see that the output from the encoder seem to have inverted the values from the input, while the output from the decoder shows some shades of grey at the grid cells where there are voids in the input. The modified loss contains the reproduction error and the MSE of target and prediction, this method seems to get stuck and is not able to further decrease the loss. It was an interesting idea for making the convolutional autoencoder semi-supervised, but unfortunately the autoencoder part seems to not work. We believe the error lies with the encoding and decoding being motivated by minimizing the target.

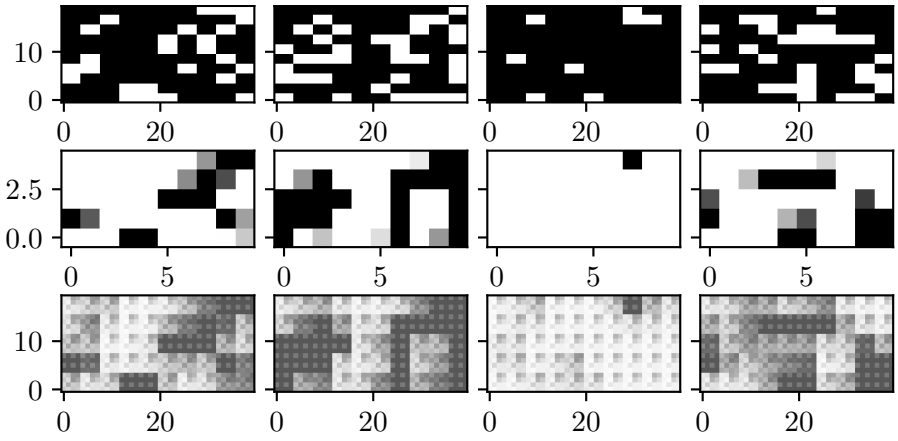


Figure 13.6: Outputs from the decoder (bottom) and the encoder (middle) compared with the original image (top) for the convolutional autoencoder neural network.

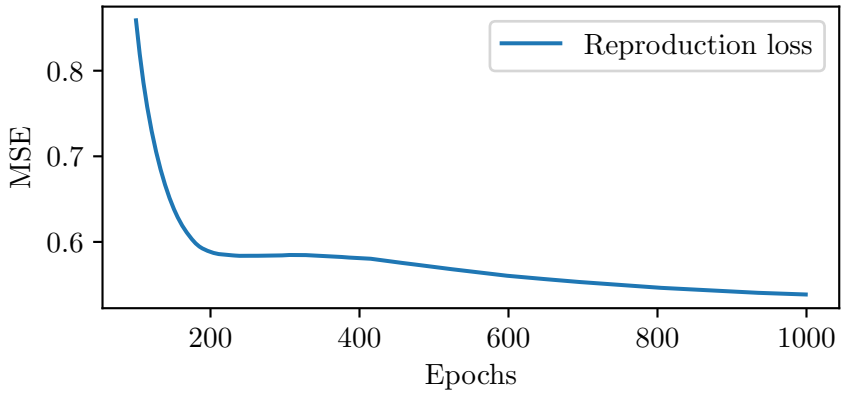


Figure 13.7: Evolution of loss for the convolutional autoencoder neural network.

13.3.1 Summary

PCA showed no interesting results to move forward with. We could inverse transform the lower dimensional representations such that they proved to be similar, meaning there is significant information stored in the principal components. Using the lower dimensional representations as inputs to the CNN did not produce good results. Autoencoders did not work well with images as inputs, but the output from the decoder did show some similarities. The output from the encoder of the convolutional autoencoder was similar to the input, and we got decent results when using the encoded images as inputs to a CNN.

Chapter 14

Griffith's Theory for Cracks

To test our results against Griffith's theory of brittle fracture we fit the yield shear stress as a function of inverse square root of the half crack length, by linear regression. Griffith's theory is valid for infinitely large media, hence the crack is an infinitely small part. The first order correction is to divide the measured yield stress with the relative area of the plane which is perpendicular to the direction we apply the force. The measured yield stress is divided by A_r/A, where A_r is the remaining area of the bulk material after we have removed particles, and A is the total area. A_r was found by calculating the theoretical area of the crack and subtracting it from the total area. There might be some small error since we are calculating the area theoretically, but it should be small as we are working with a resolution that is per particle, as standard in the Python package molecular-builder. The areas were calculated In our case we are applying a shearing force τ_{xz} until fracture, and we are removing particles in the xy-plane. The largest area of the crack was about 10% of the area of the xy-plane.

If our system follows Griffith's theory we expect a linear model to explains the yield stress well. In Figure 2.4 we see that the yield stress is close to the linear fit, as discovered by [13]. We did not pursue any analytical expressions explaining the failure stress or the energy and work found in Griffith's energy balance.

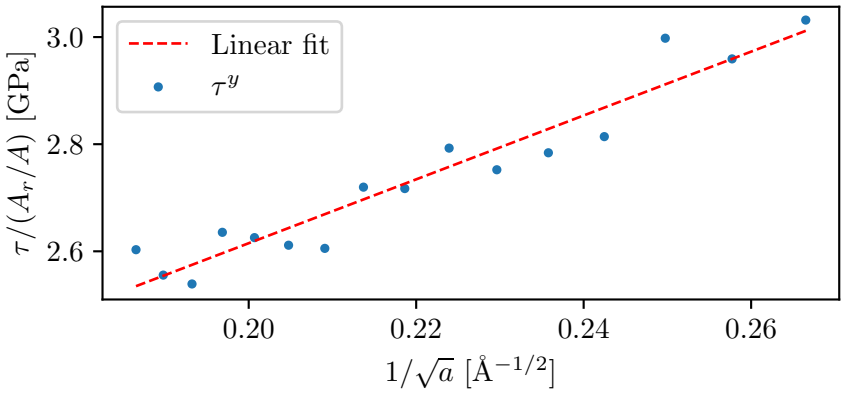


Figure 14.1: Yield shear stress weighted by the relative area of the cracked material, for increasing radius of penny shaped voids. Figure also shows the linear fit of weighted yield as a function of the inverse square root of the radius. We see that the weighted yield follows the linear fit, as Griffith's theory of brittle fractures states.

Chapter 15

Shuffling Voids

The strongest geometry we found through generative design for the target porosity $\phi = 0.15 \pm 0.01$ had a yield shear stress $\tau^y = 2.742$ GPa, Figure 15.1 shows the geometry, which contains three voids. The 3rd generation of the alternative search algorithm found the strongest geometry. We extract each void, as discussed previously, and shuffle each of them before summing the arrays containing the voids. The code is found below for shuffling a single void. We create 200 new geometries through shuffling and perform MD simulations on α-quartz for the given geometries.

```
def shuffle_void(image, shuffle_factor=1, seed=1):
    """Shuffles labeled void by rolling the array.

    Arguments:
        image (arr): Labeled image
        shuffle_factor (int): Factor of shifting. Defaults to 1.
        seed (int): Numpy seed for random choice.

    Returns:
        image (arr): Shuffled image.
    """
    np.random.seed(seed)
    Nx, Ny = image.shape * shuffle_factor
    Nx, Ny = int(Nx), int(Ny)
    x, y = np.random.choice(
        range(-Nx, Nx), size=1)[0], np.random.choice(range(-Ny, Ny
    ), size=1)[0]
    image = np.roll(image, shift=(x, y), axis=(0, 1))

    return image
```

The code for shuffling a image containing multiple voids is shown below.

```
def shuffle_voids_single_image(image,
                                    overlap=False,
                                    max_iter=100,
                                    shuffle_factor=1,
                                    seed=1):
    """Shuffle multiple voids in an image. If overlap is not
    allowed, the method will try to shuffle a given number of
    times.

    Arguments:
```

```
        image (arr): Array containing image values.
        overlap (bool): Whether to allow overlap when shuffling voids.
            Defaults to False.
        max_iter (int): Max iterations of while loop. Defaults to 100.
        shuffle_factor (int): Factor of shifting. Defaults to 1.

    Raises:
        ValuError: If there is no possible shuffling without overlap for current
            max_iter.

    Returns:
        image (arr): Image containing shuffled voids.
    """
    voids = get_voids(image, allow_split=True)
    if overlap: # If we allow overlap, simply shuffle and skip to return.
        for i in range(voids.shape[0]):
            voids[i, :, :] = shuffle_void(
                voids[i, :, :], seed=seed)
    else:
        c = 0
        asp_sum = 2  # Dummy to initiate while loop
        # To check if there are overlapping voids we simply check if there are any value greater than 1, as we are summing binary arrays
        while np.any(asp_sum > 1) and c < max_iter:
            for i in range(voids.shape[0]):
                voids[i, :, :] = shuffle_void(
                    voids[i, :, :], seed=seed + i) # + i to use different seed for each void
                asp_sum = voids.sum(axis=0)
            c += 1
        if c == max_iter:
            print(np.where(asp_sum > 1), np.any(asp_sum > 1))
            raise ValueError(
                'Non-overlap not possible when shuffling voids. Try increasing max_iter.')

    # Sum all the arrays containing voids along axis 0
    return voids.sum(axis=0)
```

We assume that the way the voids are distributed through he cut space is what makes the specific geometry strong. The distribution of the shuffled voids can be seen in Figure 15.3, as the void intensity. We see by the values of the intensities that there are few large concentrations, and no large areas corresponding to zero voids, this means that the voids are distributed well.

We then exposed the trained CNN to the 200 shuffled samples. We used the model that was trained for the 5th generation of the full-search approach of the search algorithm. Figure 15.4 shows how the network performed, comparing the true yield to the predicted yield. We see that the predictions fit fairly well with the observed yield, showing a clear trend. This shuffling might break the patterns we find native in Simplex noise, e.g. we can have large voids very close, which we typically do not observe

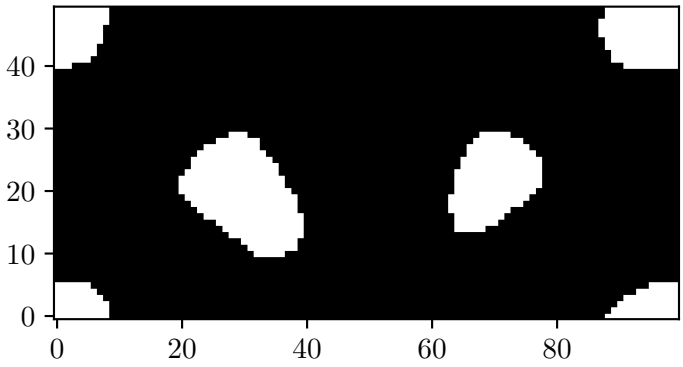


Figure 15.1: Original geometry with $\tau^y = 2.742$ GPa. The geometry contains three voids, two are apparent in the middle, while the last is split across all boundaries.

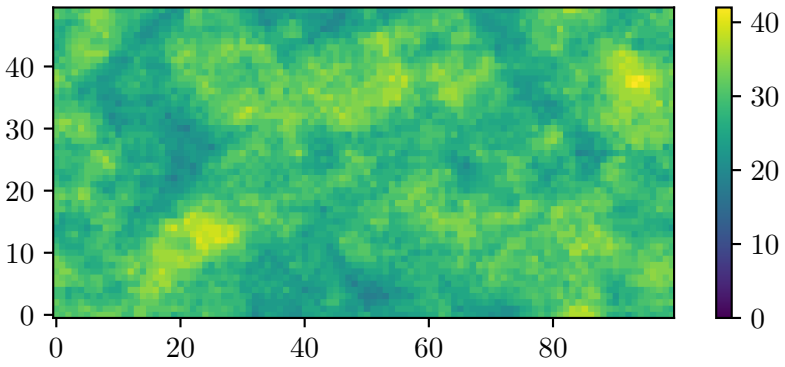


Figure 15.2: Void intensity.

Figure 15.3: Void intensity for the 200 shuffled geometries. The voids for the strongest samples at $\tau^y = 2.742$ GPa was shuffled 200 times to create 200 new samples. The colormap shows how many samples has a void for a specific grid cell.

with Simplex noise. Figure 15.5 shows the true yield stress of the shuffled geometries vs. the value of the original unshuffled true yield. We see that all the shuffled geometries turned out to be weaker, although the strongest shuffled geometry could show the same yield as the strongest geometry, taking the MD precision into account.

Figure 15.6 shows a comparison of the two strongest and two weakest geometries. We observe that for the case of the strongest geometries the voids are placed evenly apart, while for the weaker the voids are placed close. Our intuition tells us that placing voids close together creates a significant weakness in the system, as the areas between the voids become smaller, making it behave as one large void rather than two separate.

