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

Designs for a negative friction coefficient.

Mikkel Metzsch Jensen

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

Department of Physics
Faculty of mathematics and natural sciences

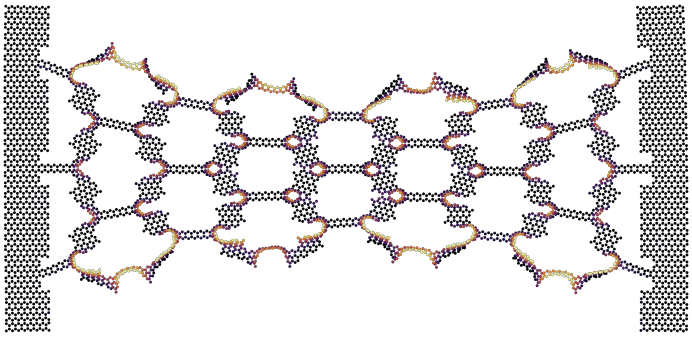
UNIVERSITY OF OSLO

Spring 2023

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

Designs for a negative friction coefficient.

Mikkel Metzsch Jensen



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

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

Printed: Reprosentralen, University of Oslo

Abstract

Abstract.

Acknowledgments

Acknowledgments.

List of Symbols

F_N Normal force (normal load)

Acronyms

CNN Convolutional Neural Network. 8, 9, 10

EMA Exponetial Moving Average. 6

GA Genetic Algorithm. 14

MD Molecular Dynamics. 3, 19

MSE Mean Squared Error. 4

RMSProp Root Mean Square Propagation. 6, 7

SGD Stochastic gradient descent. 5

Contents

Part I

Background Theory

Chapter 1

Machine Learning

We will use machine learning to predict the friction resulting from the stretching and loading of a given Kirigami pattern. To this end, we will generate data through MD simulations that will serve as the ground truth for training the machine learning model. The advantage of using machine learning is that it can significantly speed up the exploration of new configurations compared to full MD simulations. However, there is no guarantee that the machine learning model can accurately capture the physical mechanisms governing our system. Hence, a key objective is to assess the viability of this approach for further studies of Kirigami friction, It is not guaranteed that the machine learning model can accurately capture the physical mechanisms of our system. Hence, one of our objectives is to evaluate the applicability of this approach in the study of Kirigami friction, which we will pursue using a rather traditional machine-learning approach. In this chapter, we introduce the key concept behind machine learning and some of the concepts and techniques relevant to our implementation. For the numerical implementation, we will use the machine learning framework PyTorch [1]

1.1 Neural network

The neural network, or more precisely the *feed forward dense neural network*, is one of the original concepts in machine learning arising from the attempt of mimicking the way neurons work in the brain [2, 3] brain. The neural network can be considered in terms of three major parts: The input layer, the so-called *hidden layers* and finally the output layer as shown in Fig. 1.1. The input is described as a vector $\mathbf{x} = x_0, x_1, \ldots, x_{n_x}$ where each input x_i is usually denoted as a *feature*. The input features are densely connected to each of the *nodes* in the first hidden layer as indicated by the straight lines in Fig. 1.1. Each line represents a weighted connection that can be adjusted to configure the importance of that feature. Similar dense connections are present throughout the hidden layers to the final output layer. For a given note $a_j^{[l]}$ in layer l the input from all the nodes in the previous layer $l-1$ are processed as

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

where $w_{ij}^{[l]}$ is the weight connection node $a_i^{[l-1]}$ of the previous layer to the node $a_j^{[l]}$ in the current layer. Note that having the weight belong to layer l as opposed to $l-1$ is simply a notation choice. $b_j^{[l]}$ denotes a bias and $f(\cdot)$ the *activation function*. The activation function provides a non-linear mapping of the input to each node. Without this, the network will only be capable of approximate linear functions [2]. Two common activation functions are the *sigmoid*, mapping the input to the range $(0, 1)$, and the *ReLU* which cuts off negative contributions

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

Often the same activation function is used throughout the network, except for the output layer where the activation function is usually omitted or the sigmoid is used for classification tasks. The whole process of sending data through the model is called *forward propagation* and constitutes the mechanism for mapping an input $\mathbf{x}$ to

the model output $\hat{\mathbf{y}}$. In order to get useful predictions we must *train* the model which involves tuning the model parameters, i.e. the weight and biasses.

