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

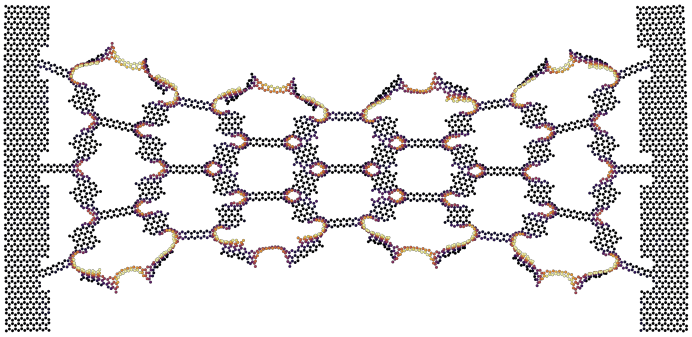
UNIVERSITY OF OSLO

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, 26

EMA Exponetial Moving Average. 6

GA Genetic Algorithm. 14

MD Molecular Dynamics. 3, 19, 26, 32, 38

ML Machine Learning. 34, 35, 36, 37, 38

MSE Mean Squared Error. 4, 27

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.

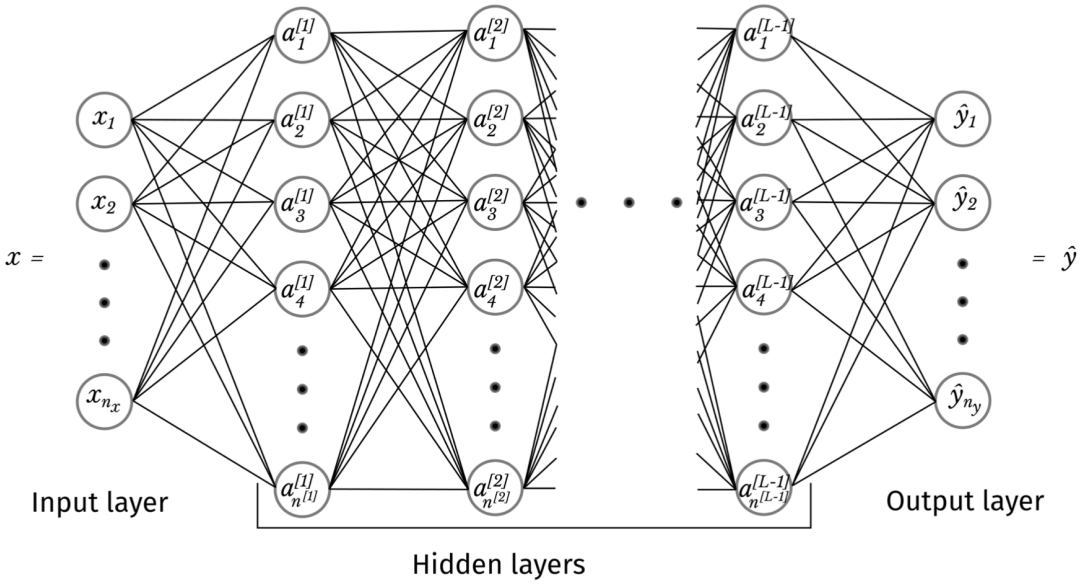


Figure 1.1: From overleaf IN5400

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)

$$L_{\text{MSE}} = \frac{1}{N_y}\sum_{i=1}^{N}(y_i - \hat{y}_i)^2. \tag{1.1}$$

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)

$$L_{\text{BSE}} = -\sum_{i=1}^{n}\Big[y_i \log(\hat{y}_i) + (1-y_i)\log(1-\hat{y}_i)\Big] = \sum_{i=1}^{n}\begin{cases} -\log(\hat{y}_i), & y_i = 1 \\ -\log(1-\hat{y}_i), & y_i = 0. \end{cases} \tag{1.2}$$

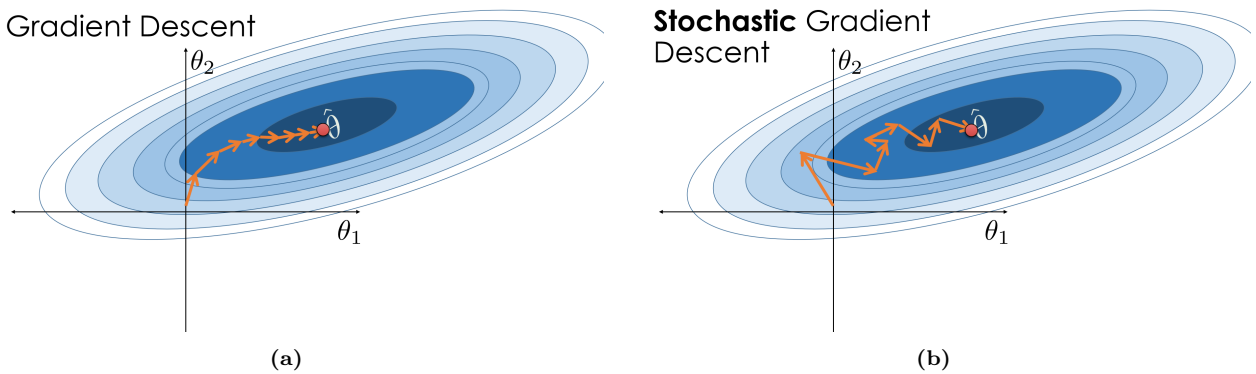
The cross-entropy loss can be derived from a maximum likelihood estimation SOURCE. Without going into details with the derivation we can convince ourselves that the error is minimized for the correct prediction and maximized for the worst prediction. When $y_i = 1$ we get the negative term $-\log(\hat{y}_i)$ 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

$$\theta_t = \theta_{t-1} - \eta\nabla_\theta L(\theta_t). \tag{1.3}$$

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

$$\theta_t = \theta_{t-1} - m_t, \qquad m_t = \alpha m_{t-1} + \eta \nabla_\theta L(\theta_t) \tag{1.4}$$

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

$$\begin{aligned} m_1 &= \alpha m_0 + \eta g_1 = \eta g_1 \\ m_2 &= \alpha m_1 + \eta g_2 = \alpha^1 \eta g_1 + \eta g_2 \\ m_3 &= \alpha m_2 + \eta g_3 = \alpha^2 \eta g_1 + \alpha \eta g_2 + \eta g_3 \\ &\vdots \\ m_t &= \eta \left(\sum_{k=1}^{t} \alpha^{t-k} g_k \right). \end{aligned} \tag{1.5}$$

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

It also provides a general steadiness to the 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. (1.5), but with the explicit weighting by $(1-\alpha)$. The second moment of the exponential moving average is utilized in the root mean square propagation method (RMSProp) which is motivated by the issue of passing long loss plateaus in the parameter space. Since the size of the updates are 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

$$\theta_t = \theta_{t-1} - \eta\frac{g_t}{\sqrt{\text{EMA}(||g_t||^2)} + \epsilon}, \tag{1.6}$$

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. (1.6)

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

Since 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]

$$\theta_{t+1} = \theta_t - \eta\frac{\hat{m}_t}{\sqrt{\hat{v}_t} + \epsilon}, \qquad \hat{m}_t = \frac{m_t}{1-\beta_1^t}, \qquad \hat{v}_t = \frac{v_t}{1-\beta_2^t}. \tag{1.7}$$

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

$$L_{l^2}(\theta) = L(\theta) + \frac{1}{2}\lambda||\theta||_2^2 \tag{1.8}$$

$$\nabla_\theta L_{l2}(\theta) = \nabla_\theta L(\theta) + \lambda\theta, \tag{1.9}$$

where $\lambda \in [0,1]$ is the weight decay parameter. The name *weight decay* relates to the fact that some practitioners only apply this penalty to the weights in the model, but we will include the biases as well (standard in PyTorch). Following the original gradient descent scheme Eq. (1.9) we get

$$\theta_{t+1} = \theta_t - \eta g_t - \eta\lambda\theta_t = \theta_t \underbrace{(1-\eta\lambda)}_{\text{weight decay}} - \eta g_t. \tag{1.10}$$

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. (1.10) we have simply used the original gradient descent scheme Eq. (1.3) since this makes it easier to demonstrate the consequences of introducing the L2 regularization into the loss function Eq. (1.8).

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

$$N_d^{[l+1]} = \left\lfloor \frac{N_d^{[l]} - F_d + 2P}{S} + 1 \right\rfloor, \tag{1.11}$$

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. (1.11). 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.