If we perform a single prediction of the geometry which was found to

be strongest, it is predicted to have 2.617 GPa yield shear stress. As the model does not recognize the geometry as being strongest, we assume that this is a problem with the representation of strong geometries in the data set. If there is significant similarities between the geometries which the model is trained on, and the strong geometries are few, it is less likely to pick up on some complex relationship in the data which explains why the geometry has a higher yield.

We also tried shuffling the strongest geometry until we found a shuffled sample which was predicted to have a higher yield by the model. This was done by shuffling the voids of the strongest geometry and predict the yield for each shuffled geometry. This was done iteratively until we either hit a threshold for the number of iterations, or if we found a geometry that was predicted to be $\tau^y_{pred,new} = 1.01\tau^y_{pred,strong}$, where $\tau^y_{pred,strong} = 2.617$ GPa is the prediction of the strongest geometry, and not the true yield. We could not find a geometry which was predicted to be 1% stronger during 10000 iterations.

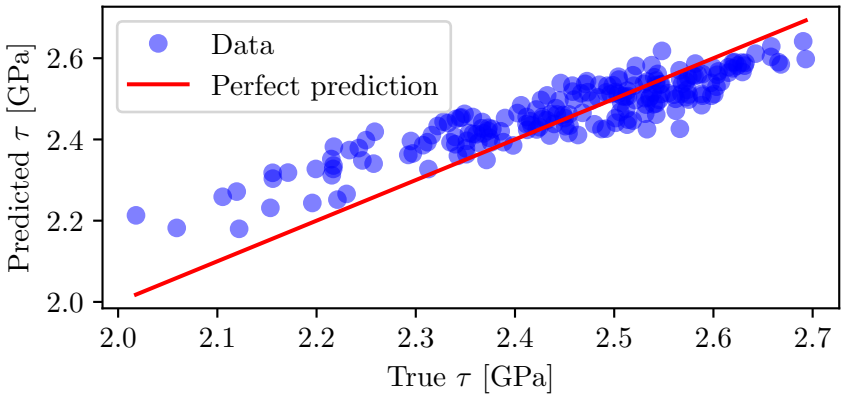


Figure 15.4: Comparison of predicted and true yield shear stress for the shuffled geometries.

15.1 Summary

When shuffling the voids of the geometry that was found to have the highest yield stress by MD simulation, we found that all the 200 shuffled geometries were weaker when performing MD simulations. When exposing the trained CNN to the shuffled geometries we saw that the predicted yield was similar to the true yield.

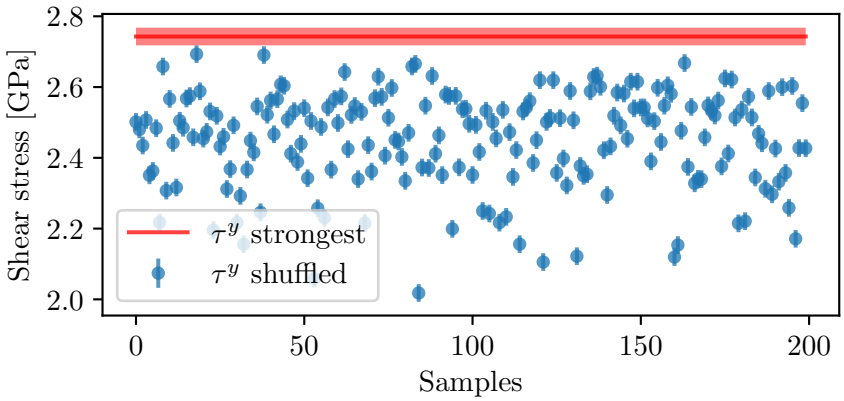
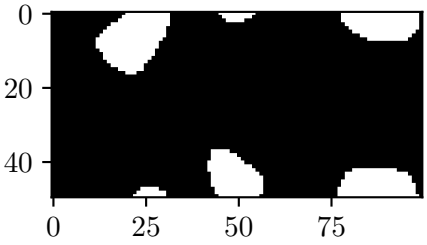
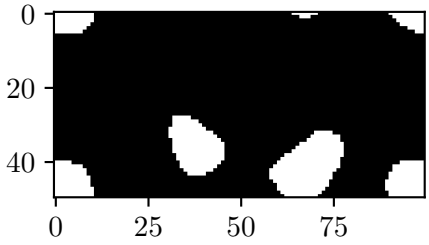


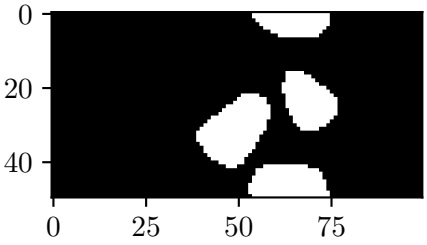
Figure 15.5: Comparison of the yield shear stress for the unshuffled geometry and the shuffled. Including the MD precision RMSE = 0.025 GPa. The strongest geometry was found to have 2.742 GPa yield shear stress. Largest yield for the shuffled geometries was $\tau^y = 2.693$ GPa, lowest $\tau^y = 2.018$ GPa, while the average $\bar{\tau}^y = 2.451$ GPa.



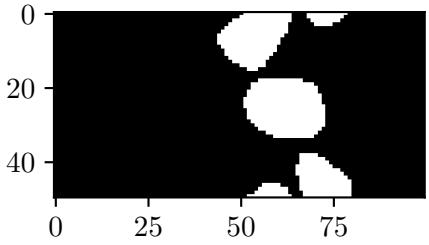
(a) $\tau^y = 2.693$ GPa.



(b) $\tau^y = 2.690$ GPa.



(c) $\tau^y = 2.018$ GPa.



(d) $\tau^y = 2.059$ GPa.

Figure 15.6: Comparison of the two strongest and two weakest samples of the shuffled geometries. Top shows the two strongest, bottom the two weakest. Original geometry shown in Figure 15.1.

Chapter 16

Exposing the Model to New Noise

To challenge the model we trained with geometries made from Simplex noise, we expose it to geometries which was created by different methods.

16.1 Circular Noise

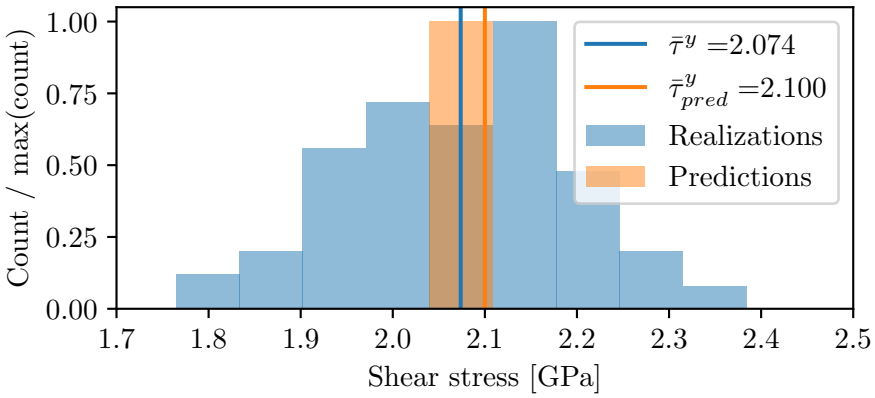
Seeing as many of the strongest geometries show some circular shape to the voids, we find it natural to expose the model to noise in the form of disks, represented by cylinders in the cut space.

16.1.1 Target Yield Shear Stress

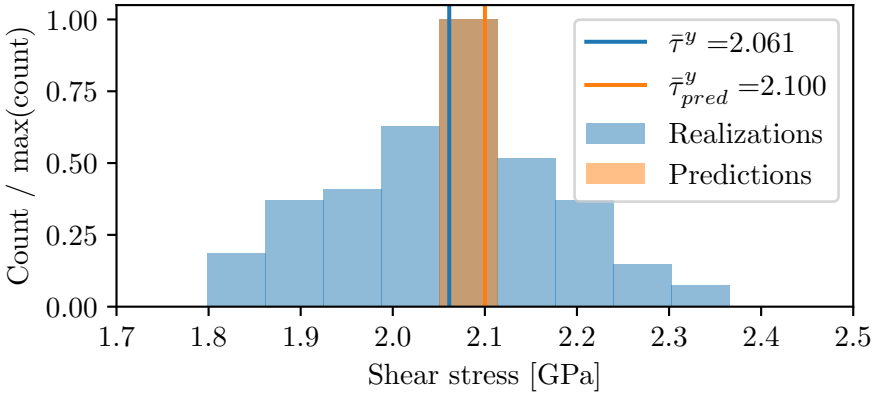
For the case of the circular noise we performed three iterations of the generative design with a target yield shear stress $\tau = 2.1 \pm 0.001$ GPa, and one generation for target $\tau = 1.5 \pm 0.0015$ GPa. The reason we had to choose a larger band, ± 0.0015, is shown in the full distribution of the predicted samples for the 1st generation (which is the same for both target yields) in Figure 16.2b, where we see that there are fewer predictions as the predicted yield stress becomes lower. Figure 16.1 shows the distributions for the stress for the 1st and 3rd generation. We see that the average true yield is close to the average predicted yield, but during the three generations there was no real improvement. The case for target $\tau = 1.5 \pm 0.0015$ is shown in Figure 16.2a.

16.1.2 Target Porosity $\phi = 0.15 \pm 0.01$

Further we tried to restrict the porosity to see how the model performed. Again we chose the predictions which had the highest predicted yield shear stress. Figure 16.3 shows the distribution for the yield, we see that network predicts a yield which is lower than the average true yield.



(a) 1st generation. Standard deviation 0.123 GPa of measurements done by MD simulations.

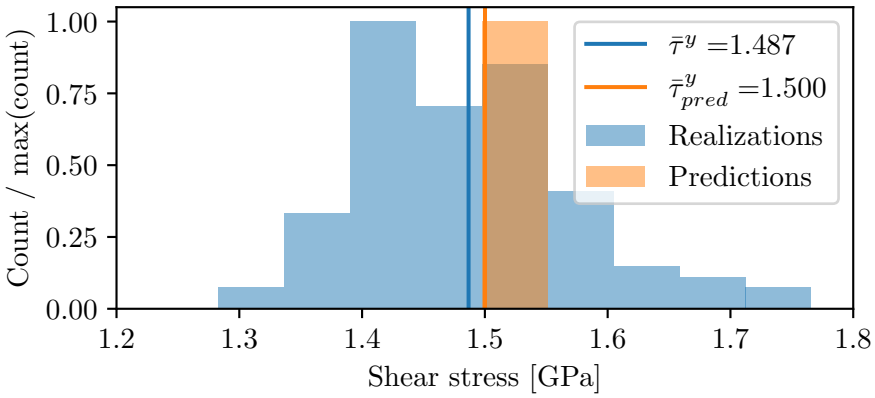


(b) 3rd generation. Standard deviation 0.115 GPa of measurements done by MD simulations.

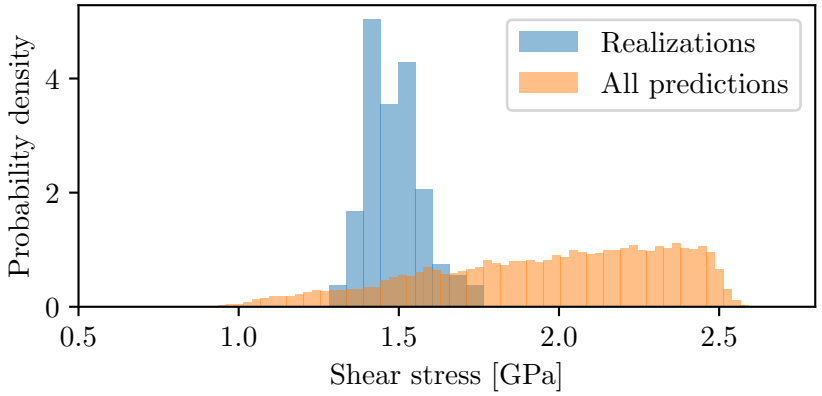
Figure 16.1: Distribution of stress for generative design using circular noise. Predictions are made with the model which was trained with geometries from Simplex noise. Target shear stress $\tau = 2.1 \pm 0.001$ GPa.

16.2 Randomized Squares Revisited

In most cases Simplex noise creates voids that are circular, or they have some smooth boundary. The representation of the Simplex noise is bounded by the resolution of the grid we are working with, but in most cases we observe a smooth boundary, unless the voids have small radii. To challenge the model we expose it to random squares as we did initially, prior to applying simplex noise. The squares are represented as cubes in the cut space.



(a) Standard deviation 0.088 GPa.



(b) Full distribution of for the first generation of the circular noise. The predictions are the same as for the 1st generation at target $\tau = 2.1 \pm 0.001$ GPa.