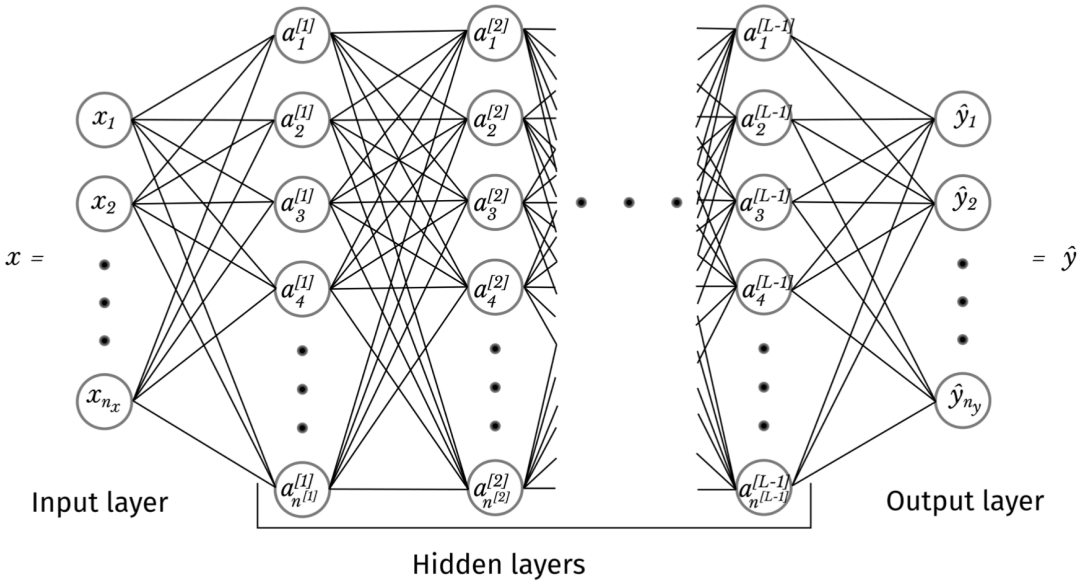


Figure 1.1: From overleaf IN5400

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

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

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

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

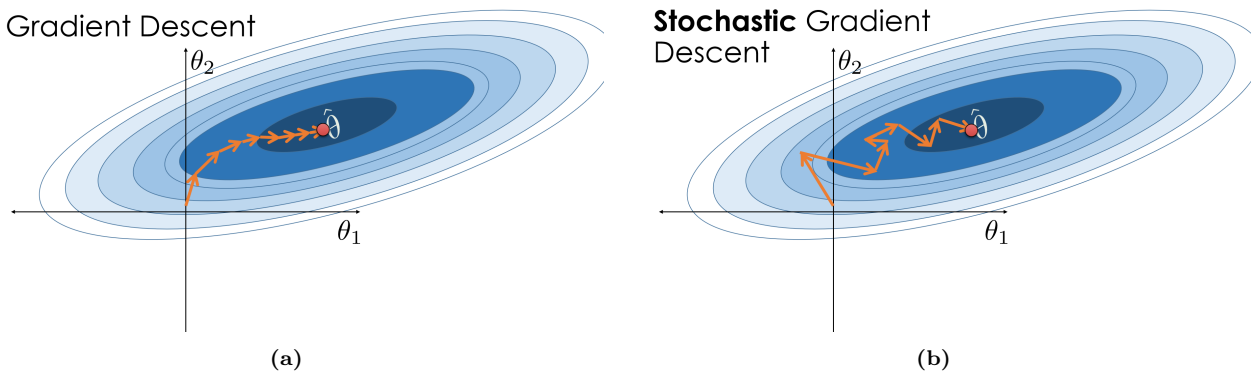
The cross-entropy loss can be derived from a maximum likelihood estimation SOURCE. Without going into details with the derivation we can convince ourselves that the error is minimized for the correct prediction and maximized for the worst prediction. When $y_i = 1$ we get the negative term $-\log(\hat{y}_i)$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

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

$$\theta_t = \theta_{t-1} - \eta\nabla_\theta L(\theta_t). \tag{1.1}$$

Gradient descent is analog to taking a step in parameter space in the direction that yields the biggest decrease in the loss. If we imagine a simplified case with only two parameters θ_1 and θ_2 we can think of these as directions

on a map and the loss being the terrain height. The gradient descent steps in the direction perpendicular to the contour lines shaped by loss function terrain as shown in Fig. 1.2a. Notice, however, that state-of-the-art models in general contain on the order of $10^6 - 10^9$ parameters [4] which poses some challenges for the visualization. The length of each step is proportional to the gradient norm and the learning rate η. There are three main flavors to the gradient descent: Batch, stochastic and mini-batch gradient descent. In *batch gradient descent* we simply calculate the gradient based on the entire dataset by averaging the contribution from each data point before updating the parameters. This gives the most robust estimate of the gradient and thus the most direct path through parameter space in terms of minimizing the loss function as indicated in Fig. 1.2a. However, for big datasets, this calculation can be computationally heavy as it must carry the entire dataset in memory at once. A solution to this issue is provided by *stochastic gradient descent* (SGD) which considers only one data point at a time. Each data point is chosen randomly and the parameters are updated based on the corresponding gradient. This leads to more frequent updates of the parameters and a more "noisy" path through parameter space as shown in Fig. 1.2b. Under some circumstances, this might compromise the precision. However, the presence of noise can actually increase the chances of avoiding local minima in parameter space The *mini-batch gradient descent* serves as a middle ground between the above methods by dividing the full dataset into a subset of mini-batches. Each parameter update is then based on the gradient within a mini-batch. By choosing a suitable batch size we get the robustness of the (full) batch gradient descent and the computational efficiency and resistance to local minima of the SGD method.



(a) **(b)**

Figure 1.2: TMP

1.1.1 Optimizers

The name *optimizers* covers a variety of gradient descent methods. In our study, we will use the ADAM (adaptive moment estimation) [5]. ADAM combines several "tricks in the book" which we will introduce in the following.

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

$$\theta_t = \theta_{t-1} - m_t, \qquad m_t = \alpha m_{t-1} + \eta \nabla_\theta L(\theta_t) \tag{1.2}$$

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

$$\begin{aligned} m_1 &= \alpha m_0 + \eta g_1 = \eta g_1 \\ m_2 &= \alpha m_1 + \eta g_2 = \alpha^1 \eta g_1 + \eta g_2 \\ m_3 &= \alpha m_2 + \eta g_3 = \alpha^2 \eta g_1 + \alpha \eta g_2 + \eta g_3 \\ &\vdots \\ m_t &= \eta \left(\sum_{k=1}^{t} \alpha^{t-k} g_k \right). \end{aligned} \tag{1.3}$$

Hence m_t is a weighted average of the gradients with an exponentially decreasing weight. This act as a memory of the previous gradients and aid to pass local minima and to some degree plateaus in the parameter space.

It also provides a general steadiness to the descent which counteracts the transition from batch to mini-batch gradient descent. A variation of momentum can be achieved with the introduction of the exponential moving average (EMA) which builds on the recursion

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

which is similar to that of momentum Eq. (1.3), but with the explicit weighting by $(1-\alpha)$. The second moment of the exponential moving average is utilized in the root mean square propagation method (RMSProp) which is motivated by the issue of passing long loss plateaus in the parameter space. Since the size of the updates are proportional to the norm of the gradient

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

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

$$\theta_t = \theta_{t-1} - \eta\frac{g_t}{\sqrt{\text{EMA}(||g_t||^2)} + \epsilon}, \tag{1.4}$$

where ϵ is simply a small number to avoid division by zero issues.