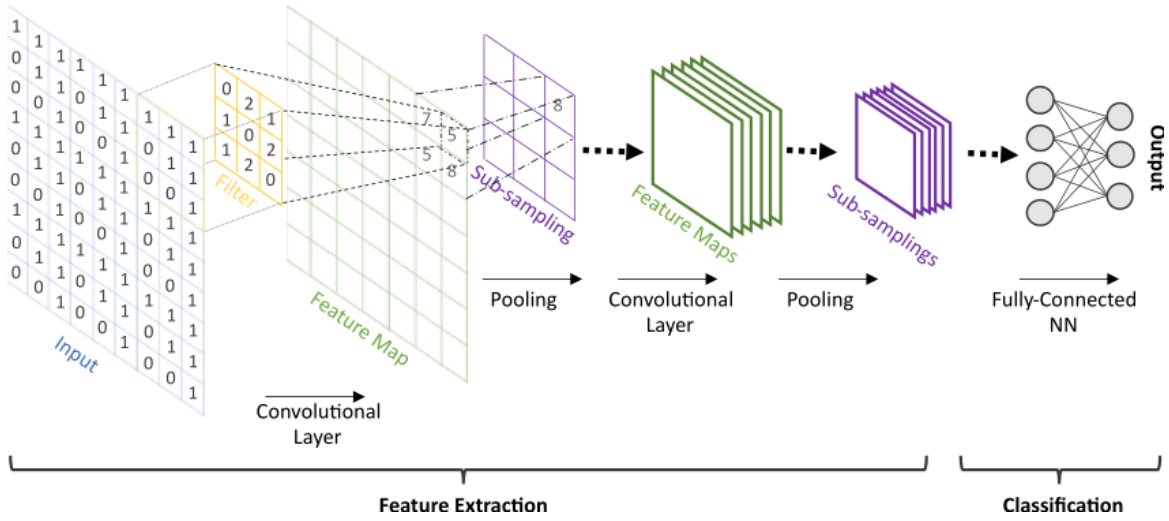
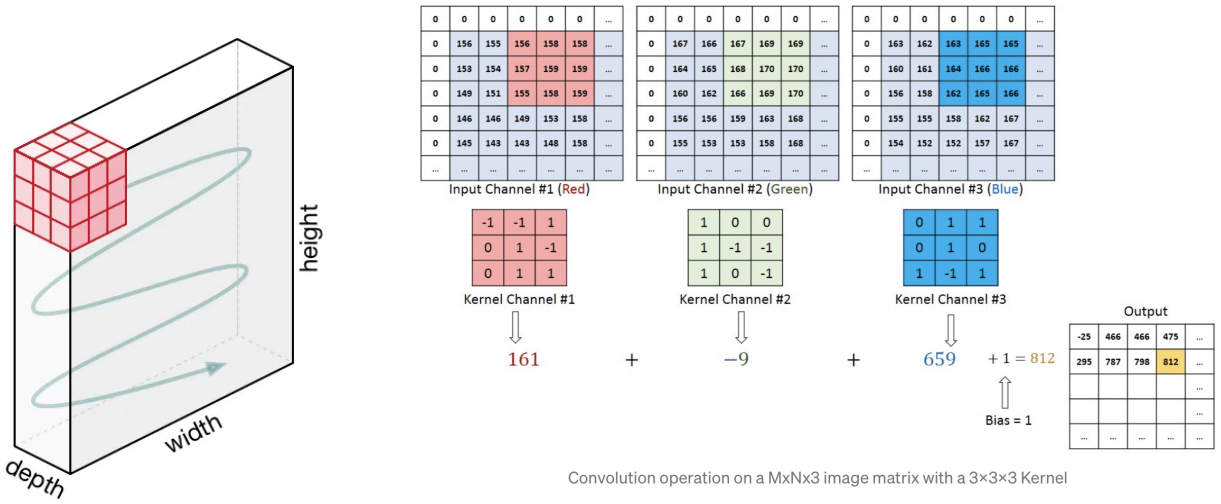


Figure 1.3: TMP



(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?

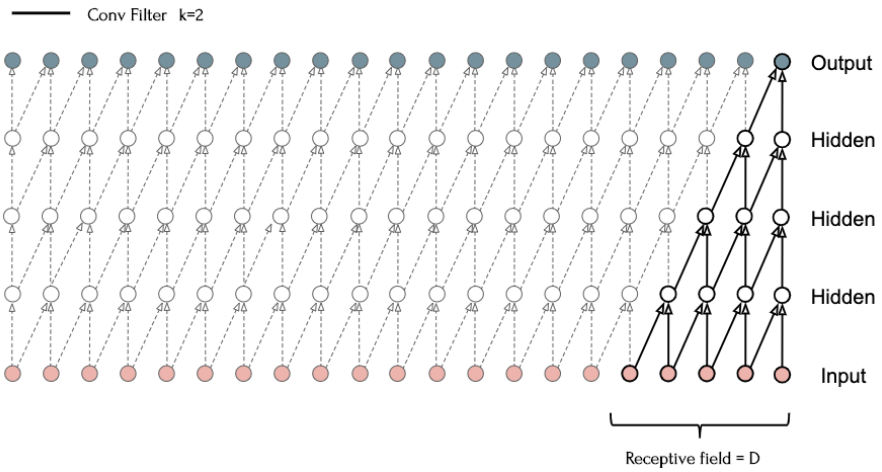


Figure 1.5: TMP

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

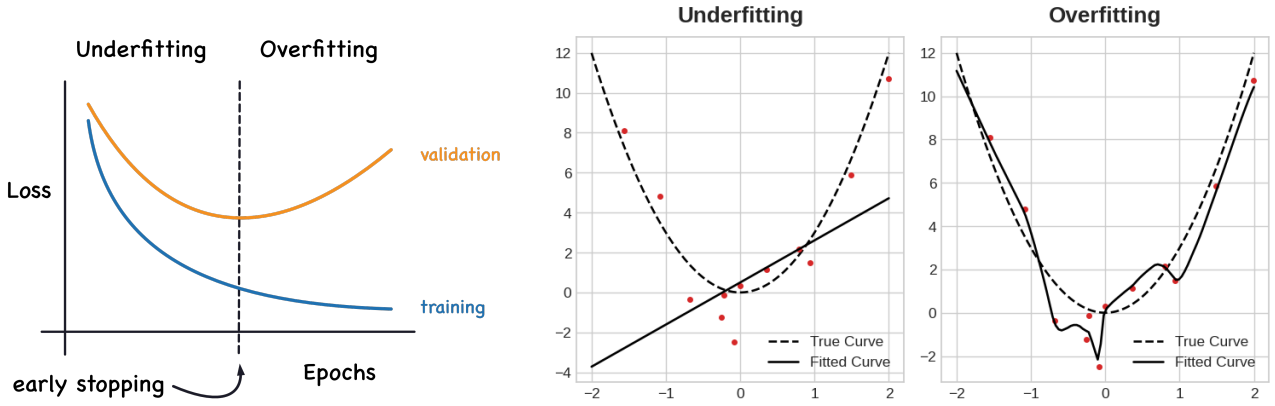
1.2.1 Training, validation and test data

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].

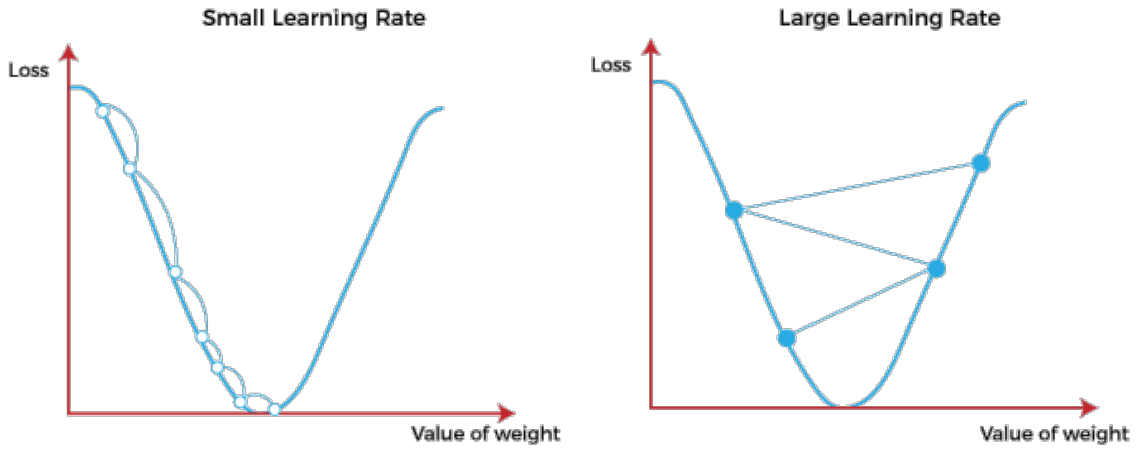


Figure 1.7: TMP

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. (1.4) we see that the momentum parameter α and the learning rate η have a similar effect on the parameter update

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

since m_t is a moving average of the gradient g_t as well. Like the learning rate, we want to set the momentum value as high as possible without causing instabilities in the training. However, 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

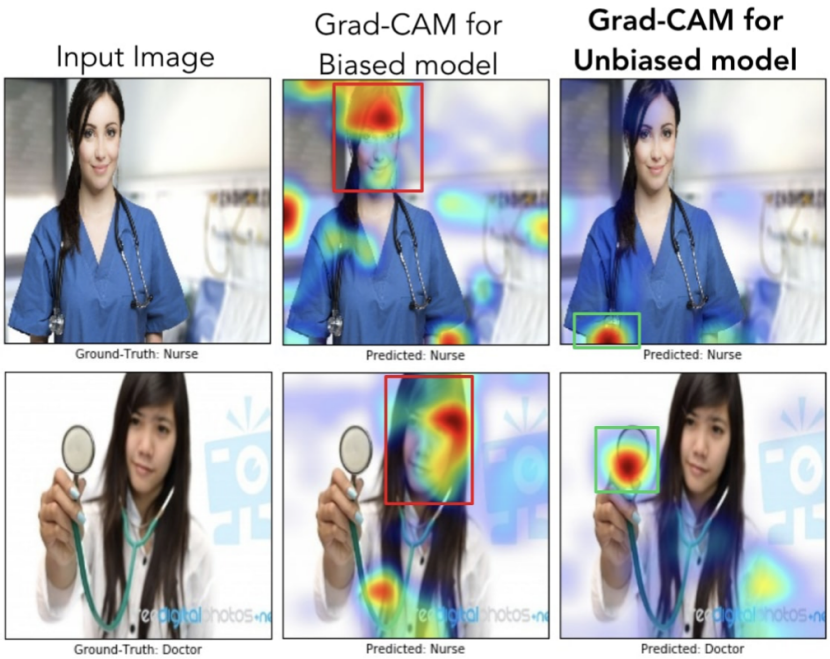


Fig. 8: In the first row, we can see that even though both models made the right decision, the biased model (model1) was looking at the face of the person to decide if the person was a nurse, whereas the unbiased model was looking at the short sleeves to make the decision. For the example image in the 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

$$n_0(j,t+1) = \frac{C_0'(j)}{C_0'(j) + C_1'(j)}, \quad n_1(j,t+1) = \frac{C_1'(j)}{C_0'(j) + C_1'(j)}. \tag{1.12}$$

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 $P_{00}(j,t)$

$$P_{10}(j,t) = \frac{n_0(j,t+1) - P_{00}(j,t) n_0(j,t)}{n_1(j,t)} \tag{1.13}$$
$$P_{01}(j,t) = 1 - P_{00}(j,t) \tag{1.14}$$
$$P_{11}(j,t) = 1 - P_{10}(j,t) \tag{1.15}$$

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

$$p_{ij}(t) = \begin{cases} a_i(t) P_{01}(t), & A_{ij}(t) = 0 \\ a_i(t) P_{10}(t), & A_{ij}(t) = 1 \end{cases} \tag{1.16}$$

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. (1.12) and the transition probabilities using Eq. (1.13) to (1.15) 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 Eq. (1.16).