Figure 16.2: Distribution of stress for generative design using circular noise. Predictions are made with the model which was trained with geometries from Simplex noise. Target shear stress $\tau = 1.5 \pm 0.001$ GPa.

16.2.1 Target Yield Shear Stress

First we chose to target $\tau = 2.1 \pm 0.001$. Figure 16.4 shows the distribution of yield shear stress and the full distribution of predictions. We see that the model predicts these geometries as stronger than they prove to be by simulations. The standard deviation was 0.347 GPa. We did not expect the model to perform well on this case.

16.2.2 Target Porosity $\phi = 0.15 \pm 0.01$

Further we tried restricting the porosity and realize the geometries which had the highest yield shear stress. Figure 16.5 shows the distribution of stress. We see that the model predicted the geometries to be strongest than

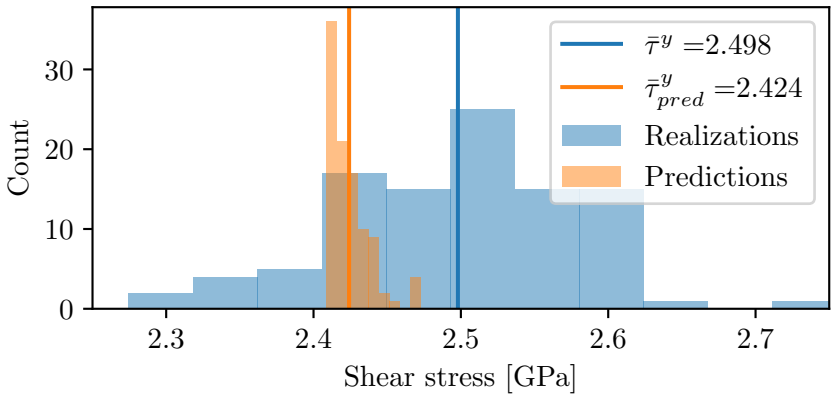


Figure 16.3: Distribution for the yield stress for target porosity $\phi = 0.15 \pm 0.01$ for the circular noise. Predictions are made with the model which was trained with geometries from Simplex noise. Standard deviation 0.080 GPa of measurements done by MD simulations.

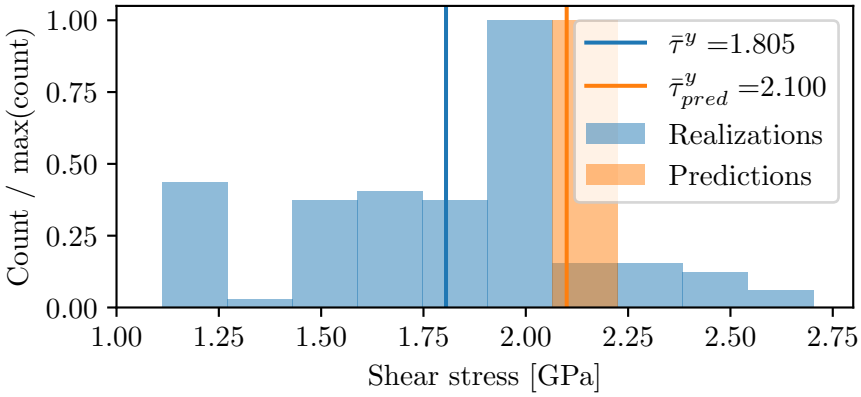


Figure 16.4: Distribution of stress for the squared noise on the model which was trained with geometries from Simplex noise. Target shear stress $\tau = 2.1 \pm 0.001$ GPa. Standard deviation 0.347 GPa of measurements done by MD simulations.

they were. The distribution however has a standard deviation of 0.030 GPa, and we would expect the network to be able to accurately predict the yield, for this specific case, after some iterations of the search algorithm.

16.3 Summary

The model trained on Simplex noise showed good performance on the case of circular noise when we target specific yield stresses and limiting the geometries to a specific porosity, while choosing the predictively strongest geometries. For the case of squared noise the model did not perform well.

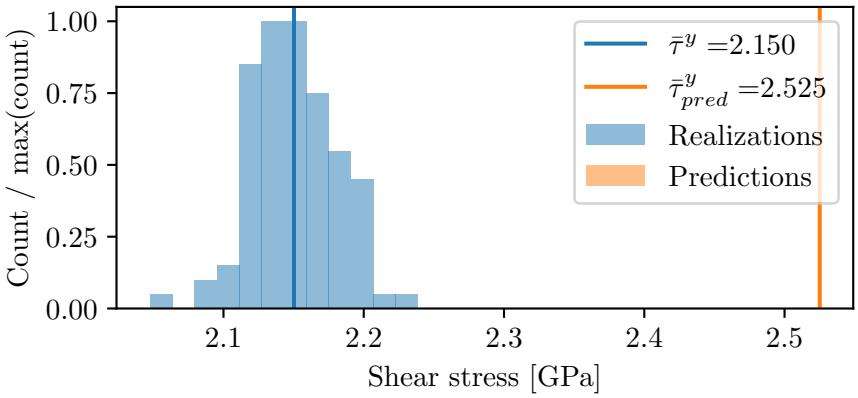


Figure 16.5: Distribution of stress for the squared noise on the model which was trained with geometries from Simplex noise. Target porosity $\phi = 0.15 \pm 0.01$ GPa. Standard deviation 0.030 GPa of measurements done by MD simulations.

Chapter 17

Explainability – Saliency Maps

Explainability is a term used for exploring methods which explain what neural networks has learned. Saliency maps shows what a neural network finds to be important for a specific input. We will present the findings as heatmaps, showing the intensity of the importance of the feature.

When comparing saliency maps of CNN with kernels 3×3 and 9×9 we find that the latter option yields results that are more in line with the observed stress. Figure 17.1 shows a comparison of the time averaged τ_{xy} for atoms binned according to unit cell size, saliency map and original structure for convolutional kernel of size 3×3, while in Figure 17.2 we see the same for 9×9 kernel. Figure 17.3 shows the time averaged displacement of atoms, binned as discussed above for convolutional kernel 3×3, Figure 17.4 shows the same for 9×9 kernel. Recall that saliency maps shows which features are most sensitive to a change in the input, a small perturbation of these input features leads to a large perturbation in the output. We observe that for the smaller kernel there is a clear trend that the features inside the voids have the most importance. For the larger kernel, the areas between the voids and the shape of the voids seems to carry the most importance. When comparing the saliency maps for the two kernel sizes we see that the larger kernel shows importance for areas that are close to or at the areas of high values of stress. For the case of displacement it is less obvious that the larger kernel performs better, but comparing the 3×3 kernel with the displacement, there are is not much overlap. Based on the total evaluation of both displacement and stress we believe 9×9 convolutional kernel is a better choice for results which are in line with physics.

In semantic image segmentation[1] the size of the kernel is important when classifying multiple objects in an input image [41]. A CNN with a small effective receptive field may struggle to learn information about large objects [35]. The kernel size, and therefore the receptive field, must be large enough to capture objects of relatively large size. In our case we are not classifying objects located in the input, but we can consider a non-classifying neural network as a classifier with an infinite amount of

[1]Semantic image segmentation is a method for classifying each pixel in an input image. E.g. an image containing a car and a person should classify car, person and background.

classes. We want to decide if a subset of pixels are important considering the output, which is the same mindset as deciding if a subset of pixels are important when trying to find a specific object. Whether we are looking for a specific object, or *some* object, that maximizes the output is in practice the same when considering how and what the network is learning. We will ultimately benefit from the network having a larger field of view when the objects (voids in our case) are of varying size, shape and numbers.

If the network had learned any physics, we expect that the saliency map shows high intensity in places where we observe large stress intensities or large displacement. Figure 17.1 and Figure 17.2 shows the stress fields compared to saliency map for convolutional kernels 3×3 and 9×9, respectively. We see that the 9×9 kernel is more in line with the observed stress field.

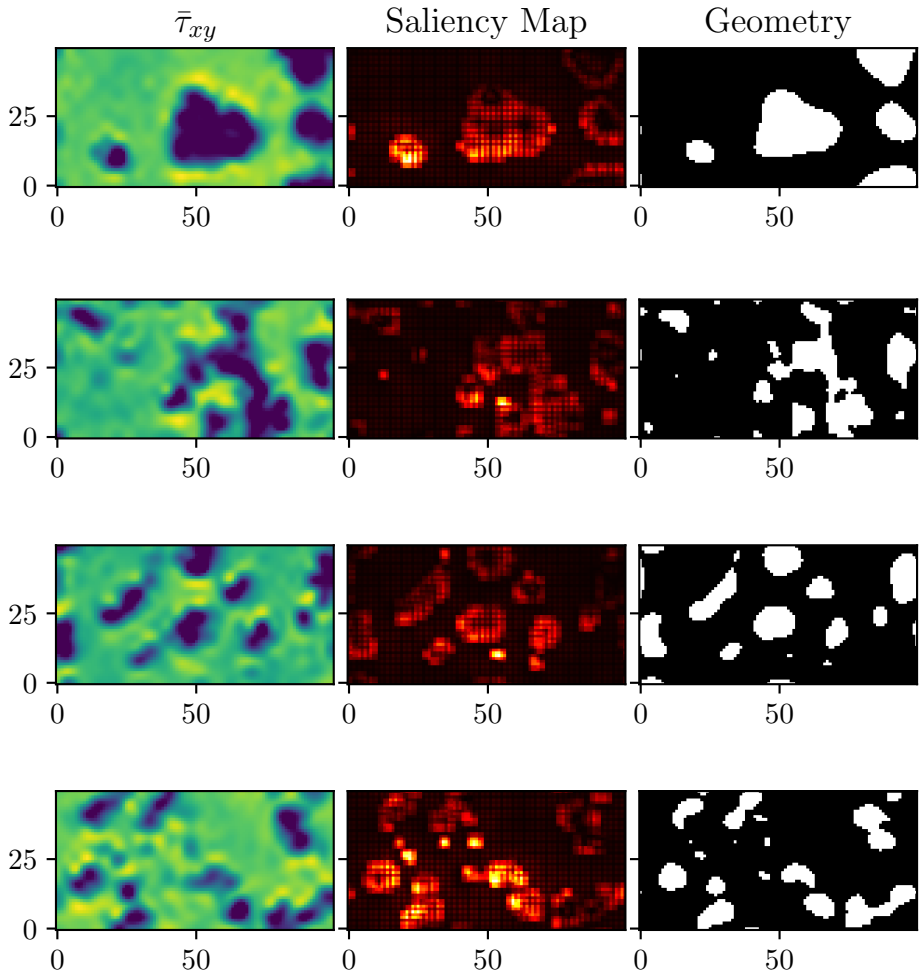


Figure 17.1: Comparison of time averaged τ_{xy} prior to yield, saliency map and original geometry. The stress was time averaged over the last 25000 steps and binned according to unit cell size 28×13. CNN trained with a 3×3 convolutional kernel. Figures for stress made using Ovito, Stukowski [50].

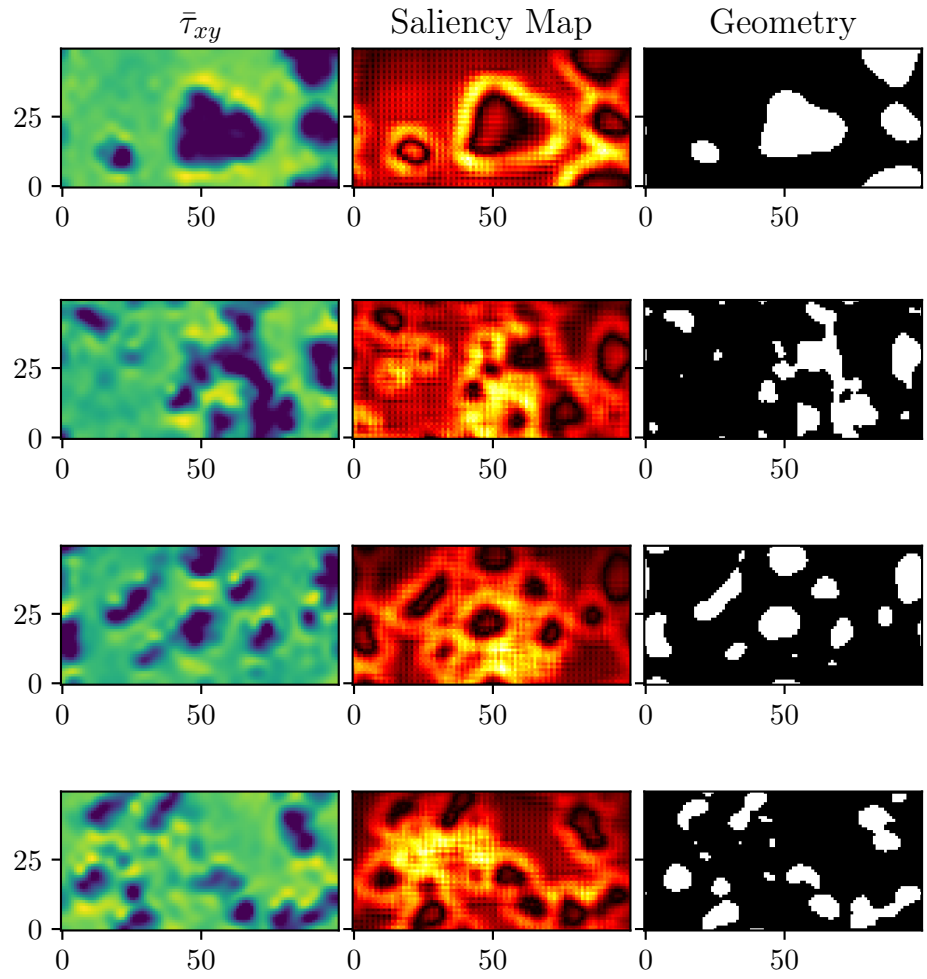


Figure 17.2: Comparison of time averaged τ_{xy} prior to yield, saliency map and original geometry. The stress was time averaged over the last 25000 steps and binned according to unit cell size 28×13. CNN trained with a 9×9 convolutional kernel. Figures for stress made using Ovito, Stukowski [50].