ADAM merges the idea of first order EMA for the momentum m_t, and the second order EMA v_t, as used in the root mean square propagation technique in Eq. (1.4)

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

Since these are initially set to zero ADAM introduces the scaling terms $(1-\beta_1^t)$ and $(1-\beta_2^t)$ to correct for a bias towards zero. The ADAM scheme is given [5]

$$\theta_{t+1} = \theta_t - \eta\frac{\hat{m}_t}{\sqrt{\hat{v}_t} + \epsilon}, \qquad \hat{m}_t = \frac{m_t}{1-\beta_1^t}, \qquad \hat{v}_t = \frac{v_t}{1-\beta_2^t}. \tag{1.5}$$

1.1.2 Weight decay

By adding a so-called *regularization* to the loss function we can penalize high magnitudes of the model parameters, usually intended for the model weights primarily. This is motivated by the idea of preventing overfitting during training. The most common way to do this is by the use of L2 regularization, adding the squared l^2 norm $||\theta||_2^2$, where $||\theta||_2 = \sqrt{\theta_1^2 + \theta_2^2 + \ldots}$, to the model. The loss and gradient then become

$$L_{l^2}(\theta) = L(\theta) + \frac{1}{2}\lambda||\theta||_2^2 \tag{1.6}$$

$$\nabla_\theta L_{l2}(\theta) = \nabla_\theta L(\theta) + \lambda\theta, \tag{1.7}$$

where $\lambda \in [0,1]$ is the weight decay parameter. The name *weight decay* relates to the fact that some practitioners only apply this penalty to the weights in the model, but we will include the biases as well (standard in PyTorch). Following the original gradient descent scheme Eq. (1.7) we get

$$\theta_{t+1} = \theta_t - \eta g_t - \eta\lambda\theta_t = \theta_t \underbrace{(1-\eta\lambda)}_{\text{weight decay}} - \eta g_t. \tag{1.8}$$

Thus we notice that choosing a high weight decay ($\lambda \to 1$) will downscale the model parameters while choosing a low weight decay ($\lambda \to 0$) yields the original gradient descent scheme. Note that we will use the weight decay principle in combination with ADAM. In Eq. (1.8) we have simply used the original gradient descent scheme Eq. (1.1) since this makes it easier to demonstrate the consequences of introducing the L2 regularization into the loss function Eq. (1.6).

1.1.3 Parameter distributions

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

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

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

Batch normalization is another technique that can also help reduce the issue of poor gradient flow. Furthermore, it can benefit by speeding up convergence and making the training process more stable SOURCE. In general, model parameters are modified throughout training meaning that the range of values coming from a previous layer will shift, even though the same training data is fed through the network repeatedly. By scaling the input for a given layer, for each mini-batch, we can mitigate this problem and make for a more standardized input range. This often results in a faster training convergence. For layer l we calculate the mean $\mu^{[l]}$ and variance $\sigma^{2[l]}$ across the layer with nodes $x_1^{[l]}, x_2^{[l]}, \ldots, x_d^{[l]}$ for each mini-batch of size m as

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

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

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

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

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

with trainable parameters γ and β. Comment about the reason for the final step.

1.1.4 Learning rate decay strategies

Until now we have assumed a constant learning rate, but many training variations use a changing learning rate beyond the adaptiveness included in the optimizers covered so far. Under some circumstances, it can be beneficial to start with a higher learning rate to speed up the initial part of training and then lower the learning rate for the final gradient descent [6]. One straightforward strategy is a step-wise learning rate decay where the learning rate is reduced by a factor $\gamma \in (0, 1)$ every K steps. A more smooth change can be achieved by for instance a polynomial decay $\eta_t = \eta_0/t^\alpha$ for $\alpha > 0$. More advanced approaches use multiple cycles of increasing and decreasing cycles. We will mainly concern ourselves with a one-cycle policy for which we start at an intermediate value, increase toward a maximum bound and then decrease toward a final lower learning rate bound. We do this by following a cosine function that is shifted and stretched to increase towards the first 30% of the training length and decrease toward the lower bound learning rate for the remaining epochs.

1.2 Convolutional Neural Network

Convolutional Neural Networks (CNNs) build upon many of the same concepts as introduced with the feed-forward neural network in Sec. 1.1. The difference lies in its specialization for a spatially correlated input, such as pixels in an image. In a dense neural network, every node is connected to each of the nodes from the previous layers which is not ideal for image recognition. For instance, if we want the model to recognize images of animals the dense network will be very sensitive to where that animal is placed within the frame. The CNN is motivated by the idea of capturing spatial relations in the input, but without being sensitive to the relative placement within the input, i.e. being translational invariant. This is achieved by having a so-called *kernel* or *filter* which slides over the images[1] as it processes the input. The overall flow of data for a typical convolutional network is illustrated in Fig. 1.3. A convolutional layer contains multiple kernels, each consisting of a set of trainable weights and a bias. Each kernel will produce a separate output channel to the resulting *feature map* layer. The kernel has a 2D spatial size, specific to the model architecture, and a depth that matches the number of input channels to the layer. For instance, a typical RGB will have three channels, while the number of channels usually increases for each layer in the model. The kernel lines up with the image and calculates the feature map output as a dot product between the weights in the kernel and the aligning subset of the input. This is done for each input channel and summed up with the addition of a bias as illustrated in Fig. 1.4b. The kernel then slides over by a step size given by the *stride* model parameter and repeats the calculation. Choosing a stride of 2 or higher results in a reduction of the output spatial size. If we want to preserve the spatial size we must keep a stride of one and additionally apply *padding* to the input images, such that we can achieve one kernel position for each input "pixel". The spatial size of the feature map is given as

$$N_d^{[l+1]} = \left\lfloor \frac{N_d^{[l]} - F_d + 2P}{S} + 1 \right\rfloor, \tag{1.9}$$

for padding P, stride S, spatial size of the kernel filter F_d, spatial size of input $N_d^{[l]}$, for dimension $d = x, y$ and layer l. The *down-sampling* is often done through a pooling layer. A pooling layer is reminiscent of a kernel, but instead of calculating the output as a dot product, it utilizes the mean (mean pooling) or the max value (max pooling) of the values within its scope. For instance, by using a max pooling of size 2×2 and stride two we essentially half the dimensions of the image as dictated by Eq. (1.9). CNNs will often use repeating series of convolution (applying a kernel), pooling and then an activation function. Most architectures aim to slowly down-sample the spatial input while increasing the number of channels throughout the model layers.