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

Kirigami configuration exploration

Building upon the discoveries of the Pilot Study **??**, we will further explore the impact of Kirigami designs on strain-dependent friction. Our focus is primarily to optimize the friction force and friction coefficient towards their maximum or minimum values. To achieve this goal, we will utilize MD simulations to generate an extended dataset that encompasses a wider range of Kirigami designs. This is motivated by the aim of gaining gain a broader understanding of the friction-strain relationship. We will then leverage this dataset to explore the potential of employing machine learning for predicting friction behavior based on Kirigami design, strain, and load. Finally, we plan to utilize the developed machine learning model for an accelerated search.

2.1 Generating the dataset

We aim to create a dataset that contains an extended series of Kirigami design configurations based on the pattern generation methods developed in **??** for which we will vary the strain and load for each configuration. For each configuration, we sample 15 pseudo uniform strain values (see **??**) between zero and the rupture strain according to the rupture test. Since the normal force did not prove to be dominant in the friction description we only sample 3 values per configuration, uniformly sampled in the range $[0.1, 10]$ nN. In total, this gives $3 \times 15 = 45$ data points for each configuration. For the remaining parameters, we use the values presented in the pilot study (see **??**). We are mainly concerned with the mean friction and whether the sheet ruptures during the simulation. However, we also include the maximum friction, the relative contact, the rupture strain (from the rupture test) and the porosity (void fraction) in the dataset. We generate 68 configurations of the Tetrahedron pattern type, 45 of the Honeycomb type and 100 of the Random walk type. For the Tetrahedron and Honeycomb patterns, we choose a random reference position that results in translation of the patterns. A summary of the dataset is given in Table 2.1 while all configurations are shown visually in **??**. The Tetrahedron and Honeycomb parameters are chosen to provide additional variations of the configurations evaluated in **??** which exhibited interesting properties. The Random walk configurations are chosen with the aim of introducing as much variety as possible within our Random walk framework. Notice that not all submitted data points "make it" to the final dataset. This is due to a small bug in the data generation procedure[2].

[2]The issue arises from the fact that the rupture point in the rupture test does not completely match the rupture point in the following simulations. After performing the rupture test the simulation is restarted with a new substrate size, corresponding to the measured rupture strain limit, but also with a new random velocity and thermostat initialization. The sheet is then strained and checkpoints of the simulation state (LAMMPS restart files) are stored for each of the targeted strain samples. However, if the rupture point arrives earlier than suggested by the rupture test, due to randomness from the initialization, some of the planned strain samples do not get a corresponding checkpoint file. Thus, these data points are not included in the dataset even though they ideally should have been noted as a rupture event. This could quite have been mitigated by a rewrite of the code, but it was first discovered after the dataset had been created. We notice, however, that the dataset still contains 11.57 % rupture events which provide a reasonable amount of rupture events to incorporate in the machine learning model

Table 2.1: Summary of the number of generated data points in the dataset. Due to slight deviations in the rupture strain and the specific numerical procedure not all submitted simulations "make it" to the final dataset. Notice that the Tetrahedon $(7, 5, 2)$ and Honeycomb $(2, 2, 1, 5)$ from the pilot study are rerun as a part of the Tetrahedon and the Honeycomb datasets separately. In the latter datasets, the reference point for the pattern is randomized and thus these configurations are not fully identical. This is the reason for the ambiguousness in the total sum.

Type	Configurations	Submitted data points	Final data points	Ruptures
Pilot study	3	270	261	25 (9.58 %)
Tetrahedon	68	3060	3015	391 (12.97 %)
Honeycomb	45	2025	1983	80 (4.03 %)
Random walk	100	4500	4401	622 (14.13 %)
Total	214 (216)	9855	9660	1118 (11.57 %)

2.2 Data analysis

In order to gain insight into the correlations in the data we calculate the correlation coefficients between all variable combinations. More specifically, we calculate the Pearson product-moment correlation coefficient which is defined, between data set X and Y, as

$$\mathrm{corr}(X,Y) = \frac{\mathrm{Cov}(X,Y)}{\sigma_X \sigma_Y} = \frac{\langle (X-\mu_X)(Y-\mu_Y)\rangle}{\sigma_X \sigma_Y} \in [-1,1],$$

where $\mathrm{Cov}(X,Y)$ is the covariance, μ the mean value and σ the standard deviation. The correlation coefficients range from a perfect negative correlation (-1) through no correlation (0) to a perfect positive correlation (1). The correlation coefficients are shown in Fig. 2.1.

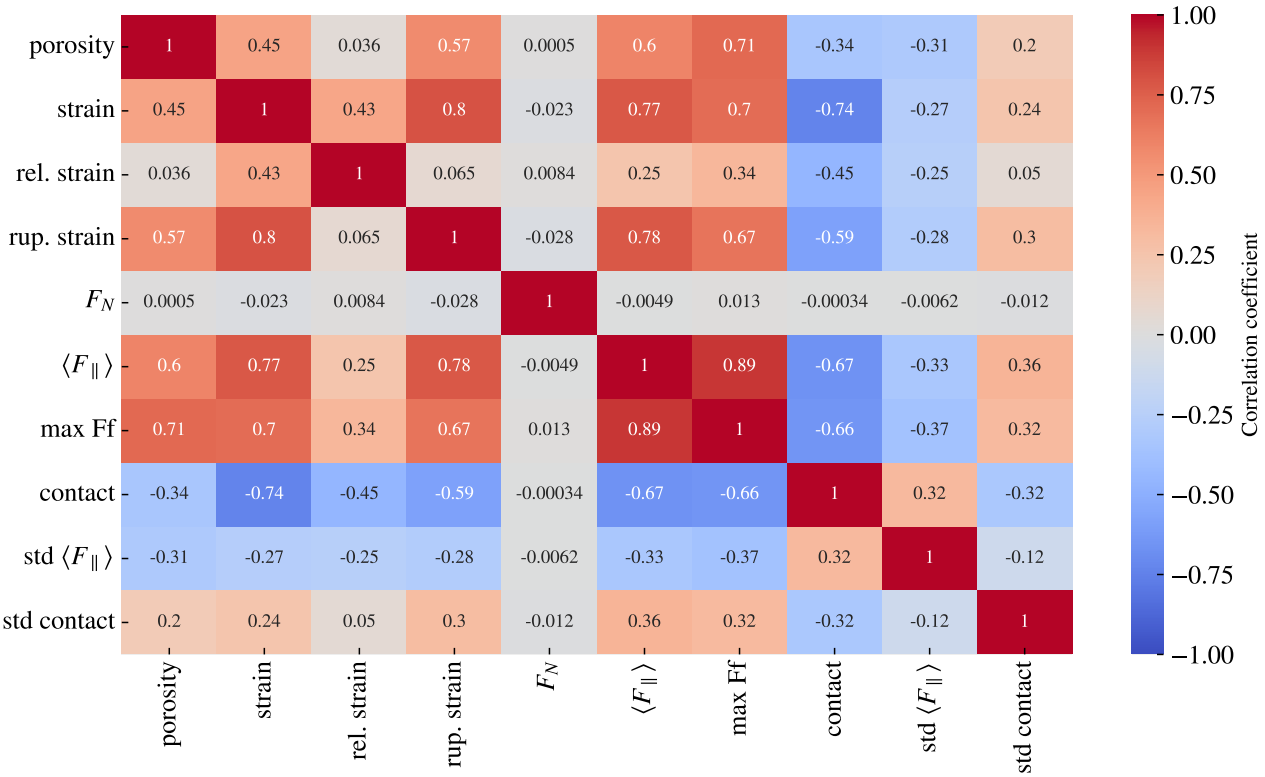


Figure 2.1: Pearson product-moment correlation coefficients for the full datset (see Table 2.1).

From Fig. 2.1 we especially notice that the mean friction force $\langle F_{\parallel} \rangle$ has a significant positive correlation with strain (0.77) and porosity (0.60). However, the relative strain, the strain scaled by the rupture strain, has a weaker correlation of only 0.25. This indicates that the correlation might be associated with the flexibility of the configurations since these can be taken to higher absolute values of strain. This is further supported by the fact that the mean friction and the rupture strain are also strongly positively correlated (0.78). From Fig. 2.1 we also observe that the contact is negatively correlated with the mean friction (-0.67) and the strain value (-0.74). This is generally consistent with the trend observed in the pilot study in **??** where the increasing strain was correlated with a decreasing contact and mainly increasing mean friction. However, we must note that the correlation coefficient is a measure of the strength and slope of a forced linear fit on the data. Since we have observed a non-linear trend between friction and strain (**??**) we should not expect any near 100 % correlations. Additionally, we also notice that all correlations to normal load are rather low, which aligns well with the findings in the pilot study.

Fig. 2.2 shows a visualization of the data (excluding the pilot study configurations) for chosen variable pairs on the axes. This provides a visual clue on some of the correlations and provides a qualitative feeling for the diversity in various planes of the feature space that we eventually will base our machine learning model on. We notice that the honeycomb pattern is spanning a significantly larger range of strain, contact and mean friction.