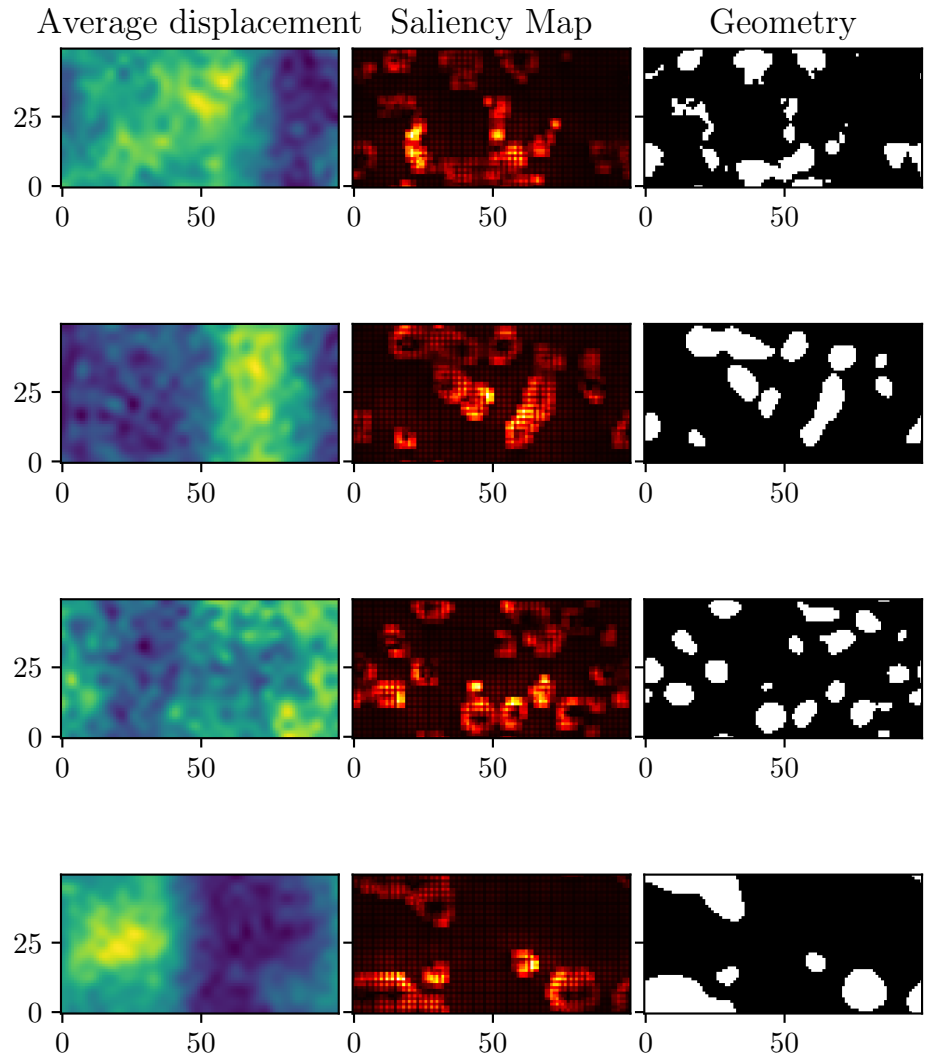


Figure 17.3: CComparison of time averaged displacement prior to yield, saliency map and original geometry. Displacement was time averaged over the last 2500 time steps prior to yield. Spatial binning was according to unit cell size 28×13. CNN trained with a 3×3 convolutional kernel. Figures for displacement made using Ovito, Stukowski [50].

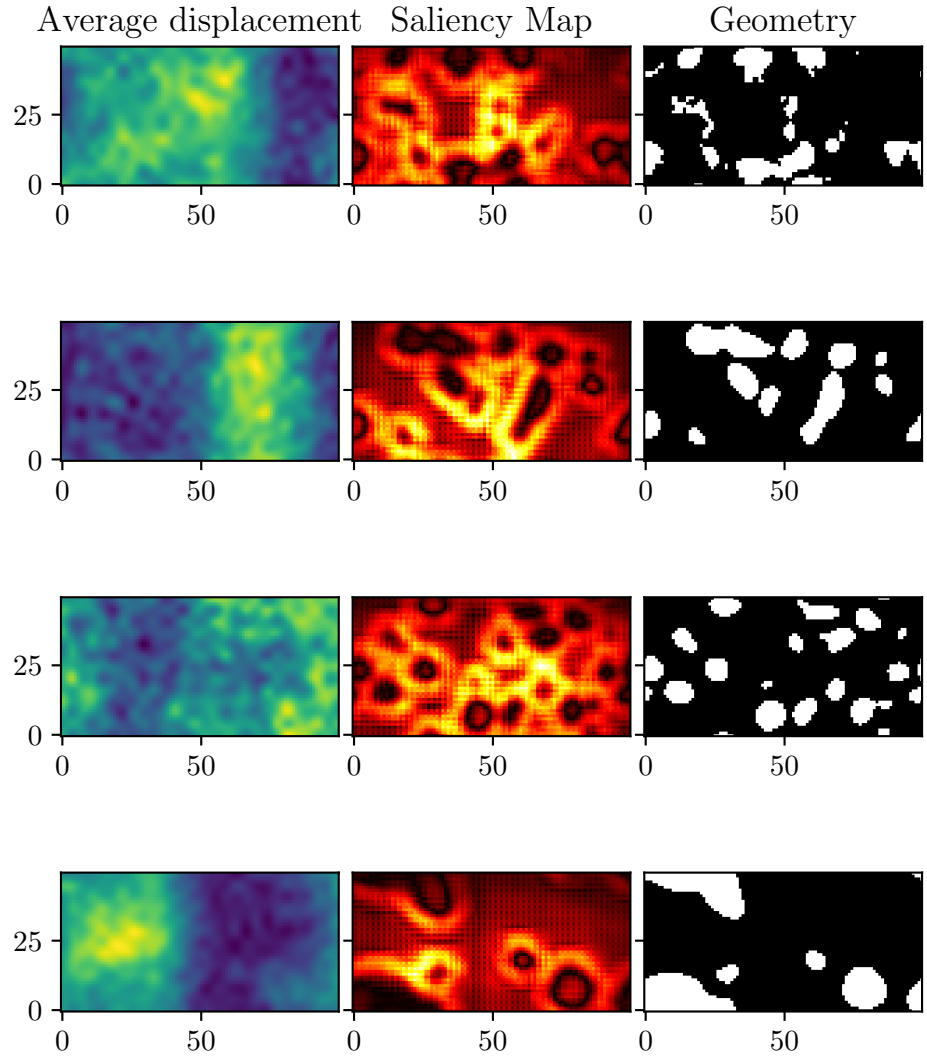


Figure 17.4: Comparison of time averaged displacement prior to yield, saliency map and original geometry. Displacement was time averaged over the last 2500 time steps prior to yield. Spatial binning was according to unit cell size 28×13. CNN trained with a 9×9 convolutional kernel. Figures for displacement made using Ovito, Stukowski [50].

Chapter 18

Implementation

We will in the following go through how we used MD to simulate α-quartz, how we set up the CNN, autoencoder, convolutional autoencoder and how we created geometries for the trained model to evaluate.

18.1 LAMMPS Script

Following is the LAMMPS script for the α-quartz simulations.

To begin with we set variables for the temperature, timestep and decide how long we run the simulations for by $t/\Delta t$, which in units of choice means we are choosing the appropriate amount of steps that is required to run for t ps. We also define the strain rate and set a seed.

```
variable        T equal 300.0
variable        dt equal 0.00075
variable        N equal $(ceil(120 / v_dt))
variable        Nrelax equal $(ceil(30 / v_dt))

variable        srate equal 0.02

variable        seed equal 23898
```

To use the Vashishta potential we must enable calculations for Newton's third law for pairwise and bonded interactions.

```
newton          on
units           metal
dimension       3
boundary        p p p   # Periodic boundary conditions
atom_style      atomic

read_data       alpha_quartz_ortho.data

mass            1 15.9994
mass            2 28.0855

timestep        ${dt}
```

As our box is orthogonal, we must enforce a small displacement for the tilt factor we impose a change on, as this is required by LAMMPS.

```
21 change_box       all triclinic
22 change_box       all xy final 0.1
```

Further we set the formulas for pairwise interactions and the coefficients according to the Vashishta potential [5]. The velocities are then created according to the chosen temperature. We then minimize the energy of the system by conjugate gradient method. We also set load balancing.

```
23 pair_style       vashishta
24 pair_coeff       * * SiO.1997.vashishta O Si
25
26 velocity         all create $T ${seed}
27
28 minimize         1.0e-4 1.0e-5 1000 10000
29
30 fix              fixloadbal all balance 1000 1.05 shift xyz 10 1.05
```

We then relax the system by barostating, allowing the volume, and the angles to change by adjusting the tilt factor. In line 21 we changed the initial tilt factor xy to 0.1, and we know from earlier that the tilt factor changes as $T(t) = T_0 \exp(\Delta t t)$, meaning that the tilt factors which are non-zero ($T_0 \neq 0$) are the only angles which are subjected to change. We chose to let the xy-angle change during barostating, as we imposed a small change in the angle earlier. The temperature is set to 300K. After the barostating we perform the main computations with the Velocity Verlet integration scheme for NVE in combination with a Langevin thermostat, including deformation.

```
31 fix              fixnpt all npt temp $T $T $(100.0*dt) tri 1.0 1.0
32                  $(1000.0*dt)
33 run              ${Nrelax}
34 unfix            fixnpt
35
36
37 fix              fixnve all nve
38 fix              fixlang all langevin $T $T 100.0 ${seed}
39 fix              fixdfrm all deform 1 xy trate ${srate}
40
41 run              $N
```

Following is the code snippets for the convolutional neural network which was used to predict new generations during the search for new geometries.

18.2 Convolutional Neural Network Code

Below is the convolutional neural network we used for training the model and the search algorithm. There are three convolutional layers, followed by ReLU and max pooling. The dropout is performed between the final convolutional layer and the fully connected layer. This CNN is similar to the one proposed by Hanakata et al. [15], but with different convolutional kernel, weight decay and dropout.

```
1 class Model(nn.Module):
2     def __init__(self):
```

```
        super().__init__()
        self.conv2d = nn.Sequential(
            nn.Conv2d(in_channels=1, out_channels=16,
                      kernel_size=(9, 9), padding=4),
            nn.ReLU(),
            nn.MaxPool2d(kernel_size=(2, 2), stride=2),
            nn.Conv2d(in_channels=16, out_channels=32,
                      kernel_size=(9, 9), padding=4),
            nn.ReLU(),
            nn.MaxPool2d(kernel_size=(2, 2), stride=2),
            nn.Conv2d(in_channels=32, out_channels=64,
                      kernel_size=(9, 9), padding=4),
            nn.ReLU(),
            nn.MaxPool2d(kernel_size=(2, 2), stride=2),
        )

        self.linear = nn.Sequential(
            nn.Dropout(p=0.5),
            nn.Linear(in_features=10880, out_features=1)
        )

    def forward(self, x):
        x = self.conv2d(x)
        x = x.view((-1, x.shape[1] * x.shape[2] * x.shape[3]))
        x = self.linear(x)

        return x
```

The code was written for Pytorch – a popular Python package for machine learning. The next section introduces the code snippet for the autoencoder, which was also written for Pytorch.