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

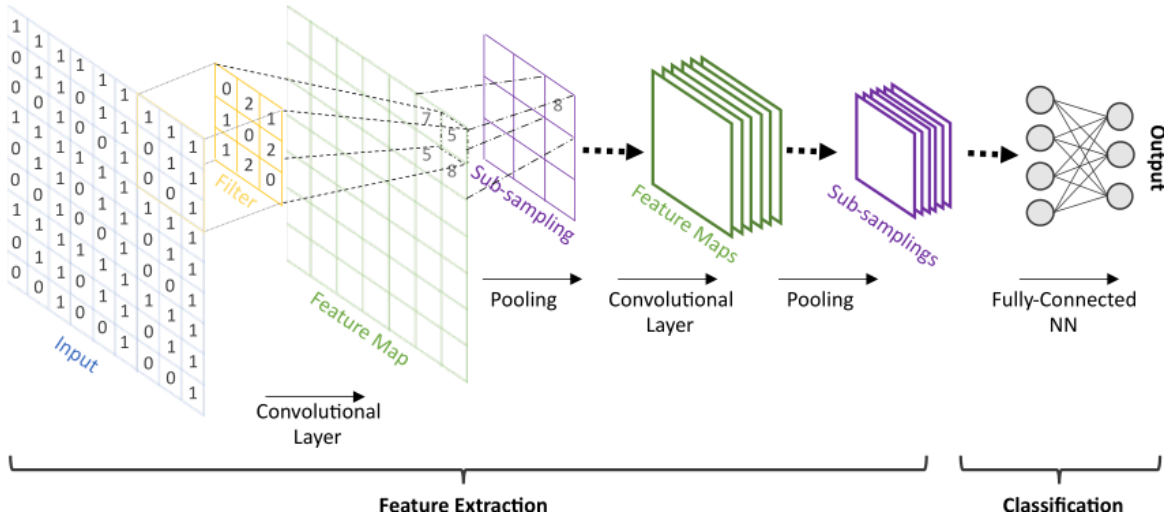
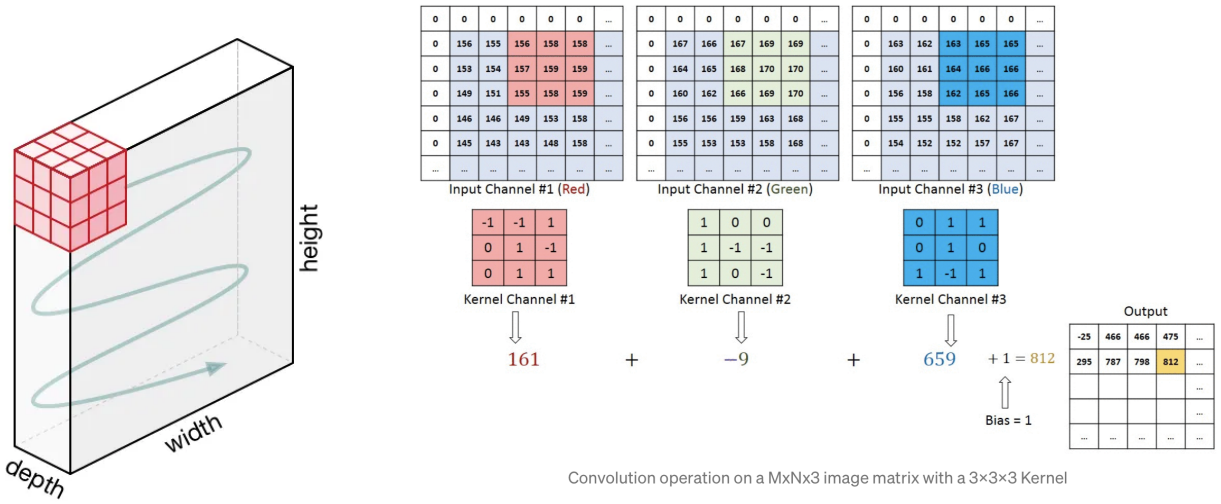


Figure 1.3: TMP



(a) **(b)**

Figure 1.4: TMP

For a CNN, we often consider the *receptive field*. The receptive field relates to the spatial size of the input that affects a given node in the feature map at a given layer of the model. Often this term is used in consideration of the output nodes. In Fig. 1.5 the receptive field is illustrated for a 1D representation of a CNN with repetitive use of a kernel of width 1 and stride 1. Going from the output and backward, we see that the output layers are connected to two nodes in the previous layer. Each of these nodes is connected to two nodes in the layer before that, however with one of them being the same due to the stride of 1. By back-tracking to the input we see that this corresponds to a receptive field of $D = 5$. By increasing the filter size and the stride the 2D receptive field will grow a lot faster than shown in this simple 1D example. For a receptive field D_d, with respect to the spatial dimension d, a spatial size of the filter F_d, stride S_l (from layer $l-1$ to l) we have

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

with $D_0 = 1$ and $l = 0$ as the input layer. Note that by convention, the product of zero elements is 1, such that for the first layer, the product is 1. The receptive field is important in understanding the connectivity

in the model. The model output will be completely independent of the inputs and feature maps outside the receptive field. Furthermore, we differentiate between the theoretical receptive field and the effective receptive field. The effective receptive field will have a Gaussian distribution within the theoretical receptive field because the nodes in the center of the receptive field will have more connections leading to the output, as seen in Fig. 1.5. Thus, in practice, the effective receptive field will be smaller than the theoretical. Implementations like dilated convolutions, which make the filter expand in circumference and skipping positions within the filter, can be used to further increase the effective field. The receptive field is perhaps not that relevant... Should I remove it?

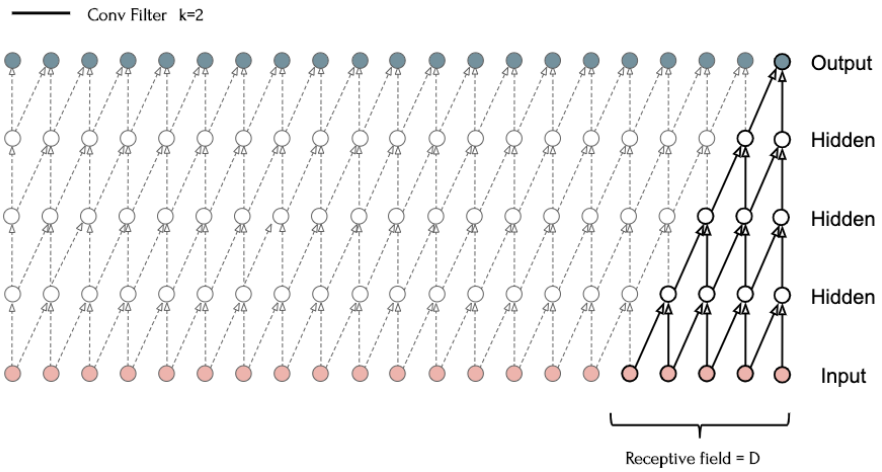


Figure 1.5: TMP

On a final note regarding the CNN we point out that convolution is often used in combination with a dense network, or *fully connected*, at the end. We can then think of the convolution part to handle the translation from a spatial input to some internal features. For the animal detection network, we would perhaps think of features such as the number of legs, size, color and so on. In practice, the network will not create easily interpreted features for the processing of the fully connected layer to see. We discuss one approach for the interpreting of the model internals in Sec. 1.5