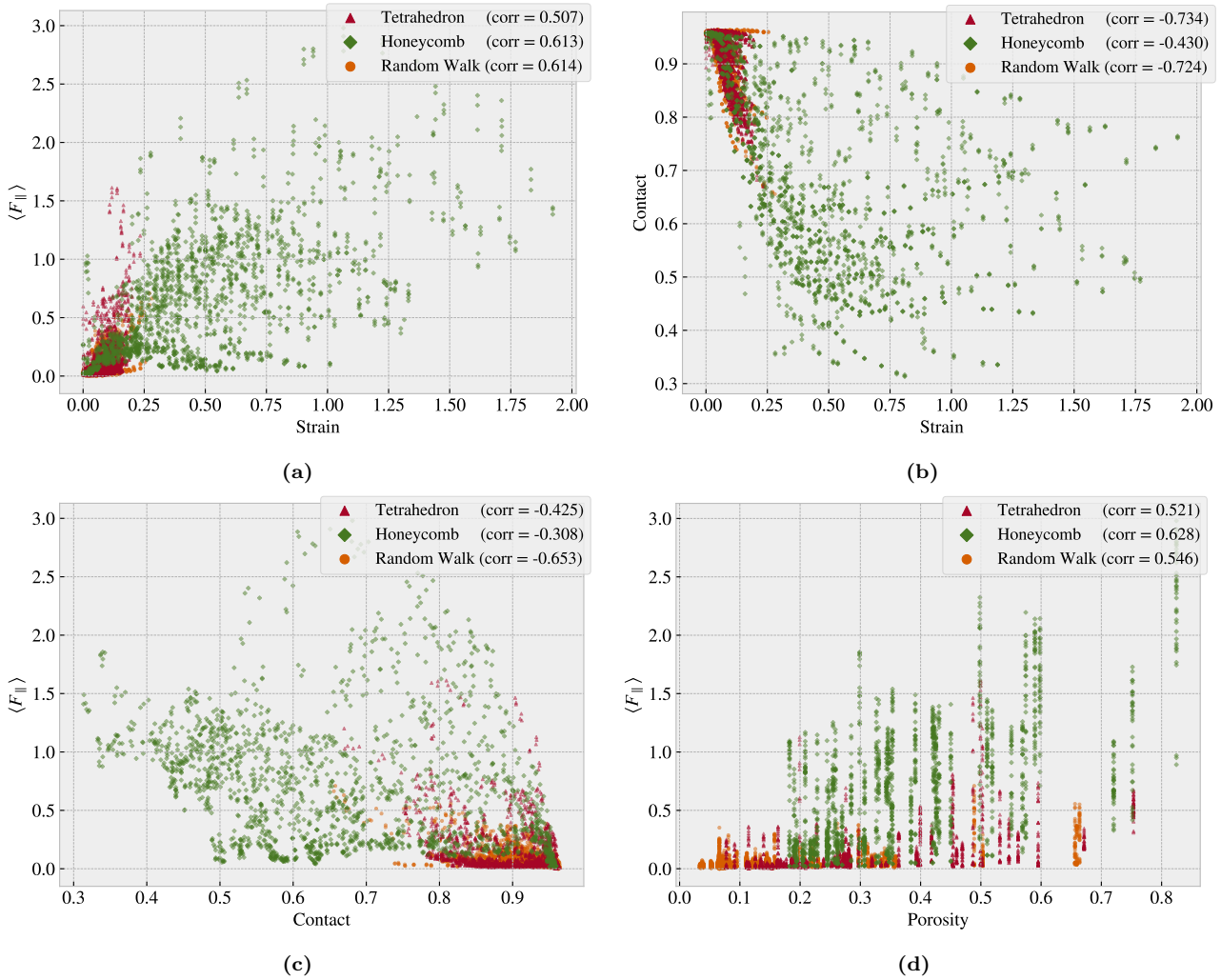


Figure 2.2: Scatter plot of the data sets Tetrahedron, Honeycomb and Random Walk (excluding the pilot study) for various variable combinations in order to visualize some chosen correlations of interest and distributions in the data

2.3 Properties of interest

In the pilot study (**??**) we found promising results for the idea of achieving a negative friction coefficient under the assumption of a system with coupled normal force and strain. Hence, we will consider this as a main property of interest for our further exploration. However, it is not obvious how one should rigorously quantify this. The friction coefficient is by our definition (**??**) given as the slope of the friction F_f vs. normal force F_N curve. Hence, for two data points $\{(F_{N,1}, F_{f,1}), (F_{N,2}, F_{f,2})\}$, $F_{N,1} < F_{N,2}$ we can evaluate the associated friction coefficient $\mu_{1,2}$ as

$$\mu_{1,2} = \frac{F_{f,2} - F_{f,1}}{F_{N,2} - F_{N,1}} = \frac{\Delta F_f}{\Delta F_N}.$$

In the pilot study, it became apparent that the effects of friction under the change of load is negligible in comparison to the effects related to strain ε. Thus, by working under the assumption $F(F_N, \varepsilon) \sim F(\varepsilon)$ and a coupling $F_N \propto R \cdot \varepsilon$ with linear coupling ratio R we get

$$\mu_{1,2}(\varepsilon_1, \varepsilon_2) = \frac{\Delta F_f(\varepsilon_1, \varepsilon_2)}{R(\varepsilon_2 - \varepsilon_1)} \propto \frac{\Delta F_f(\varepsilon_1, \varepsilon_2)}{\Delta \varepsilon}. \tag{2.1}$$

With this reasoning, we can in practice substitute load F_N for strain ε in the expression for the friction coefficient of our coupled system. This justifies the search for a negative slope on the friction-strain curve since this can be related to a negative friction coefficient in our proposed coupled system. The remaining question is then how to evaluate the strength of this property. By definition, the minimum (most negative) slope value would give the lowest friction coefficient. However, two data points with a small $\Delta\varepsilon$, corresponding to a small denominator in Eq. (2.1), would potentially lead to a huge negative slope value without any significant decrease in friction. Hence, we choose to consider the drop in friction with increasing strain as a better metric. Numerically we compute this by locating the local maxima on the friction-strain curve and then evaluating the difference to the succeeding local minima. The biggest difference corresponds to the *max drop* which serves as our indicator for a negative friction coefficient. In this evaluation, we do not guarantee a monotonic decrease of friction in the range of the biggest drop, but when searching among multiple configurations this is considered a decent strategy to highlight configurations of interest worthy of further investigation. In addition to the biggest drop in friction, we also consider the minimum, $\min F_{\text{fric}}$, and maximum, $\max F_{\text{fric}}$, friction along with the maximum difference, $\max \Delta f_{\text{fric}} = \max F_{\text{fric}} - \min F_{\text{fric}}$. The extrema of these four properties for each of the categories: Tetrahedron, Honeycomb, Random walk and Pilot study, are summarized in Table 2.2. The corresponding strain profiles and configurations are shown in Fig. 2.3 to 2.6 (excluding the pilot study). The strain profiles for the full dataset are shown in appendix **??**.

From the property comparison in Table 2.2 we find that both the Tetrahedron and Honeycomb subsets contain improved candidates for each of the property scores in comparison to the Tetrahedron $(7, 5, 1)$ and Honeycomb $(2, 2, 1, 5)$ examined in the pilot study. Overall, the Honeycomb pattern type is still resulting in the highest scores for the maximum properties while the minimum friction is still achieved by the non-cut sheet. This latter observation reveals that our dataset does not provide any indication that friction can readily be reduced for a Kirigami sheet under strain. However, the improvement in the remaining properties indicates that the dataset contains the necessary information to provide a direction for further optimization of the maximum properties. Considering the Random walk we find that the max property scores are generally lower than that of the Tetrahedron and Honeycomb structures. However, since these are found to be on a comparable order of magnitude we argue that contain relevant information for populating configuration space with respect to machine learning training. The Random walk patterns also contribute with some immediate insight into the structures that we can associate with each of the properties of interest due to its increased diversity in comparison to the other patterns.

For the $\min F_{\text{fric}}$ top candidates (Fig. 2.3) we find that the Random walk candidate has a rather cut density (low porosity) and with vertical cuts. Since these cuts run parallel to the stretching direction one can hypothesize that this minimizes the induced buckling effect which agrees with the constant level of contact. For the minimum candidate for the Tetrahedron pattern, we also observe a low decrease in contact, and in both these cases this corresponds with a relatively flat friction-strain curve. When considering the remaining friction-strain curve throughout Fig. 2.3 to 2.6 we find a rising friction-strain curve which is accompanied by a decreasing relative contact. When looking at the Random walk 96 pattern, which is the top candidate for both the $\max F_{\text{fric}}$ (Fig. 2.4) and $\max \Delta f_{\text{fric}}$ (Fig. 2.5) property, we find a rather porous configuration with mainly horizontal-orientated cuts.

This has some structural reminiscence with the Honeycomb pattern. Finally, the best random walk candidate for the max drop category, Random walk 01, did not produce a big drop. We find that the contact area is decreasing with strain but not as strongly as seen for the other configurations. The configuration contains some slanted cuts which might be reminiscent of parts of the Tetrahedron pattern.

Table 2.2: Evaluation of the properties of interest for our dataset.