18.3 Autoencoder Code

For our autoencoder we have three linear layers in the encoder and decoder. We used autoencoders on the manual and randomized data, which were images of the size 40×20. The input images are flattened from 40×20 to an 800×1 array. The encoder reduces the features from $800 \rightarrow 400 \rightarrow 100 \rightarrow 20$, where each layer is followed by ReLU activation. The decoder follows the same input and output features as the encoder, but in the opposite order. The two first linear layers are followed by ReLU activation, while the final layer is followed by a Sigmoid function. The Sigmoid function forces the pixel values to be between 0 and 1. When the autoencoder is fully trained, we can pass our data through the encoder and extract the compressed data. Below is the Python code for the autoencoder.

```
class AE(nn.Module):
    def __init__(self):
        super(AE, self).__init__()
        self.encoder = nn.Sequential(
            nn.Linear(in_features=800, out_features=400),
            nn.ReLU(),
            nn.Linear(in_features=400, out_features=100),
            nn.ReLU(),
            nn.Linear(in_features=100, out_features=20),
```

```
            nn.ReLU()
        )

        self.decoder = nn.Sequential(
            nn.Linear(in_features=20, out_features=100),
            nn.ReLU(),
            nn.Linear(in_features=100, out_features=400),
            nn.ReLU(),
            nn.Linear(in_features=400, out_features=800),
            nn.Sigmoid()
        )

    def forward(self, x):
        x = x.view((-1, x.shape[1] * x.shape[2] * x.shape[3]))
        x = self.encoder(x)
        x = self.decoder(x)

        return x
```

18.4 Convolutional Autoencoder Code

The encoder consists of two convolutional layers, where the first increases the channels from $1 \rightarrow 16$, with a kernel size of 3×3 and with padding of 1. The layer is followed by the ReLU activation and then a 2×2 max pooling with a stride of 2. The second convolutional layer then reduces the channels from $16 \rightarrow 1$, followed by ReLU and another identical max pooling. Below is the Python code fro the convolutional autoencoder.

```
class ConvAE(nn.Module):
    def __init__(self):
        super().__init__()
        self.encode = nn.Sequential(
            nn.Conv2d(in_channels=1, out_channels=16,
                      kernel_size=(3, 3), padding=1),
            nn.ReLU(),
            nn.MaxPool2d(kernel_size=(2, 2), stride=2),
            nn.Conv2d(in_channels=16, out_channels=1,
                      kernel_size=(3, 3), padding=1),
            nn.ReLU(),
            nn.MaxPool2d(kernel_size=(2, 2), stride=2)
        )

        self.decode = nn.Sequential(
            nn.ConvTranspose2d(in_channels=1, out_channels=16,
                               kernel_size=(3, 2), stride=2),
            nn.ReLU(),
            nn.ConvTranspose2d(in_channels=16, out_channels=1,
                               kernel_size=(2, 2), stride=2),
            nn.Sigmoid()
        )

    def forward(self, x):
        x = self.encode(x)
        x = self.decode(x)

        return x
```

Since we have chosen to define the encoder and decoder by a sequence of operations (nn.Sequential) for both the autoencoder cases above, we can extract the output from the encoder and decoder by calling

```
model = SomeNN() # Dummy names
encoder_output = model.encode(SomeInput)
decoder_output = model.decode(SomeInput)
```

This makes it simple to work with the autoencoders when we want to either look at the encoder output, or if we wanted to generate new geometries by inserting unseen images into the decoder.

18.5 Code for Generating Simplex Geometries

The following code generates a Python object SimplexGrid which generates $(n_1 \times n_2)$ arrays of Simplex noise values, which are mapped

```
class SimplexGrid:
    def __init__(self, scale, threshold, l1, l2, n1, n2,
                 **kwargs):
        """ Creates tileable Simplex noise.

        Arguments:
            scale (float): Simplex scale.
            threshold (float): Simplex threshold.
            l1, l2 (float): Length of MD system.
            n1, n2 (int): Grid sizes.
        """
        self.grid1 = np.linspace(0, l1, n1)
        self.grid2 = np.linspace(0, l2, n2)
        self.n1 = n1
        self.n2 = n2
        self.noise_grid = np.zeros((n1, n2))
        self.scale = scale
        self.threshold = threshold
        self.kwargs = kwargs
        self.kwargs['repeatx'] = l1 / self.scale
        self.kwargs['repeaty'] = l2 / self.scale

    def simplex(self):
        """ Creates Simplex noise values.

        Randomizes the permutation table for Simplex noise. Array
        noise_vals is calculated using a nested double for loop.
        Nested loops run in C and are therefore significantly
        faste than standard for loops.
        """
        noise_grid = np.zeros((self.n1, self.n2))
        randomize(period=4096, seed=self.seed)  # Randomize noise
        noise_vals = np.array(
                [snoise2(x / self.scale,
                         y / self.scale,
                         **self.kwargs)
                 for x in self.grid1 for y in self.grid2]
           ).reshape(self.n1, self.n2)
        # Choose only values which are large than threshold
        noise_grid += noise_vals > self.threshold
```

```
41
42          return noise_grid
43
44      def __call__(self, seed, base=None):
45          """ Executed the Simplex function with a given seed and
46          potentially a given base.
47
48          Arguments:
49              seed (int): Seed for noise function.
50              base (float): Displacement for coordinates of noise.
51
52          Returns:
53              grid (np.ndarray): Array containing noise values.
54          """
55          self.seed = seed
56          if base is not None:
57              self.kwargs['base'] = base
58
59          grid = self.simplex()
60
61          return grid
```

The code above is executed for generating geometries during prediction. The code above was necessary for creating millions of geometries in a short time. If we were to use molecular-builder we would have had to generate the atom objects for each geometry.

Part III

Summary and Conclusions

Chapter 19

Summary and discussion

The aim for this study was to predict the yield shear stress for various material structures of α-quartz by using a convolutional neural network. We used an algorithm that is inspired by genetics, in how it improves the population of data through careful selection. We explored some different machine learning methods which are typically used for dimensionality reduction, but we found no interesting results to move on with for this study. We had hoped that by the generative design of new geometries we could find some optimal geometry that showed to have a higher yield shear stress than other geometries, but we did not have the time to continue the search for geometries for it to potentially stabilize at some optimal average true yield.

19.1 Discussion

We found that we can accurately predict the yield shear stress using convolutional neural networks for different α-quartz material structures. The performance metrics for the CNN are similar to those found by Hanakata et al. [15] when they predicted the strength for a graphene sheet. Recent study by Yang et al. [54] found that they can accurately predict the loading curve using a convolutional neural network in binary images of composites. The inputs in the CNN used by Yang et al. [54] are similar to ours, in that they are two dimensional binary images. A study by Rahman et al. [44] has shown that CNNs can be used to predict the shear strength for carbon nanotube-polymer interfaces using similar methods to ours.

We found that we could find increasingly stronger geometries by sampling the same space of geometries with generative design. Hanakata et al. [15] showed that they could find the strongest (or close to) geometry after 3 generations when searching the full space of unseen geometries. For our case we cannot sample the entire space. However we saw an increasing strength during five generations of the generative design, when we searched through a constant space containing almost 30 million unseen geometries. We would expect our method to require more generations to stabilize at some optimal geometry, as we are working with a much larger space of geometries. Yu et al. [56] shows that CNNs can be used to

optimize the toughness of nanocomposites through a generative algorithm, by finding how to distributed soft and stiff materials in the composite to optimize the strength. The generative algorithm was performed 100 times for their study. Using these types of methods for designing materials can be an important tool for future material design.

The method we named poly-CNN for rapidly expanding the variance of a data set by evaluating a transformed target value for the CNN can be useful for model building, with the goal of familiarizing the model with a wider range of material structures, for fewer samples than we would get from random sampling. This can greatly reduce the amount of simulations and generations needed to accurately predict material properties, or to locate geometries with specific material properties.

We found that when shuffling the voids the stronger geometries had voids lined up along the shearing direction, while the weaker geometries had voids lined up perpendicular to the shearing direction, the voids being closer together. Pook [42] found that the stress intensity decreases as the distance between the cracks in an array of cracks decreases. The study from Pook [42] considered tensile loading of a cylinder with cracks formed in an array, it was found that the stress intensities decreased as the distance between the cracks along the direction of the applied load decreased. In our case we are applying a shearing force. From the shuffled geometries we found a difference of over 0.6 GPa, where the system was stronger when the voids lined up in the shearing direction. The results we found are in line with observations of stress intensities by Pook [42].

We discussed how CNNs are not periodic, and that this might be problematic, as our MD simulation has periodic boundary conditions, and we therefore made the geometries periodic. We touched upon a potential solution, which was to modify the input images by periodic padding. The method which was mainly used for this study was to pad the input by 1/8th in each direction. We saw a small increase in the performance of the CNN.

When we increased the space of unseen geometries during prediction we found no improvement in the distribution of the realized yield shear stress. We also tried doubling the training data set by flipping the 4866 geometries, yielding a data set consisting of 9732 samples, which also did not improve the distribution. In other words, we tested both increasing the search space and the size of the training data set without any improvement in the results. We believe that a different approach to sampling the Simplex noise might be possible to improve the results. Another approach could be to create a data set which includes geometries made with more than one noise algorithm. We could add use Perlin noise, which is similar to Simplex noise, but with more artifacts, and it is generally slower due to an increase in calculations for each noise value. Cellular noise or Worley noise, which is a type of procedural cellular noise, could be used to choose specific regions where you remove particles, or to create pores that span the material.

We observe that the yield is slowly increasing when we perform generative design while searching the same space of 30 million geometries for each generation. We believe that if were to perform more generative

designs, which means the data set will slowly increase the amount of strong geometries, the model would be able to consistently predict strong geometries.

When we let the model that was trained on Simplex noise predict yield on geometries made by randomized circles, we saw that it performed decent. When applying geometries made from squares we saw that the model performed poorly, as we would expect considering how different this geometry is from Simplex noise.

For purposes of dimensionality reduction we found that principal component analysis and autoencoders did not work well for this specific task. We know that PCA can work well, where one can see similarities of the input data and the principal components with the naked eye, as shown by Hastie [17, p. 537-538] on handwritten digitized images of the number 3. In a study by Yang et al. [54] it was shown that PCA can be used to represent the loading curve in a lower dimensional space. Their study used CNNs with the lower dimensional loading curve as target, and binary representation of composites as inputs to the CNN, similar to the input in our study.

Chapter 20

Conclusion

We have seen that by building a model using convolutional neural networks we can predict the strength for material given a material structure with few generations of the search algorithm. We can use this machine learning model to scan a space of new geometries to create new material structures with specific physical properties. The average predicted yield shear stress for the geometries was found to be very close to the yield shear stress found by molecular dynamics simulations.

We found that the standard deviation of the realized geometries was smaller when we introduced a 9×9 convolutional kernel. We also argued that the observed physics was more in line with the observed saliency map for 9×9 kernel compared to 3×3.

When targeting a specific yield shear stress we saw that the generative designed showed decent results. We also tried different yield stresses to show the robustness of the method. For the extreme case of target $\tau^y = 2.5$ GPa the predicted geometries were found to have ≈ 0.1 GPa larger average yield.

The main attempt at generative design was done by choosing the predictively strongest geometries for a given range of porosity. During the generative design with target porosity $\phi = 0.15 \pm 0.01$, we saw an increase in the average yield shear stress from 2.574 GPa for the first generation of the original approach, to 2.617 GPa in the final generation of the full-search approach. During the first 10 generations we explored 2.95 million new geometries each generation, and we saw no notable increase in the yield. When we changed the approach to searching over the 30 million samples the model had already seen we saw a slow trend of increasing strength.

This method for generating material structures can be a useful tool for applications where we want materials with specific properties.

Graphene shows variations of ≈ 80 GPa in yield for different material structures due to the stretchable nature of the material and out-of-plane buckling. We would expect a machine learning algorithm to find the extreme cases due to the nature of the problem. α-quartz shows smaller variations of ≈ 3 GPa, the material structures are not as defining for the strength as it is for graphene.

When we shuffled the voids of the geometry which had the highest

true yield stress we observed that all the 200 shuffled geometries were weaker. This result indicates that the model has found a candidate which has some quality, which is not just the porosity or shape of the voids, that makes it stronger. Taking the MD precision into account there could be one of the shuffled geometries that has about the same yield as the strongest geometry.