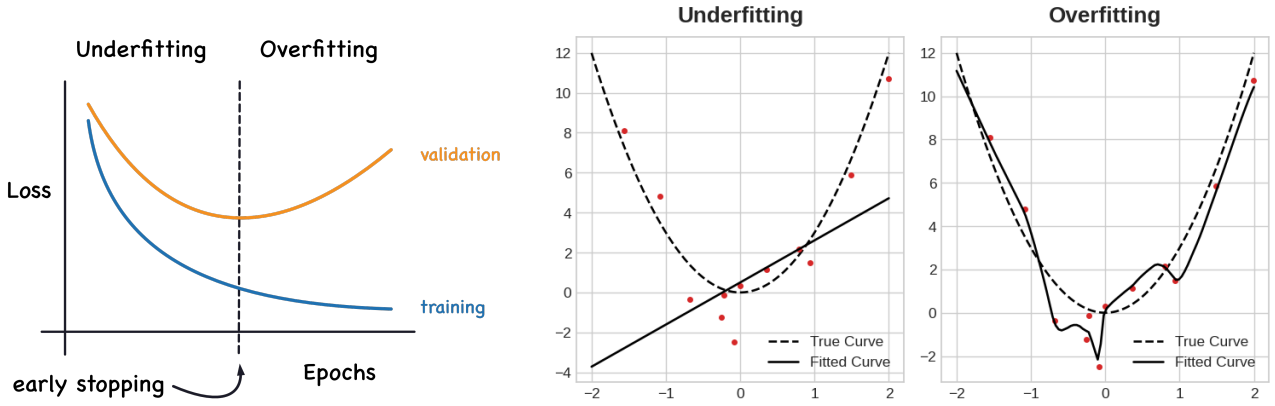
1.2.1 Training, validation and test data

So far, we have simply considered the concept of *training data* as a means to update the model parameters. Yet, we want to evaluate the model performance as it improves. The problem arises immediately from the fact that a complex model can fit about any function. More precisely, it has been proven that a deep convolutional neural network is universal (follows the universal approximation theorem), meaning that can approximate any continuous function to an arbitrary accuracy when the depth of the network is large enough [7]. Thus for a complex model, it is just a matter of time before the model eventually finds a good approximation for the training data. However, we want the model to learn general trends and not to "memorize" all the data points which are known as *overfitting*. While the predictions for the training data can grow arbitrarily good in most cases, the performance on unseen data within the domain will yield poor performance in the case of overfitting. The common way to address this issue is by putting aside a subset of the data, the so-called *validation* data, which we use to validate the model performance during and after training. By keeping this *validation* set separate from the training data we can get a more reliable performance estimate for the model. Random partitioning is crucial for ensuring an equal distribution of data across both sets. To strike a balance between the quality of training and validation, a commonly used partitioning ratio is usually around 20:80 in favor of the training set. Other techniques exist which aim to optimize the data used for sparse data situations, like cross-validation and bootstrap right?, but we will not consider such methods for this thesis. A third data set that is often forgotten is the *test* set. While the validation set should be kept unseen from the model training, the test set should be kept unseen from the model developer. As we choose the model architecture and hyper-parameters. We define a hyper-parameter as a variable to be set prior to the actual application of the learning algorithm, one that is not selected by the learning algorithm itself [8]. This includes parameters such as learning rate, momentum and

weight decay, but not the weights and biases as these are updated by the learning algorithm. When adjusting the hyper-parameters we will use the performance on the validation set as a guiding metric. Hence, our choices can eventually lead to a higher level of overfitting through the hyper-parameter choices. Hence, we should denote a test set for the final evaluation of our model which has not been considered before the end. Formally, this is the only reliable performance metric for the model.

1.3 Overfitting and underfitting

Underfitting and overfitting represent a crucial balance going on when training a model. This concept is highly related to the model complexity and the chosen hyper-parameters. The textbook visualization of underfitting and overfitting is shown in Fig. 1.6a. As we begin to train or model both the training and validation loss is decreasing. At some point, the model will start to pick up, not only the general data trends but also specific trends in the training data. This marks the transition into the overfitting regime where the validation loss will increase again, even though the training loss is steadily declining. *Early stopping* can be utilized to detect this transition and stop the training in an attempt to hit the sweet spot between under- and overfitting. We will use a variation of this which is to store the best model based on validation performance. For this approach, we let the training finish but only keep the model corresponding to the best validation score. In principle, we can "get lucky" and find the model settings at a state that is specially overfitted for the validation set, but we consider this highly unlikely when having a reasonable amount of data and a complex model with many model parameters. The underfitting and overfitting phenomena can also be thought of as a function of the complexity and not just training time. For a certain amount of epochs a simple model will yield underfitting and an overly complex model will yield overfitting, and this can be expected to follow a similar qualitative trend as in Fig. 1.6a with the substitution for *model complexity* on the x-axis. Fig. 1.6b visualizes the concept of underfitting and overfitting in terms of the complexity regarding the fitting of a second-order polynomial. We see how a simple linear function will make a crude approximation for the true curve. An overly complex model will pick up the noise in the data and miss the general trend. However, the problem is that we do not know the true curve. If we did, we would not need machine learning to approximate it in the first place. Without having additional insight into the governing source of the data the overfitting case seems to produce the most confident fit for all we know.



(a) (b)

Figure 1.6: TMP

1.4 Hypertuning

The training of a machine learning model revolves around tuning the model parameters such as weights and biases. However, as mentioned already, a handful of *hyper-parameters* remains for us to decide. First of all, we need to choose an architecture for the model. This includes high-level considerations, for instance, whether to use a neural network or a convolutional network, but also lower-level considerations, such as the depth and the width of the model, i.e. how many layers and how many nodes/channels. In addition, we have to define and consider the loss function and the optimizer which come with hyper-parameters such as learning rate, momentum

and weight decay. This extensive list of choices makes the designing of a functional machine learning procedure more complicated than simply hitting "run" for the learning algorithm. As N. Smith [6] puts it: "Setting the hyper-parameters remains a black art that requires years of experience to acquire". In the following, we will review a general approach for choosing the learning rate, momentum and weight decay hyper-parameters based on the findings of [6]. The traditional approach is to perform a *grid search*, trying out different combinations of hyper-parameters different training sessions, but this might rather quickly become computationally expensive and ineffective. In addition, hyper-parameters will depend on the training data, the model architecture and not at least each other, which make it difficult to narrow down the choice one by one. N. Smith points to the fact that validation loss can be examined early on for clues of either underfitting or overfitting.