Tetrahedron	Configuration	Strain	Value [nN]	Hon. $(2, 2, 1, 5)$ [nN]
$\min F_{\text{fric}}$	$(3, 9, 4)$	0.0296	0.0067	0.0262
$\max F_{\text{fric}}$	$(5, 3, 1)$	0.1391	1.5875	0.8891
$\max \Delta F_{\text{fric}}$	$(5, 3, 1)$	$[0.0239, 0.1391]$	1.5529	0.8603
max drop	$(5, 3, 1)$	$[0.1391, 0.1999]$	0.8841	0.5098

Honeycomb	Configuration	Strain	Value [nN]	Tetra. $(7, 5, 1)$ [nN]
$\min F_{\text{fric}}$	$(2, 5, 1, 1)$	0.0267	0.0177	0.0623
$\max F_{\text{fric}}$	$(2, 1, 1, 1)$	1.0654	2.8903	1.5948
$\max \Delta F_{\text{fric}}$	$(2, 1, 5, 3)$	$[0.0856, 1.4760]$	2.0234	1.5325
max drop	$(2, 3, 3, 3)$	$[0.5410, 1.0100]$	1.2785	0.9674

Random walk	Configuration	Strain	Value [nN]
$\min F_{\text{fric}}$	12	0.0562	0.0024
$\max F_{\text{fric}}$	96	0.2375	0.5758
$\max \Delta F_{\text{fric}}$	96	$[0.0364, 0.2375]$	0.5448
max drop	01	$[0.0592, 0.1127]$	0.1818

Pilot study	Configuration	Strain	Value [nN]
$\min F_{\text{fric}}$	No cut	0.2552	0.0012
$\max F_{\text{fric}}$	Honeycomb	0.7279	1.5948
$\max \Delta F_{\text{fric}}$	Honeycomb	0.7279	1.5325
max drop	Honeycomb	$[0.7279, 1.0463]$	0.9674

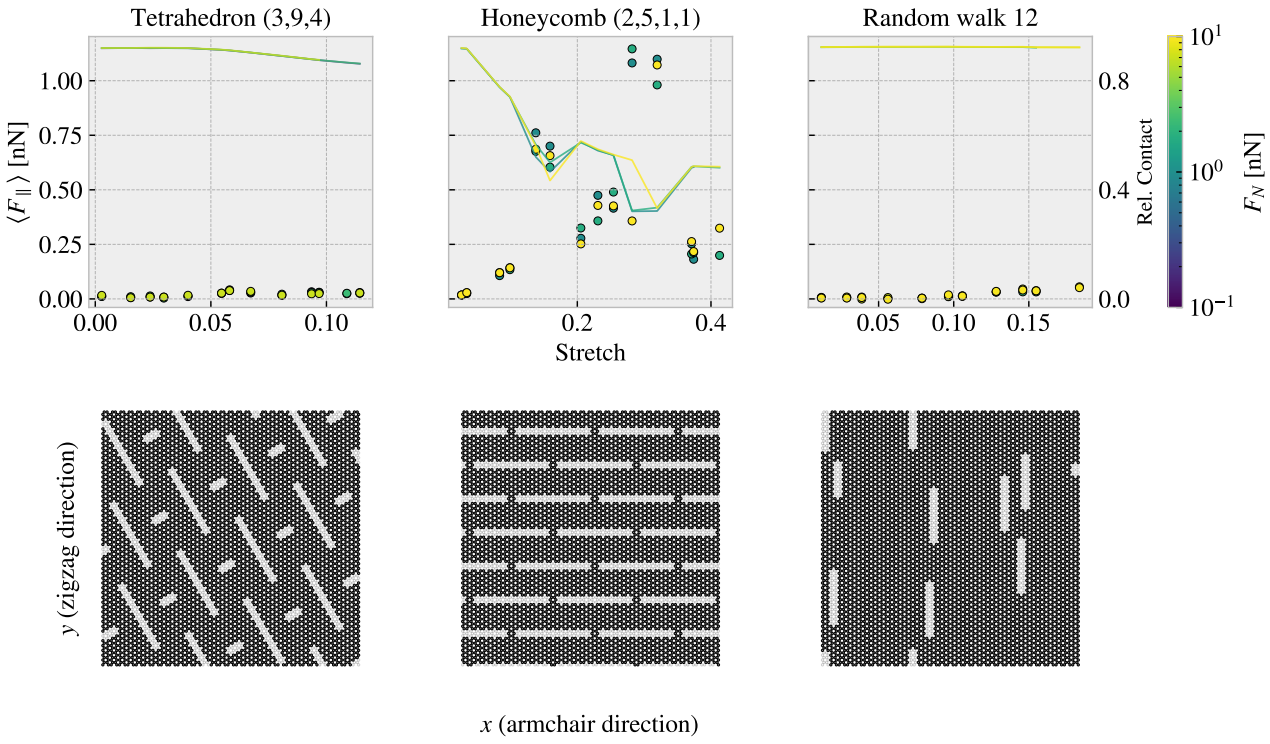


Figure 2.3: Minimum friction: Configurations corresponding to the minimum friction.

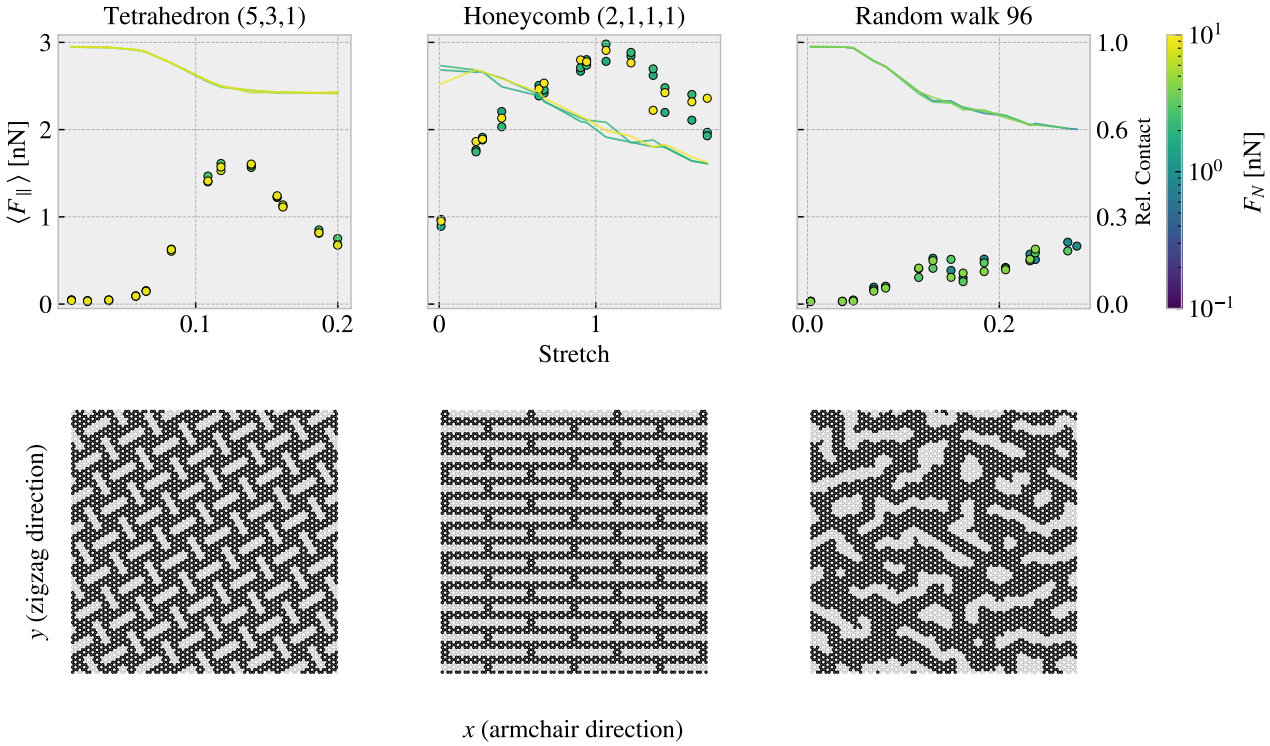


Figure 2.4: Maximum friction: Configurations corresponding to the maximum friction.

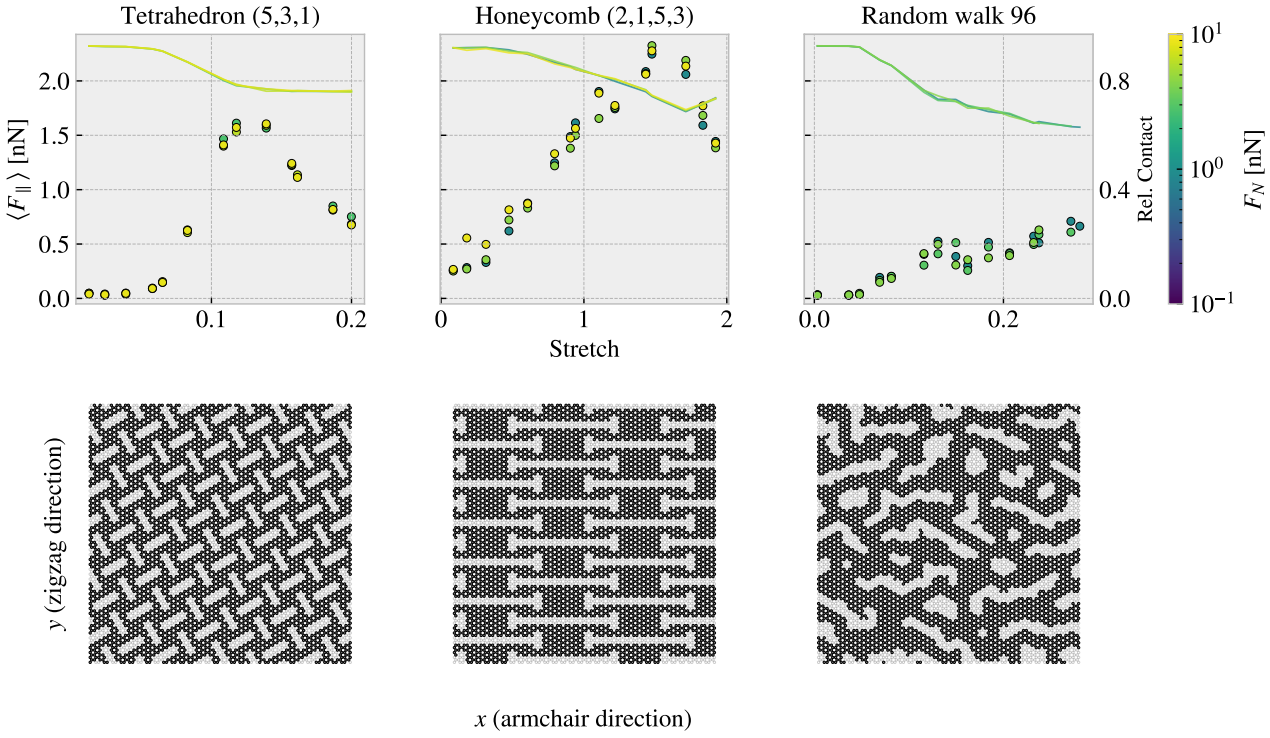


Figure 2.5: Maximum Difference: Configurations corresponding to the biggest difference in friction in the dataset for each pattern.