Our final model had MSE which was 0.0028, which equals RMSE $= \sqrt{0.0028} = 0.0529$ GPa, comparing this to the MD precision RMSE $= 0.0262$. We cannot expect the RMSE for our model to become lower than the MD precision, as this is some variance which is not explained by the data.

We have seen that the machine learning algorithm has learned that there is a spacing between the voids that are relevant for the strength of the material.

We found that our system followed Griffith's theory of brittle fracture when it was deformed by shearing.

Chapter 21

Outlook and Future work

During our study we have touched upon some results and methods that got our attention, and some of these methods can be worth expanding upon in a further study. Due to the time span that we have available, we did not get to explore all the interesting possible extensions to this study. Worst of all, we did not get to continue the searching to see if we could find some optimal geometry. In the following we will present some possible future work which is relevant both for this study and some that are relevant for the field of designing materials.

Inverse design of materials using neural networks has been done by methods such as generative adversarial network [26], neural networks in combination with finite element methods for choosing a data set for the training [34], by supervised autoencoders [16] and variational autoencoders [55]. Dimensionality reduction by principal component analysis and autoencoders did not show any interesting results in this study, but it did get us thinking about how to generate new geometries. For the case of the convolutional autoencoder it would be interesting to study the using the decoder to create new geometries. By training a convolutional autoencoder using Simplex noise, we know that the output from the encoder (which is the input to the decoder) bears some resemblance to input image. We believe that it could be possible to generate new geometries by passing unseen images through the decoder. If we trained the convolutional autoencoder with a specific data set which all contain some properties we want, the decoder could create new geometries with similar properties, possibly better. Initial testing seems to indicate we need to show care in what the input to the decoder looks like. With ReLU we observe values in the output from the encoder to be approximately 0, 1, 5 and 7. Possibly the method would benefit from using an activation function, like Sigmoid, which forces the values to be $\in (0,1)$, we would argue that we could apply some threshold here to force the values to be 0 or 1 during training. Hanakata et al. [16] showed that designing and predicting the strength of graphene Kirigami was possible using supervised autoencoders. Extending on the methods from Hanakata et al. [16] to supervised convolutional autoencoders to design more complex material structures could be an interest method for designing materials

with specific properties.

The methods for dimensionality reduction presented here could possibly be used to locate interesting features with a smart choice of data. If we had a data set that was significantly dominated by geometries which have high true yield stress, we could possibly use PCA or autoencoders to find correlations in the input. The features which are highly correlated could correspond to features which the CNN found to be important for maximizing the yield, if we chose a subset of the data which has similar mechanical properties. We could possibly apply PCA or autoencoders on the 1500 samples found by the generative design targeting $\phi = 0.15 \pm 0.01$. If these geometries are significantly correlated, the network has learned some combination of features which maximizes the yield.

When we created geometries for target $\tau^y = 2.5 \pm 0.001$ GPa we saw that the model proposed geometries which was found to have an average $\bar{\tau}^y = 2.592$ GPa. It would be interesting to test a generative design where we challenged the model to find stronger geometries by using the average true yield for generation g as the target value for generation $g + 1$. This would allow the model to use what it has learned in the previous generation to improve the predictions iteratively.

A recent study by Cho and Lin [8] has shown that CNNs with three dimensional convolutional layers can be used to predict adsorption properties of nanoporous media. Using three dimensional CNNs for predicting the strength of an α-quartz system such as presented in this study is a natural next step. For a three dimensional input we could apply three dimensional Simplex noise, or other functions for creating noise.

We discussed how CNNs are not periodic, and that this might be problematic given our choice of method for creating geometries by Simplex noise. We touched upon a potential solution, which was to modify the inputs by periodic padding. A more interesting approach was discovered too late, padding the input instead of padding in the first convolutional layer. How to impose periodicity for CNNs could be important for molecular dynamics simulations when applying functions to create geometries which need to be periodic, to avoid complex behavior at the boundaries.

We could apply different target values for the convolutional neural network based on the input, to further study the robustness of the model. By using both the yield stress and corresponding value of shear strain as targets for the CNN we could optimize the model to find geometries based on their stiffness. Further we could try to use area under the loading curve to optimize the model to predict the toughness of geometries. Another example could be to use a classifying CNN to classify if a material structure has ultimate failure at the first point of yield. There are tons of possible applications for convolutional neural networks to generate materials.

Part IV

Appendix

Appendix A

Graphene

Figure A.1 we can see the MSE and R^2 score as a functions of epochs for the training of model based on data from graphene.

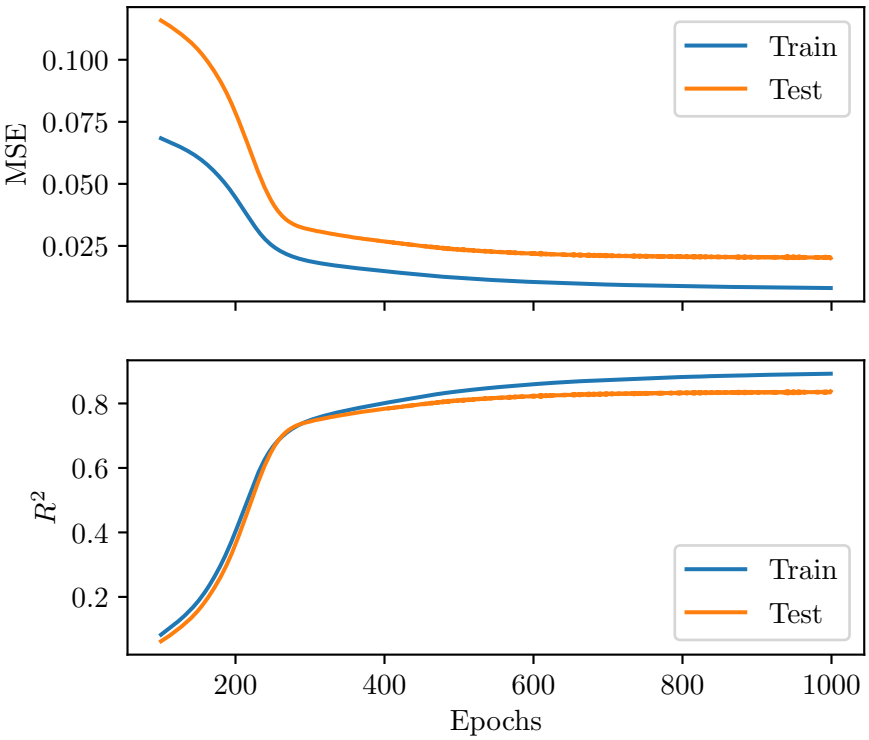


Figure A.1: Training done on 99 samples of graphene geometries. We have omitted the MSE and R^2 for epochs lower than 100.

Appendix B

Simplex

In Figure B.1 we can see a comparison of the simplex scale, porosity and threshold for a part of our data set. We see that there is a clear trend of the porosity being decided by the threshold and Simplex scale.

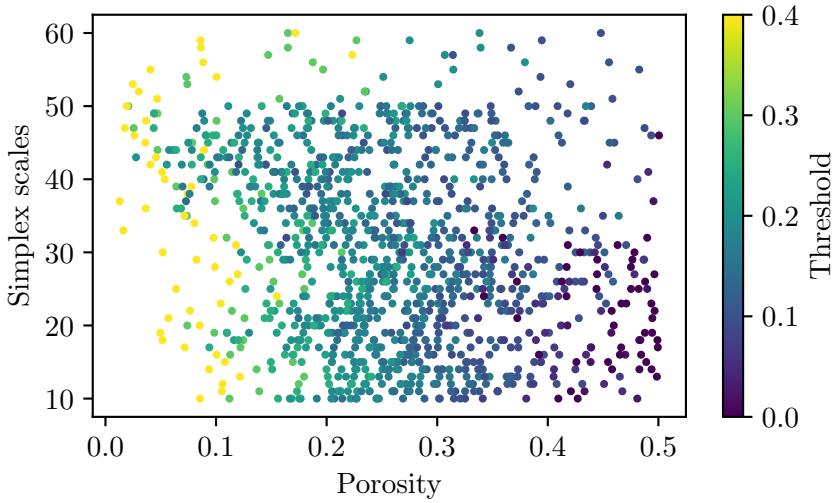


Figure B.1: Simplex scales as a function of porosity and threshold.

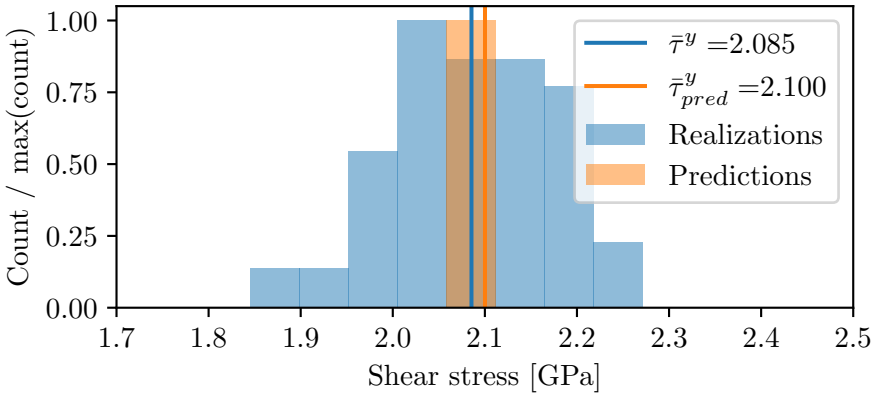
Appendix C

Search Algorithm

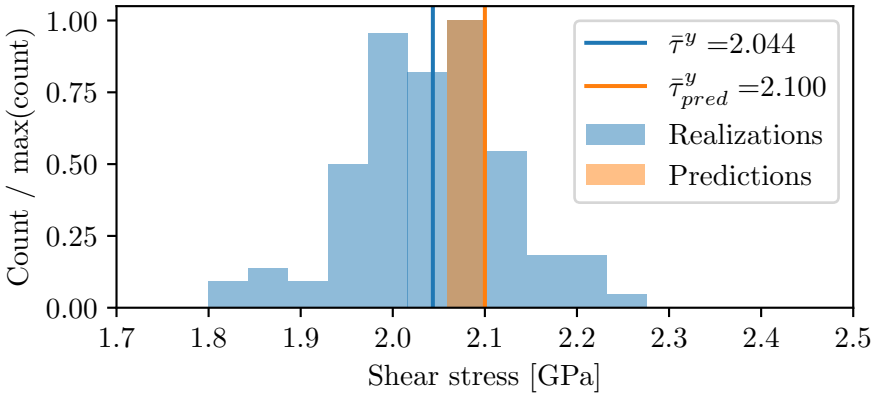
C.1 Target Yield

Figure C.1 shows the distribution for generation 2-4 for the search algorithm.

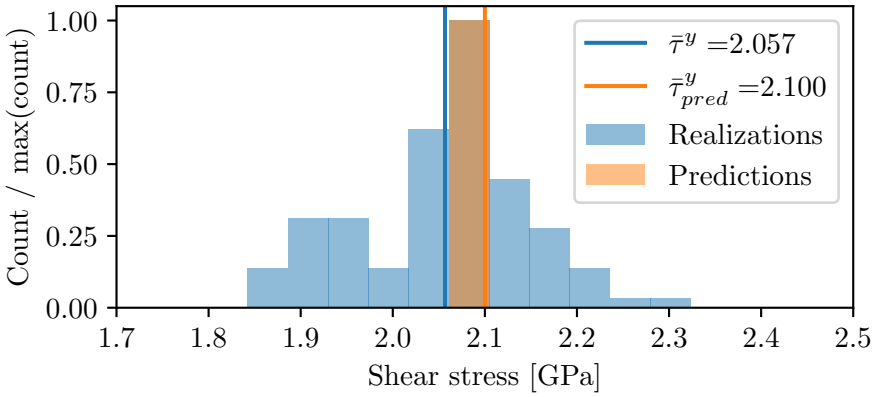
For the narrow search for target $\tau = 2.1$, the distributions are found in Figure C.2.



(a) Second generation. Standard deviation 0.090 GPa.

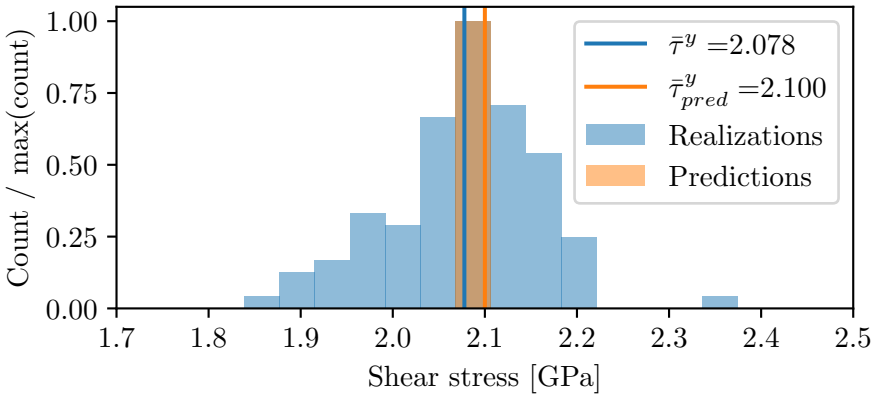


(b) Third generation. Standard deviation 0.084 GPa.