The learning rate is often regarded as the most important hyper-parameter to tune [8]. Typical values are in the range $[10^{-6}, 1]$. Instead of simply running a grid search, we can perform a so-called *learning rate range test* (LR range test). One then specifies the minimum and maximum learning rate boundaries and a learning rate step size. A minimum and maximum bound of 10^{-7} to 10 will most likely cover an appropriate range, but the test will reveal this immediately. The idea is then to vary the learning rate throughout the given range in small steps during a short pre-training. We will vary the learning rate for each iteration, i.e. each parameter update following a mini-batch, and thus we can run this test for a few epochs, or even a single one, depending on the number of mini-batches. The learning rate can be varied in a linear increasing or decreasing manner which is found to produce similar results [9]. We chose the linear increasing version for simplicity. For small learning rates, the model will converge slowly. As the learning rate approaches an appropriate value the convergence will accelerate which we see as a drop in the validation loss. Eventually, the convergence will stop and the validation loss will pass a minimum for which it will begin to diverge. This general behavior can be understood for the simplified 1D example of finding the minima of a second-order polynomial as shown in Fig. 1.7. Small learning rates will step in the right direction, but for very small values this will result in a slow convergence. If the learning rate becomes too large, we will effectively step past the minimum. Each following step will overshoot the minimum more and more (the step is proportional to the gradient os the loss) leading to a diverging trend. The point of divergence can be used as an upper bound for the learning rates when considering a cyclic learning rate scheme. The steepest decline of the validation loss can be used as an estimate for the best constant learning rate choice [6].

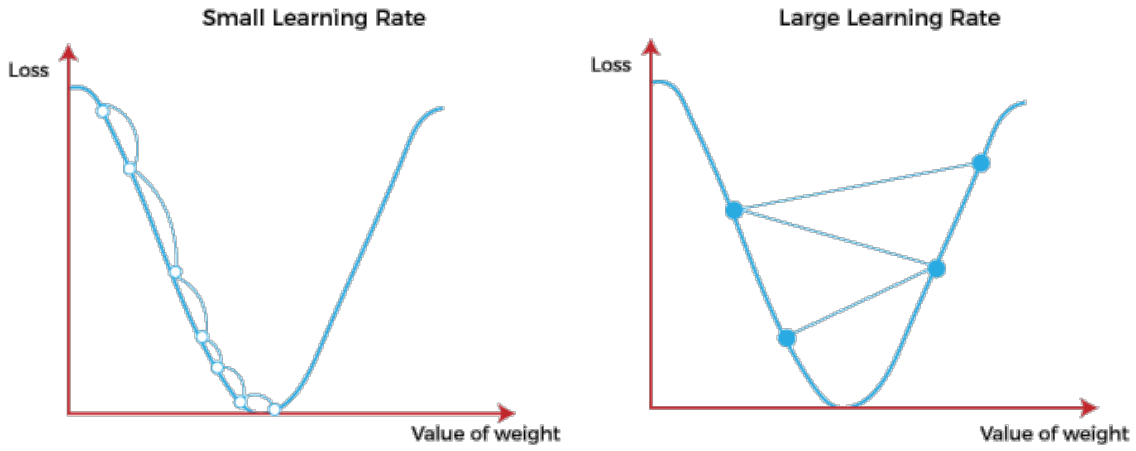


Figure 1.7: TMP

Next, we consider the choice of momentum. Momentum and learning rates are found to affect each other considerably. From the gradient descent scheme with momentum Eq. (1.2) we see that the momentum parameter α and the learning rate η have a similar effect on the parameter update

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

since m_t is a moving average of the gradient g_t as well. Like the learning rate, we want to set the momentum value as high as possible without causing instabilities in the training. However, it is found that these values are somewhat inversely related. Choosing a high learning rate should be coupled with a lower momentum and

vice versa. N. Smith [6] reports that a momentum range test is not useful to find the right momentum. Instead, he suggests doing a few short runs with different values of momentum, such as 0.99, 0.97, 0.95, and 0.9, to determine a suitable choice. By including momentum in the LR range test we can balance the learning rate accordingly for such test. Moreover, for a cyclic learning rate scheme he suggests using a cycling momentum scheme reversed with respect to the learning rate. When the learning rate increase toward the upper bound the learning rate should decrease toward the lower bound and vice versa. Choosing a lower momentum of 0.80–0.85 often gives similar stabile results [6].

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

1.5 Prediction explanation

On a final note, we present a simple method for providing some insight into the prediction from a convolutional neural network. The high complexity of deep learning models limits our ability to gain insight into the decision-making process behind a prediction beyond the input data. This is known as the *black box* problem. A lot of effort is currently being developed for making more transparent models, like decision trees with interpretable rules, and numerical tools for unpacking the inner workings of the model. We will consider a gradient based method called *Grad-CAM* [10] which aim to highlight some of the important features from the input image. The algorithm is based on the idea use the gradients for a certain feature map with respect to the loss.

First, we forward propagate the input through the model and decide on a feature map of interest. We then calculate the gradients for the feature map with respect to the loss of a certain target output. For a classification task, one would often choose the predicted class, the class with the highest score, as the target output. The gradients can then be used as an estimate of which part of the feature maps is most important for the prediction. a ReLU activation layer is then applied to keep only the positive contributions. Since the convolutional layers preserve spatial information we can rescale the heatmap provided by the feature map gradient to make an input-sized heatmap allowing for an overlaid visualization of the on the input image. This provides a visual clue of which part of the image the prediction is most strongly based on. We can do this for different depths of the model and even combine the results for multiple layers. Fig. 1.8 show an exemplary use, where the Grad-CAM analysis reveals the difference between a biased and unbiased prediction model for the task of predicting professions. The biased model shows to be considering the person more than the actual objective clues given by relevant equipment and work-related clothing