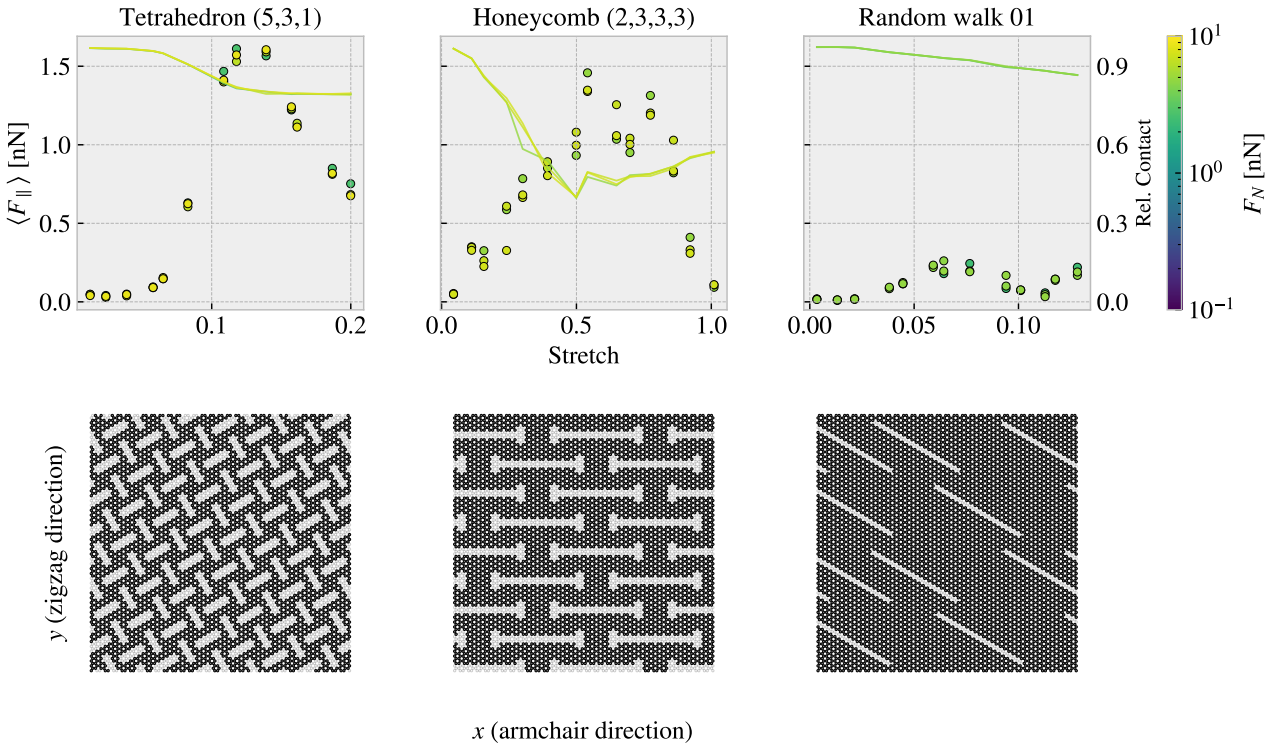


Figure 2.6: Maximum drop: Configuratiosn corresponding to the biggest friction drop in the dataset for each pattern.

2.4 Machine learning

Given the MD-based dataset we investigate the possibilities of training a machine learning model to predict the friction behavior from a given Kirigami configuration, strain and load.

2.4.1 Architecture

Due to the spatial dependencies in the Kirigami configurations, we use a convolutional neural network (CNN). Similar studies which predict mechanical properties for graphene sheets have used a VGGNet style network, Hanakata et al. [17, 18] and Wan et al. [19], which we adopt for this study as well. The VGGNet-16 architecture illustrated in Fig. 2.7 shows the key features that we will include:

- The image is processed through a series of 3×3 convolutional filters (the smallest size capable of capturing spatial dependencies) using a stride of 1 with an increasing number of channels throughout the network. We use padding to conserve the spatial size during a convolution. Each convolutional layer is followed by a ReLU activation function.
- The spatial dimensions are reduced by a max pooling, filter size 2×2 and a stride of 2, which halves the spatial resolution each time.
- The latter part of the network consists of a fully connected part using the ReLu activation as well. The transition from the convolutional to the fully connected part is achieved by applying a filter with the same dimensions as the last convolutional feature map. This essentially performs a linear mapping from the spatial output to the fully connected layer with the number of channels corresponding to the nodes in the first fully connected layer.

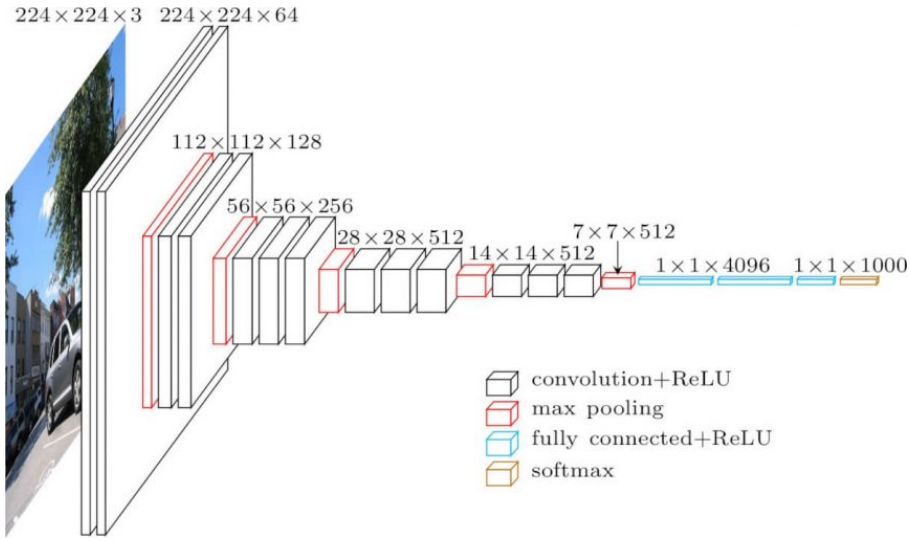


Figure 2.7: VGGNet 16. Source https://neurohive.io/en/popular-networks/vgg16/.

We deviate from the VGGNet-16 architecture by including batch normalization and restricting ourselves to setting up the convolutional part of the network in terms of the blocks: (Convolution → Batch normalization → ReLU → Max pooling). Similarly, we define a fully connected block by two elements (Fully connected → ReLU) which match the VGGNet model. Hanakata et al. and Wan et al. used a similar architecture with the parameters

$$\begin{aligned} \text{Hanakata et al. [17]} \quad & C16\ C32\ C64\ D64, \\ \text{Wan et al. [19]} \quad & C16\ C32\ D32\ D16, \end{aligned}$$

where C denotes a convolutional block with the number denoting the number of channels, and D a fully connected (dense) block with the number denoting the number of nodes. For the process of determining a suiting complexity for the architecture, we adopt the approach by Wan et al. [19] who used a "staircase" pattern for combining

convolutional and fully connected blocks. By defining a starting number of channels S and network depth D we fill the first half of the network layers with convolutional blocks, doubling in channel number for each layer, and the latter half with fully connected blocks having the number of nodes decrease in a reverse pattern. For instance, the architecture $S4D8$ will take the form

$$\text{Input} \rightarrow \underbrace{\overbrace{C4}^{S=4}\ C8\ C16\ C32\ D32\ D16\ D8\ D4}_{D=8} \rightarrow \text{Output}.$$

This provides a simple description where S and S can be varied systematically for a grid search over architecture complexity.

2.4.2 Data handling

2.4.2.1 Input

We use three variables as input: Kirigami configuration, strain of the sheet and applied normal load. The configuration is given as a two-dimensional input by the binary matrix while the strain and load are both scaler values. This gives rise to two different options for the data structure:

1. Expand the scalar values (strain and load) into 2D matrices of the same size as the Kirigami configuration by copying the scaler value to all matrix coordinates. This can then be merged into an image of three channels used as a single input.

2. Pass only the Kirigami configuration through the convolutional part of the network and introduce the remaining scalar values into the fully connected part of the network halfway in.

Both options utilize the same data, but the latter option is more directed towards independent processing of the data while the first makes for an intertwined ude of the configuration, strain and load. We implemented both options but found immediately that option 1 was producing the most promising results during the initial test runs, and thus we settled for this data structure.

2.4.2.2 Output

For the output, we are mainly concerned with mean friction and the rupture detection. In combination, this will make the model able to produce a friction-strain curve with an estimated stopping point as well. However, in order to retain the option to explore other relations in the data we include the maximum friction, relative contact, porosity and rupture strain in the output as well. Notice that we weigh the importance of these output variables differently in the loss as described in Sec. 2.4.3.

2.4.2.3 Data augmentation

In order to increase the utility of the available data one can introduce data augmentation. For most classification tasks this usually includes distortions such as color shifts, zoom, flip etc. However, such distortions are only valid since the classification network should still classify a cat as a cat even though it is suddenly a bit brighter or flipped upside down. For our problem, we can only use augmentation that matches a physical symmetry. Such a symmetry exists for reflection across the y-axis. We cannot use a reflection across the x-axis as the sheet is sliding in a positive y-direction. This would correspond to a change in the sliding direction which we cannot expect to be fully symmetric.