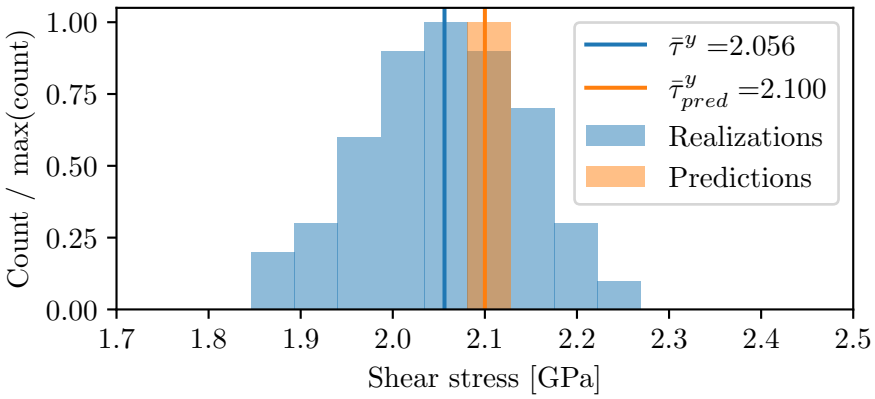


(c) Fourth generation. Standard deviation 0.092 GPa.

Figure C.1: Second-fourth generation for target yield shear stress 2.1 ± 0.001.



(a) 1st generation target $\tau^y = 2.1 \pm 2.5 \cdot 10^{-5}$. Standard deviation 0.086 GPa.



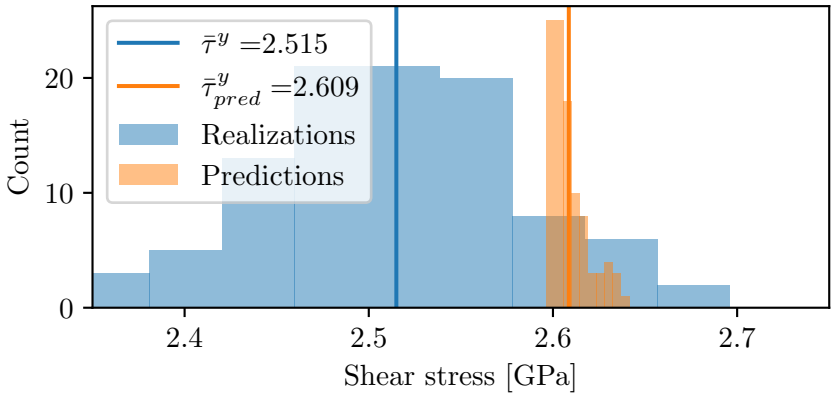
(b) 5th generation target $\tau^y = 2.1 \pm 1.5 \cdot 10^{-4}$. Standard deviation 0.085 GPa.

Figure C.2: Results from searching through a space that contains twice as many geometries as originally. The spread of the target yield was narrowed such that it contained 100 samples.

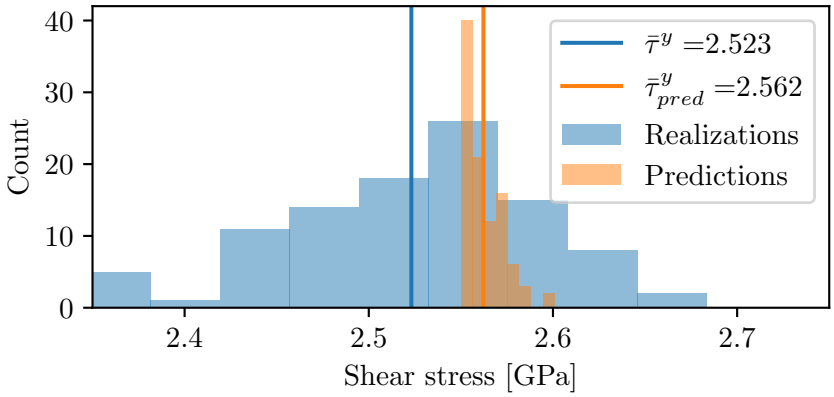
Appendix D

Periodic Padding of the Input by Four Pixels

The method for periodic padding of the input by four pixels, while having no padding in the first convolutional layer was an interesting approach for introducing some periodicity to the CNN. The first two generations are found in Figure D.1.



(a) 1st generation. Standard deviation of 0.072 GPa.



(b) 2nd generation. Standard deviation of 0.068 GPa.

Figure D.1: Distribution of yield for the model with periodic padding of 4 pixels.

Appendix E

Self-Normalizing Convolutional Neural Network

Since the SCNN essentially is CNN with SeLU activation function and alpha-dropout, these were not be discussed in the sections regarding activation functions and dropout in the results. The R^2 score was found to be -24, which means the model has very high variance, if we go as far as calling it a model. The saliency maps shows that the model has picked up something regarding the voids, which was somewhat interesting, see Figure E.1.

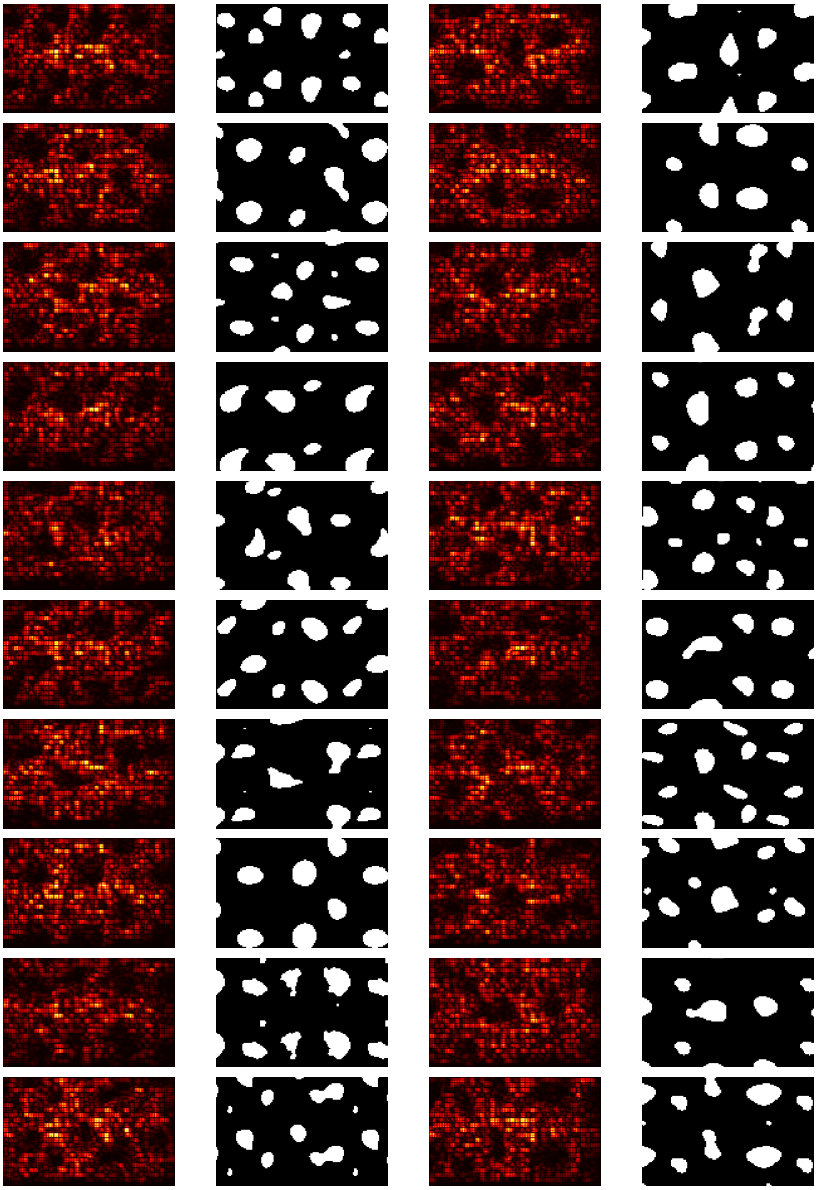


Figure E.1: Saliency maps for self-normalizing CNN.

Appendix F

Autoencoder

Figure F.1 shows the reproduction loss on the randomized data set. Figure F.2 shows a comparison of the input and the reshaped output from the encoder.

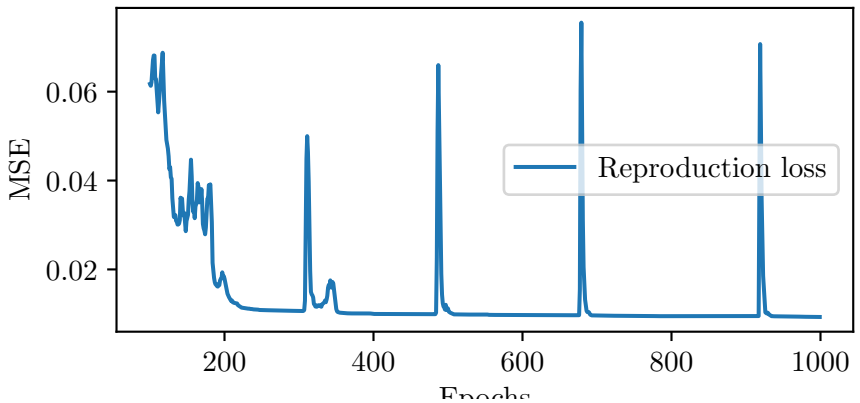


Figure F.1: Reproduction loss for the autoencoder on the randomized data set.

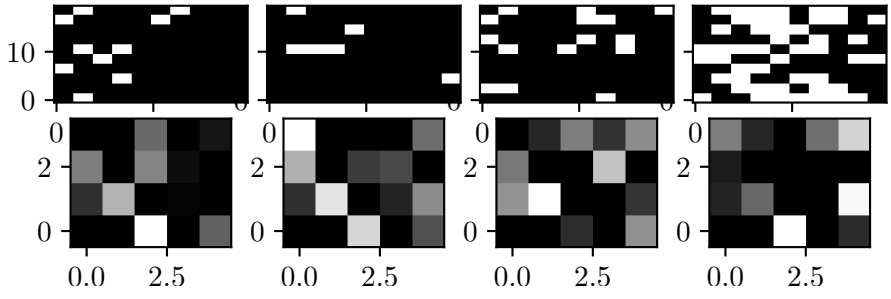


Figure F.2: Input images (bottom)compared to encoded data (top) .

Appendix G

Convolutional Autoencoder

Figure G.1 shows the reproduction loss on the randomized data set. We see that the loss has not stabilized. As the loss stabilized we saw

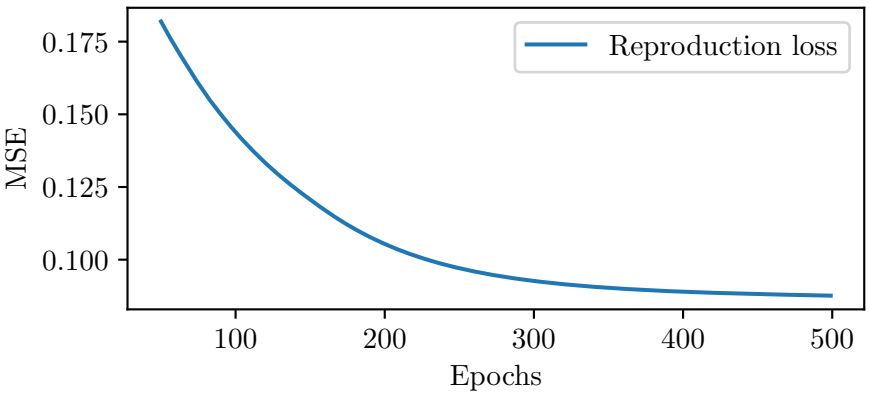


Figure G.1: Reproduction loss for the convolutional autoencoder on the randomized data set.

G.1 Convolutional Autoencoder Neural Network

Figure G.2 shows the reproduction loss on the randomized data set. We see that the loss has not stabilized.