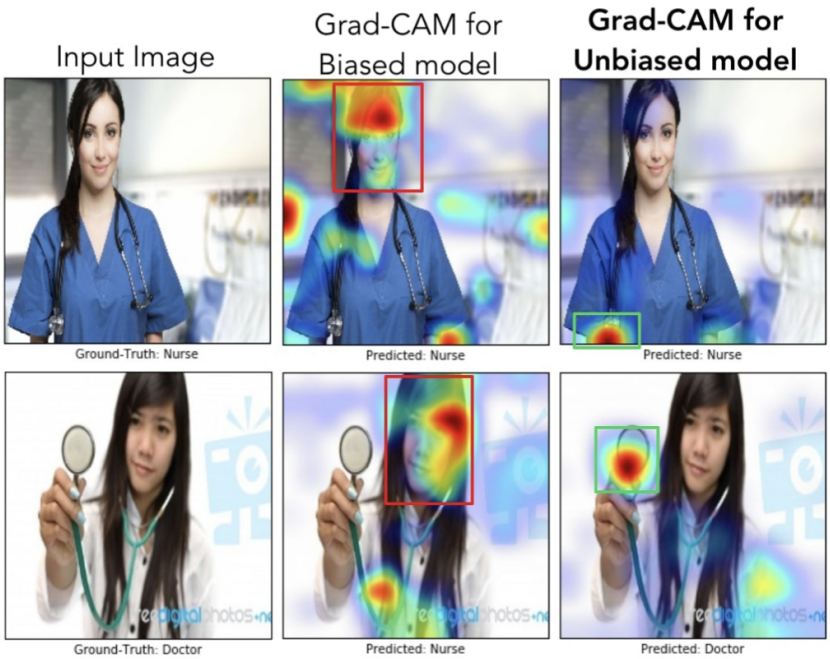


Fig. 8: In the first row, we can see that even though both models made the right decision, the biased model (model1) was looking at the face of the person to decide if the person was a nurse, whereas the unbiased model was looking at the short sleeves to make the decision. For the example image in the second row, the biased model made the wrong prediction (misclassifying a doctor as a nurse) by looking at the face and the hairstyle, whereas the unbiased model made the right prediction looking at the white coat, and the stethoscope.

Figure 1.8: TMP [10]

1.6 Accelerated search using genetic algorithm

For the scope of finding new Kirigami designs which exhibit certain frictional properties, we are interested in utilizing a trained machine-learning model for further exploration. This reverses the design process as one has to find the right input to achieve a certain output. A possible strategy is to explore a range of inputs and use the model predictions as a guiding metric. One approach to this is the genetic algorithm (GA) which is inspired by biological evolution and mimics the Darwin theory of the survival of the fittest. [11]. GA is a population-based algorithm for which the basic elements are chromosome representation, fitness selection and biological-inspired operators. The chromosomes represent the genes for each individual in the population and typically take the form of a binary string. Each position within the chromosome is called a *locus* and has two possible values (0 or 1). A fitness function is defined to assign a score for all chromosomes based on some optimization objective. This plays a role for the biologically inspired operators for which the main ones are selection, mutation and crossover. Selection is the process of selecting chromosomes based on their fitness score for further processing. In mutation, some of the loci within a chromosome are flipped and in crossover, chromosomes are merged to create offspring. GA has been implemented in many areas such as the traveling salesman problem [12], function optimization [13], adaptive agents in stock markets [14] and airport scheduling [15]. Wang et al. [16] note that a general drawback is a need for expertise when choosing parameters that match specific applications. They propose an accelerated genetic algorithm based on a Markov chain transition probability matrix to perform a guided search that reduces the number of parameter choices one has to make. The following introduction of this method is thus based on [16].

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

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

with $N' = N/2$ from [16]. We assign a row mutation probability $a_i(t)$ meaning that the probability for a mutation will increase towards the lower fitness scores. For the considerations of mutation with respect to each locus in the columns of $A_{ij}(t)$, we define the count of 0's and 1's as $C_0(j)$ and $C_1(j)$ respectively. These are normalized as

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

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

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

These are normalized

$$n_0(j,t+1) = \frac{C_0'(j)}{C_0'(j) + C_1'(j)}, \quad n_1(j,t+1) = \frac{C_1'(j)}{C_0'(j) + C_1'(j)}. \tag{1.10}$$

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

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

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

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

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

$$P_{10}(j,t) = \frac{n_0(j,t+1) - P_{00}(j,t) n_0(j,t)}{n_1(j,t)} \tag{1.11}$$
$$P_{01}(j,t) = 1 - P_{00}(j,t) \tag{1.12}$$
$$P_{11}(j,t) = 1 - P_{10}(j,t) \tag{1.13}$$

The remaining part is to define $P_{00}(j,t)$. We adopt the choice from [16] and start from $P_{00}(j, t=0) = 0.5$ and choose $P_{00}(j,t) = n_0(j,t)$ for the following generations. Thus for a locus $A_{ij}(t)$ we mutate it, changing the binary value, by the probability

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

In summary, each generation update involves the following steps.

1. For generation t calculate the fitness score $f_i(t)$ of each chromosome i and sort the population matrix $A_{ij}(t)$ row-wise according to a descending score.
2. From a defined ranking scheme $r_i(t)$ set the chromosome mutation probability to $a_i(t) = r_i(t)$ and the weighting of each row $W_i(t) = 1 - r_i(t)$.
3. Calculate the target states Eq. (1.10) and the transition probabilities using Eq. (1.11) to (1.13) and $P_{00}(j, t = 0) = 0.5$, $P_{00}(j, t > 0) = n_0(j, t > 0)$.
4. Mutate (flip) each locus $A_{ij}(t)$ by the p_{ij} given by **??**.

Notice that this algorithm treats every locus as an independent gene. Thus, we do not incorporate any effects from spatial dependencies in the Kirigami pattern matrix.

1.6.0.1 Repair function

A numerical scheme for repairing the Kirigami matrix to correspond to a non-detached sheet. This was implemented in order to get candidates that are not immediately marked as a rupture. But it might be better placed in the Random walk section even though we did not make this algorithm before creating the random walk configurations. .

Part II

Simulations

Chapter 2

Summary

The work presented in this thesis covers several topics (find a better opening line?). We have created an MD simulation which enabled us to study the frictional behavior of a graphene sheet sliding on a Si substrate. In addition, we have created a numerical framework for creating Kirigami design patterns and introducing these into the friction simulations. This was used to study the effects of the out-of-plane buckling induced by a selected pair of Kirigami designs in relation to a non-cut sheet under the influence of strain. Further, we have created a dataset of various Kirigami designs for the scope of investigating the possibilities with Kirigami design. We have investigated the possibility to use machine learning on this dataset and attempted an accelerated search. Finally we look into the prospects of achieving a negative friction coefficient for a system with coupled load and stretch. In this chapter we will summarize the findings and draw some final conclusions. We will also provide some topics for further research.