2.4.3 Loss

The output contains two different types of variables: scalar values and a binary value (rupture). For the scalar values we use the Mean Squared Error (MSE) Eq. (1.1) and for the binary output, we use binary cross entropy Eq. (1.2). We calculate the total loss as a weighted sum of the loss associated with each variable

$$L_{tot} = \sum_v W_v \cdot L_v.$$

We choose the weights to be 1/2 for the mean friction and 1/10 for the remaining 5 variables, thus sharing the loss evenly for the remaining 50% of the weight. During the introductory phase of the training, we tried different settings for these weights, but we found that the results varied little. Hence, we concluded that this was of minor importance and stuck to the values defined above.

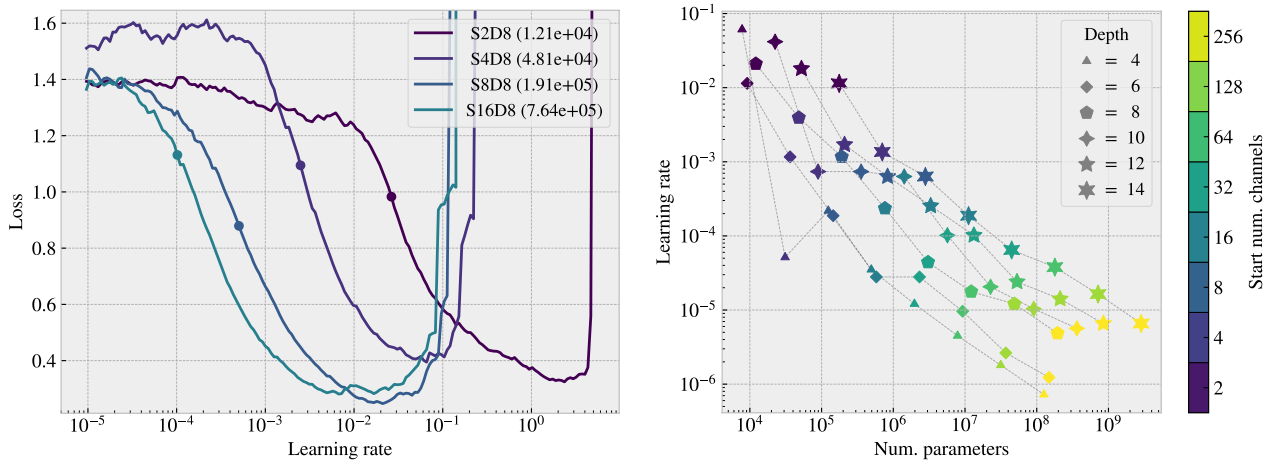
2.4.4 Hypertuning

For the hypertuning we focus on architecture complexity, learning rate, momentum and weight decay. We will use the ADAM optimizer with the initial default values of $beta_1 = 0.9, \beta_2 = 0.999$ and zero weight decay (we will change momentum β_1 and weight decay). We use a batch size of 32 and train the model for a maximum of 100 epochs while storing the best model based on the validation scores. Since the learning rate is considered to be one of the most important hyperparameters we will determine a suitable choice for the learning rate using the learning rate range test for each of the two major grid searches:

1. Architecture complexity grid search of S vs. D with individually chosen learning rates for each complexity combination.

2. Momentum vs. weight decay grid search with learning range chosen with regard to each momentum setting.

We consider first the architecture complexities in the range $S \times D = \{2, 4, 8, 16, 32, 64, 128, 256\} \times \{4, 6, 8, 10, 12, 14\}$. For each architecture complexity, we perform an initial learning rate range test and determine the suitable choice as the point for which the validation loss decreases most rapidly. The learning rate is increased exponentially within the range 10^{-7} to 10 with increments for each training batch iteration. This is done for just a single epoch where a batch size of 32 yields a total of 242 increments. This corresponds to an exponent increment of approximately 1/30 giving a relative increase $10^{1/30} \sim 108\%$ per batch iteration. The learning rate range test is presented in Fig. 2.8 for various model complexities. We notice that the suggested learning rate decreases with an increasing number of model parameters. This decrease is further independent of the specific relationship between S and D.



(a) **(b)**

Figure 2.8: Learning rate range test for various model complexities. We increase the learning rate exponentially from nume-7 to 10 during one epoch corresponding to an exponent increment of roughly 1/30 per batch iteration. (a) shows a few examples of the training loss history as a function of the learning rate. The exemplary architectures are S[2, 16]D8 with the corresponding number of model parameters shown in parentheses in the legend. The dot indicates the suggested learning rate at the steepest decline of the slope. (b) shows the full results of suggested learning rates depending on the number of model parameters with color coding differentiating the number of start channels and marker types differentiating different model depths.

With the use of the suggested learning rates from Fig. 2.8 we perform a grid search over the corresponding S and D parameters. We evaluate both the validation loss and the mean friction R_2 score for the validation data which is shown in Fig. 2.9 together with the best epoch and the number of model parameters. Additionally,

we evaluate the mean friction R_2 score for a selected set of configurations. This set consists of the top 10 configurations with respect to the max drop property for the Tetrahedron and Honeycomb patterns respectively. This is done as a way of evaluating the performance on the non-linear strain curves which we immediately found to be the more difficult trend to capture. The selected evaluation is shown in Fig. 2.10. Note that these configurations already are a part of the full dataset and thus the data points related to these configurations are most likely present in both the training and the validation data set. Hence, the performance must be considered in conjunction with the actual validation performance in Fig. 2.9.

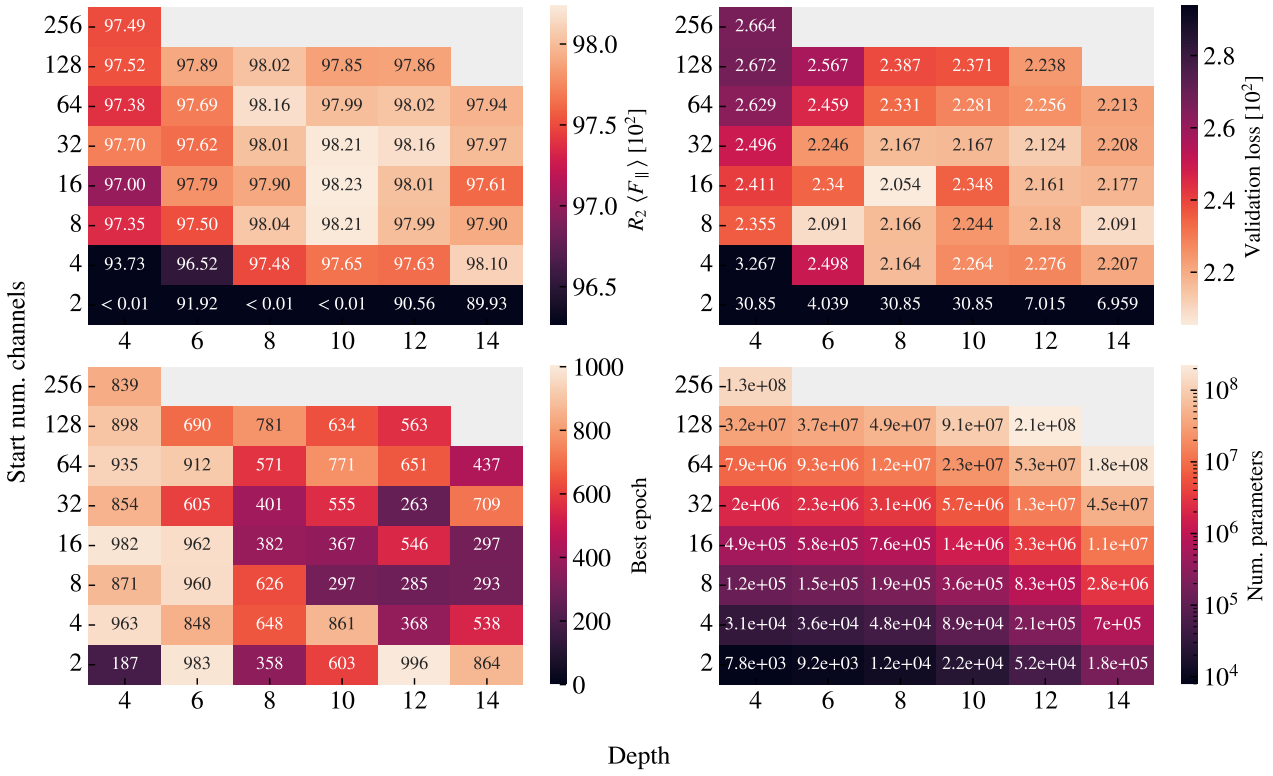


Figure 2.9: Architecture search.