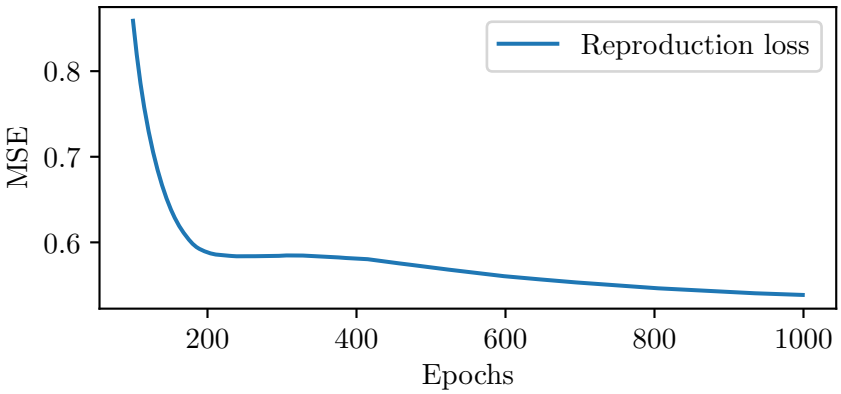


Figure G.2: Reproduction loss for the convolutional autoencoder neural network on the randomized data set.

Bibliography

[1] *Explainable AI: Interpreting, Explaining and Visualizing Deep Learning*, volume 11700 of *Lecture Notes in Artificial Intelligence*. Springer International Publishing, Cham, 1st edition, 2019.

[2] C. C. Aggarwal. *Neural Networks and Deep Learning: A Textbook*. Springer International Publishing AG, Cham, 1st edition, 2018.

[3] H. C. Andersen. Molecular dynamics simulations at constant pressure and/or temperature. *The Journal of chemical physics*, 72:2384–2393, 1980.

[4] T. L. Anderson. *Fracture Mechanics: Fundamentals and Applications*. Taylor & Francis, Boca Raton, Fla, 3rd edition, 2005.

[5] J. Q. Broughton, C. A. Meli, P. Vashishta, and R. K. Kalia. Direct atomistic simulation of quartz crystal oscillators: Bulk properties and nanoscale devices. *Physical review. B, Condensed matter*, 56:611–618, 1997.

[6] M. J. Buehler. *Atomistic Modeling of Materials Failure*. Springer-Verlag, New York, NY, 1st edition, 2008.

[7] L. Chen, L. Wang, J. Miao, H. Gao, Y. Zhang, Y. Yao, M. Bai, L. Mei, and J. He. Review of the application of big data and artificial intelligence in geology. 1684:012007, 2020.

[8] E. H. Cho and L.-C. Lin. Nanoporous material recognition via 3d convolutional neural networks: Prediction of adsorption properties. *The journal of physical chemistry letters*, 12:2279–2285, 2021.

[9] J. Dombi and A. Dineva. Adaptive multi-round smoothing based on the savitzky-golay filter. In *Soft Computing Applications*, volume 633 of *Advances in Intelligent Systems and Computing*, pages 446–454. Springer International Publishing, Cham, 2017.

[10] D. Frenkel. *Understanding Molecular Simulation: From Algorithms to Applications*. Academic Press, San Diego, 2nd edition, 2002.

[11] E. E. Gdoutos. *Fracture Mechanics: An Introduction*, volume 263. Springer International Publishing AG, Cham, 2020.

[12] X. Glorot and Y. Bengio. Understanding the difficulty of training deep feedforward neural networks. *Journal of machine learning research*, 9: 249–256, 2010.

[13] A. A. Griffith. The phenomena of rupture and flow in solids. *Philosophical transactions of the Royal Society of London. Series A, Containing papers of a mathematical or physical character*, 221:163–198, 1921.

[14] A. Géron. *Hands-On Machine Learning With Scikit-Learn, Keras, and TensorFlow: Concepts, Tools, and Techniques To Build Intelligent Systems*. O'Reilly, Sebastopol, CA, 2nd edition, 2019.

[15] P. Z. Hanakata, E. D. Cubuk, D. K. Campbell, and H. S. Park. Accelerated search and design of stretchable graphene kirigami using machine learning. *Physical review letters*, 121:255304–255304, 2018.

[16] P. Z. Hanakata, E. D. Cubuk, D. K. Campbell, and H. S. Park. Forward and inverse design of kirigami via supervised autoencoder. *Physical Review Research*, 2(4), 2020.

[17] T. Hastie. *The Elements of Statistical Learning: Data Mining, Inference, and Prediction*. Springer, New York, NY, 2nd edition, 2009.

[18] K. He, X. Zhang, S. Ren, and J. Sun. Delving deep into rectifiers: Surpassing human-level performance on imagenet classification. In *2015 IEEE International Conference on Computer Vision (ICCV)*, pages 1026–1034. IEEE, 2015.

[19] K. He, X. Zhang, S. Ren, and J. Sun. Deep residual learning for image recognition. In *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, volume 2016-, pages 770–778. IEEE, 2016.

[20] Y. Himeur, K. Ghanem, A. Alsalemi, F. Bensaali, and A. Amira. Artificial intelligence based anomaly detection of energy consumption in buildings: A review, current trends and new perspectives. *Applied energy*, 287:116601, 2021.

[21] G. E. Hinton, N. Srivastava, A. Krizhevsky, I. Sutskever, and R. R. Salakhutdinov. Improving neural networks by preventing co-adaptation of feature detectors. *arXiv*, 1207.0580, 2012.

[22] W. G. Hoover. Constant-pressure equations of motion. *Phys. Rev. A*, 34:2499–2500, 1986.

[23] C. Huang, Y. Shen, Y. Chen, and H. Chen. A novel hybrid deep neural network model for short-term electricity price forecasting. *International journal of energy research*, 45:2511–2532, 2021.

[24] P. H. Hünenberger. Thermostat algorithms for molecular dynamics simulations. *Advanced Computer Simulation*, 173:105–149, 2005.

[25] S. Ioffe and C. Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. *arXiv*, 1502.03167, 2015.

[26] B. Kim, S. Lee, and J. Kim. Inverse design of porous materials using artificial neural networks. *Science advances*, 6:eaax9324–eaax9324, 2020.

[27] D. P. Kingma and J. Ba. Adam: A method for stochastic optimization. *arXiv*, 1412.6980, 2017.

[28] G. Klambauer, T. Unterthiner, A. Mayr, and S. Hochreiter. Self-normalizing neural networks. *arXiv*, 1706.02515, 2017.

[29] A. Krizhevsky, I. Sutskever, and G. Hinton. Imagenet classification with deep convolutional neural networks. *Communications of the ACM*, 60:84–90, 2017.

[30] A. LeNail. Nn-svg: Publication-ready neural network architecture schematics. *Journal of open source software*, 4(33):747, 2019.

[31] X. Li and V. P. Nia. Random bias initialization improves quantized training. *arXiv*, 1909.13446, 2020.

[32] Q.-f. Liu, M. F. Iqbal, J. Yang, X.-y. Lu, P. Zhang, and M. Rauf. Prediction of chloride diffusivity in concrete using artificial neural network: Modelling and performance evaluation. *Construction & building materials*, 268, 2021.

[33] L. Lu, Y. Shin, Y. Su, and G. E. Karniadakis. Dying relu and initialization: Theory and numerical examples. *Communications in Computational Physics*, 28:1671–1706, 2020.

[34] L. Luo, B. Zhang, G. Zhang, X. Li, X. Fang, W. Li, and Z. Zhang. Rapid prediction and inverse design of distortion behaviors of composite materials using artificial neural networks. *Polymers for advanced technologies*, 32:1049–1060, 2021.

[35] W. Luo, Y. Li, R. Urtasun, and R. Zemel. Understanding the effective receptive field in deep convolutional neural networks. *arXiv*, 1701.04128, 2017.

[36] G. J. Martyna, D. J. Tobias, and M. L. Klein. Constant pressure molecular dynamics algorithms. *The Journal of chemical physics*, 101: 4177–4189, 1994.

[37] A. Nakano, L. Bi, R. Kalia, and P. Vashishta. Molecular-dynamics study of the structural correlation of porous silica with use of a parallel computer. *Physical review. B, Condensed matter*, 49:9441–9452, 1994.

[38] S. Nosé. A unified formulation of the constant temperature molecular dynamics methods. *The Journal of Chemical Physics*, 81:511–519, 1984.

[39] R. Pal, A. A. Sekh, S. Kar, and D. K. Prasad. Neural network based country wise risk prediction of covid-19. *Applied sciences*, 10:6448, 2020.

[40] M. Parrinello and A. Rahman. Polymorphic transitions in single crystals: A new molecular dynamics method. *Journal of applied physics*, 52:7182–7190, 1981.

[41] C. Peng, X. Zhang, G. Yu, G. Luo, and J. Sun. Large kernel matters - improve semantic segmentation by global convolutional network. *CoRR*, abs/1703.02719, 2017.

[42] L. P. Pook. Stress intensity factor expressions for regular crack arrays in pressurised cylinder. *Fatigue & fracture of engineering materials & structures*, 13:135–143, 1990.

[43] W. H. Press, S. A. Teukolsky, W. T. Vetterling, and B. P. Flannery. *Numerical Recipes: The Art of Scientific Computing*. Cambridge University Press, Cambridge, 3rd edition, 2007.

[44] A. Rahman, P. Deshpande, M. S. Radue, G. M. Odegard, S. Gowtham, S. Ghosh, and A. D. Spear. A machine learning framework for predicting the shear strength of carbon nanotube-polymer interfaces based on molecular dynamics simulation data. *Composites science and technology*, 207:108627, 2021.

[45] M. Shorfuzzaman and M. S. Hossain. Metacovid: A siamese neural network framework with contrastive loss for n-shot diagnosis of covid-19 patients. *Pattern recognition*, 113:107700–107700, 2021.

[46] K. Simonyan and A. Zisserman. Very deep convolutional networks for large-scale image recognition. *arXiv*, 1409.1556, 2014.

[47] K. Simonyan, A. Vedaldi, and A. Zisserman. Deep inside convolutional networks: Visualising image classification models and saliency maps. *arXiv*, 1312.6034, 2014.

[48] H. Song, X. ying Han, C. E. Montenegro-Marin, and S. Krishnamoorthy. Secure prediction and assessment of sports injuries using deep learning based convolutional neural network. *Journal of ambient intelligence and humanized computing*, 12:3399–3410, 2021.

[49] S. J. Stuart, A. B. Tutein, and J. A. Harrison. A reactive potential for hydrocarbons with intermolecular interactions. *The Journal of chemical physics*, 112:6472–6486, 2000.

[50] A. Stukowski. Visualization and analysis of atomistic simulation data with ovito - the open visualization tool. *Modeling and Simulation in Materials Science and Engineering*, 18, 2010.

[51] S. P. Timoshenko. *Theory of Elasticity*. Engineering societies monographs. McGraw-Hill, New York, 2nd edition, 1951.

[52] P. Vashishta, R. Kalia, J. Rino, and I. Ebbsjoe. Interaction potential for sio_2 : A molecular-dynamics study of structural correlations. *Physical review. B, Condensed matter*, 41:17, 1990.

[53] M. Wieczorek, J. Siłka, and M. Woźniak. Neural network powered covid-19 spread forecasting model. *Chaos, solitons and fractals*, 140: 110203–110203, 2020.

[54] C. Yang, Y. Kim, S. Ryu, and G. X. Gu. Prediction of composite microstructure stressstrain curves using convolutional neural networks. *Materials & design*, 189:108509, 2020.

[55] Z. Yao, B. Sanchez-Lengeling, N. S. Bobbitt, B. J. Bucior, S. G. H. Kumar, S. P. Collins, T. Burns, T. K. Woo, O. K. Farha, R. Q. Snurr, and A. Aspuru-Guzik. Inverse design of nanoporous crystalline reticular materials with deep generative models. *Nature machine intelligence*, 3: 76–86, 2021.

[56] C.-H. Yu, Z. Qin, and M. J. Buehler. Artificial intelligence design algorithm for nanocomposites optimized for shear crack resistance. *Nano Futures*, 3:035001, 2019.

[57] A. T. Zehnder. *Fracture Mechanics*, volume 62. Springer Netherlands : Imprint: Springer, Dordrecht, 1st edition, 2012.

[58] A. Zhang, Z. C. Lipton, M. Li, and A. J. Smola. *Dive into Deep Learning*. 2020. https://d2l.ai.

[59] C. Zhuang, S. Yan, A. Nayebi, M. Schrimpf, M. C. Frank, J. J. DiCarlo, and D. L. K. Yamins. Unsupervised neural network models of the ventral visual stream. *Proceedings of the National Academy of Sciences - PNAS*, 118, 2021.