2.1 Summary and conclusions

2.1.1 Design MD simulations

We have designed an MD simulation for the examination of friction for a graphene sliding on a substrate. Some of the key features for the numerical procedure were that we managed the sheet through pull blocks in the ends. We could then apply load and stretch the sheet without acting directly on the inner parts. Say something about parameter dependencies from the Pilot study.

2.1.2 Design Kirigami framework

We have desgined a numerical framework for creating Kirigami designs. By defining an indexing system for the hexagonal lattice structure we were able to define the Kirigami designs as 2D matrix for numerical implementation. We digitalized two different macroscale designs, which we named the *Tetrahedron* and *Honeycomb* pattern respectively, that successfully produced out-of-plane buckling when stretched. Through a numerical framework we could create an ensemble of perturbed variations which gave approximately 135k configurations for the Tetrahedron pattern and 2025k patterns for the size of the sheet used in our study. When considering the possibility to translate the patterns this gave roughly a factor 100 more of unique pertubations. We also created a framework for creating Kirigami designs through a random walk. This was further controlled by introducing features such as bias, avoidance of existing cuts, preference to keeping a direction and procedures to repair the sheet for simulation purposes. The capabilities of the numerical framework for generating Kirigami designs was far larger than the capabilities for producing MD designs within the time constraint of this thesis. Thus we believe that this contains the possibility to benefit more extended studies and for the creation of a larger dataset.

2.1.3 Control friction using Kirigami

We have investigated the friction behavior of the non-cut sheet and a selected Tetrahedron and Honeycomb pattern under various stretch and load. The non-cut sheet did not exhibit significant out-of-plane buckling as opposed to the Tetrahedron and Honeycomb pattern. This is even when considering that the non-cut sheet had

a yield strain of 0.35 while the Tetrahedron had a lower yield strain of 0.21 and the Honeycomb a considerable larger one at 1.27 based on a stretch in vacuum. The out-of-plane buckling resulted in a signifciant reduction of the contact area as the sheet were stretched, towards a minimum of X for the Tetrahedron and y for the Honeycomb pattern. However, this disagreed with the asperity theory hypothesis of a decreasing friciton with decreasing contact area. We found that the strain-induced buckling was initially (at low relative strain) assocated with an increase in friction. Moreover, the friction-strain curve produced a no-linear behaviour which was not compatible with the approximately monotonic decreasing contact area as strain were increased. This is shown in **??**. This led us to the conclusion that the contact area cannot be attributed a dominant mechanism for friction throughout the straining of the studied Kirigami sheets. In general we found a non-existing relationship between friction and load considering the uncertainties in the simulation. This is best attributed to the superlubric state of the graphene sheet on the substrate. The slope of the friction-load curves were not significantly affected by the straining of the Kirigami sheet and thus we conclude that the load effect on friction is neglible cimpared to the strain effects.

2.1.4 Capture trends with ML

From the dataset we found some correlations in the data. That is, we found a positive correlation between the mean friction force and strain (0.77) and porosity (0.60) and negative correlation to contact area (-0.67). These trends aligns with initial study showing that the Kirigami effects is associated to a decreasing contact area and a non-zero porosity (atoms need to be removed to produce a Kirigami sheet). By the use of machine learning we found that the trends in the data could generally be captured to a R_2 score above 0.97 for the validation data. However, the subset of configurations exhibiting the most significant effects from Kirigami and stretching provided a bigger challenge. Say something about hyperparameter search. By choosing our hyperparameters for a selected subset of challenging configurations we got a final network with a selected set score of $R^2 = 0.89$ at the worst for the Tetrahedron pattern. However these configurations was partly included in the training data and thus these numbers should be taken carefully. By creating a training set based on best candidates in an accelerated search using random walk configurations the network failed completely. This can to some extent be attributed to overfitting of the model but we belive that this is more deeply anchored in the insufficient distribution in the dataset not matching the created test set. It produced a reasonble ranking of the properties of interest within the dataset at least.

2.1.5 Accelerated search

Using the machine learning model we performed two types of accelerated search. One by going generating an extended dataset of Tetrahedron, Honeycomb and Random walk patterns and evaluating the results using the model. Another was based on the genetic algorithm were we pertubated the candidates in order to optimize for the max drop friction peroperty. The generative accelerated search method did produce some new candidates, but it is unsure how reliable this is due to the insufficienty in the machine learning model. We concluded that the minimization of friction was not applicable. The model revealed a sensitivity to edge effect which is likely due to a model insufficienty, but this serves as an interesting topic for future research. For the gentic algorithm search it did mainly not find any new candidates and we believe that the simplicity of the genetic algorithm was not immediately suited for creating new designs with the necessary spatial correlations presented

2.1.6 Negative friction coefficient

By enforcing a coupling between load and stretch, mimicking a nanomachine attached to the sheet, we investigated the load curves arising from load the Tetrahedron and Honeycomb pattern from the pilot study. The non-linear trend observed for increasing strain carried our to this simulation setup and produce a highly non-linear friction-load curve. This demonstrated a negative friction coefficient say something about the values.

2.2 Outlook / Perspective

Topics for further studies.

- How is this behavior effected by scaling?

- How does the distirbution of normal load effect the Kirigami friction behavior?
- Things to vary: load range, scan directions, adhesive forces, longer relaxation time, different potential (AIREBO)
- Investigate if the contact area is effecting the friciton non-linear by turning of friction force for atoms corresopnding to those that lift of from the sheet during the out-of-plane buckling.
- Investigations of commensurability effects.
- Study dependency of translation of the patterns as suggested by the ML results.
- Investigate effects from pull blocks...
- Investigate effects from the thermostat since the top and bottom edges was shown interest by the model prediction.

Appendices

Appendix A

Appendix A

Appendix A

Appendix B

Appendix B

Appendix C

Bibliography

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

[2]J. Lederer, *Activation functions in artificial neural networks: a systematic overview*, 2021.

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

[4]N. C. Thompson, K. Greenewald, K. Lee, and G. F. Manso, *The computational limits of deep learning*, 2022.

[5]D. P. Kingma and J. Ba, *Adam: a method for stochastic optimization*, 2017.

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

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

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

[9]L. N. Smith, *Cyclical learning rates for training neural networks*, 2017.

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

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

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

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

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

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

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