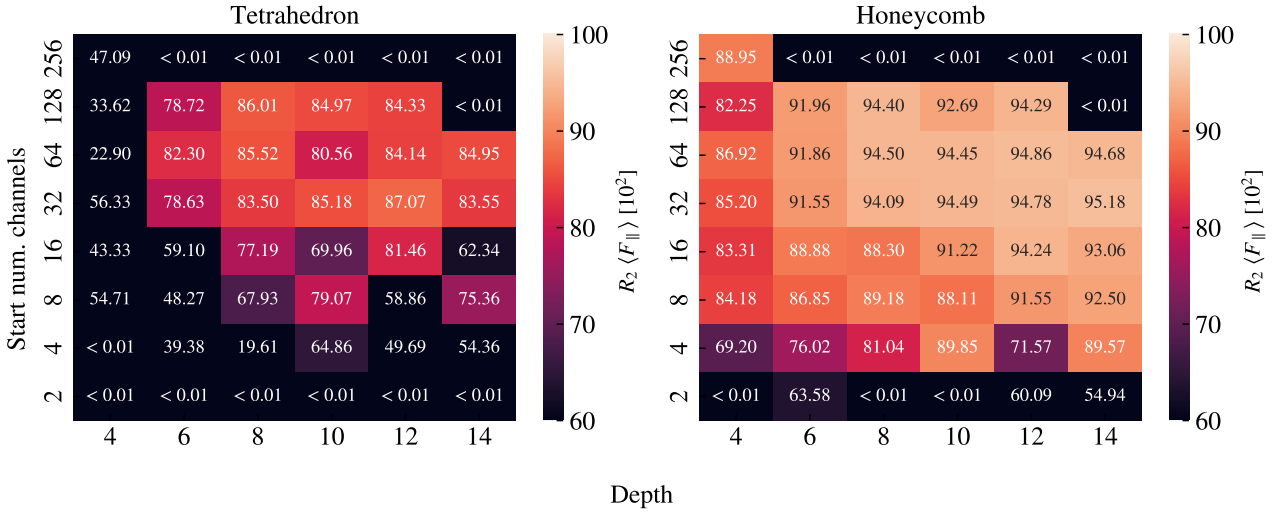
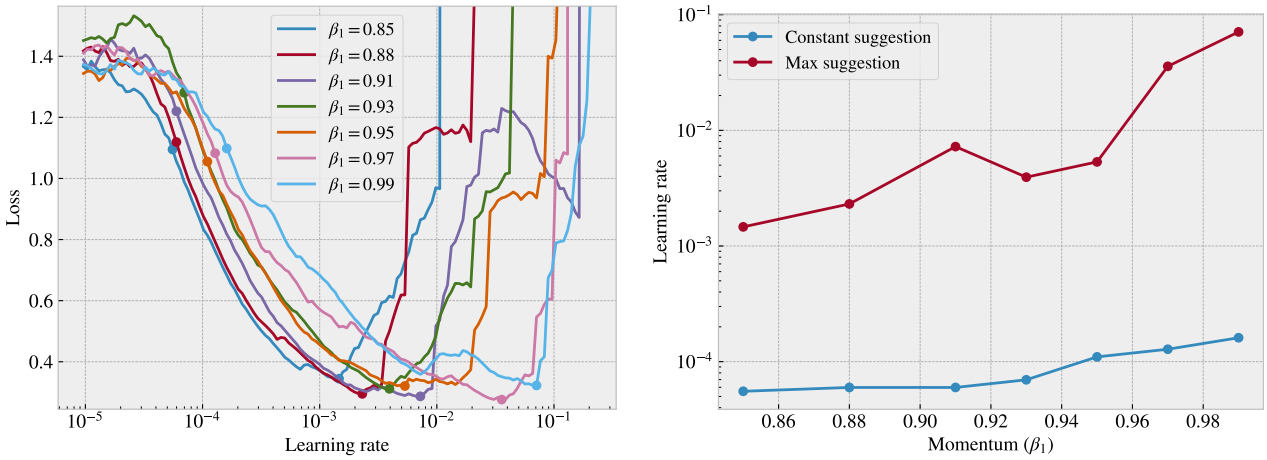


Figure 2.10: Selected pseudo validation set. Fix the missing grey fields in the top which are replaced by ¡ 0.01.

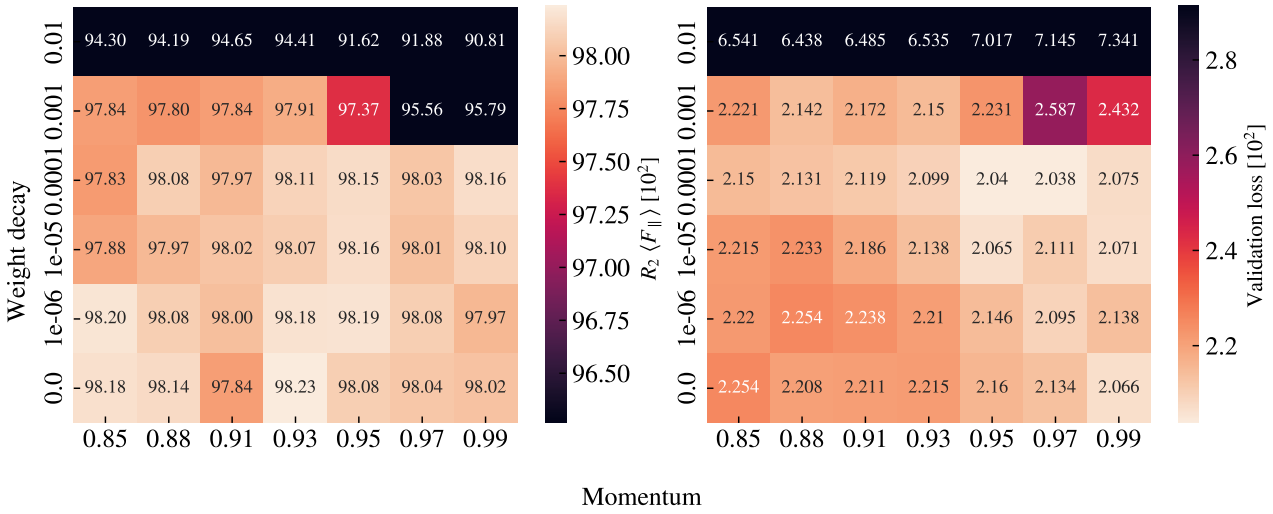
From the validation scores in Fig. 2.9, looking at both the loss and the R^2 scores, we find that models S(8-32)D(8-12) generally give the best performance. When looking at the best epoch we find that models of low depth result in a later best epoch which is compatible with underfitting. As the depth is increased we find more models with a lower best epoch, in the range $\sim [300, 600]$, which on the other hand suggest cases of overfitting. Since our training stores the best model during training, we do not have to worry too much about overfitting, but we can take this transition from underfitting to overfitting as a sign that our search is conducted in an appropriate complexity range. When consulting the evaluation on the selected set in Fig. 2.10 we find significantly lower R_2 scores, especially for the Tetrahedron pattern. Considering, that some of these data points are also present in the training data, this shows clearly that these configurations are more challenging to predict. While the peak R_2 value for the validation score in Fig. 2.9 was found for the model S16D10 model (98.23 %) the selected set test shows a slight preference for more complexity in the model. In the Tetrahedron selected set grid search, we find the best model to be S32D12, with an R_2 score of $\sim 87\%$. This model choice is more or less compatible with the overall performance as it is among the top candidates for the R_2 score and loss in Fig. 2.9 and the R_2 score for the Honeycomb pattern in Fig. 2.10 as well. Hence, we settle on this architecture.

Next, we consider momentum m and weight decays λ in the range $m \in [0.85, 0.99]$ and $\lambda \in [0, 1e-2]$. An increased momentum is expected to decrease the appropriate learning rate (check with theory), and thus we perform a new learning rate range test for each momentum choice. We propose two learning rate schemes: A constant learning rate as used until this point and a one-cycle policy. In the one-cycle policy we set a maximum bound for the learning rate and start from a factor 1/20 of this bound and increase towards the maximum bound during the first 30% of the training. For the final 70% of training we decrease towards a final minimum given as a factor $1e-4$ of the maximum bound . The increase and decrease are done by a cosine function. The suggested learning rate for the constant learning rate scheme is once again determined by the steepest slope on the loss curve while the maximum bound used for the one-cycle policy is determined as the point just before divergence. We find that the minimum point on the loss curve is a suitable choice that approaches the diverging point without getting too close and causing instabilities. The learning rate range test for momentum is shown in Fig. 2.11. We observe that a higher momentum giver higher suggested learning rates, so I need to check with theory on that. Using the results for the momentum learning rate range test we perform a grid search of momentum and weight decay. We examine again the validation loss and validation mean friction R_2 score in addition to the friction mean R_2 score for the selected set of Tetrahedron and Honeycomb patterns. This is shown for the constant learning rate scheme in Fig. 2.12 and for the cyclic scheme in Fig. 2.13.

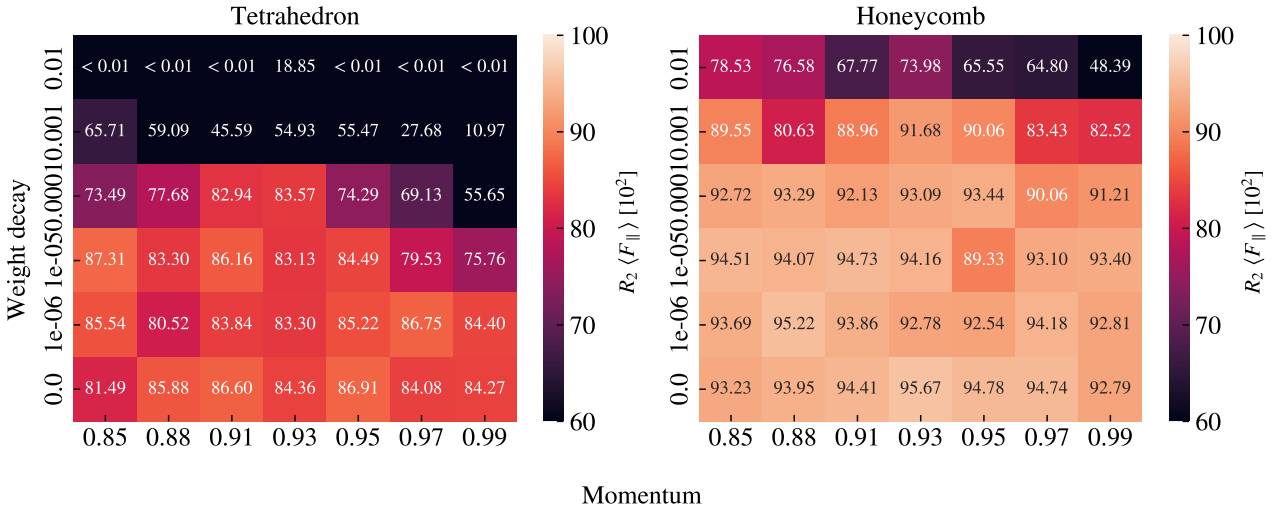


(a) **(b)**

Figure 2.11: Momentum learning rate range tets

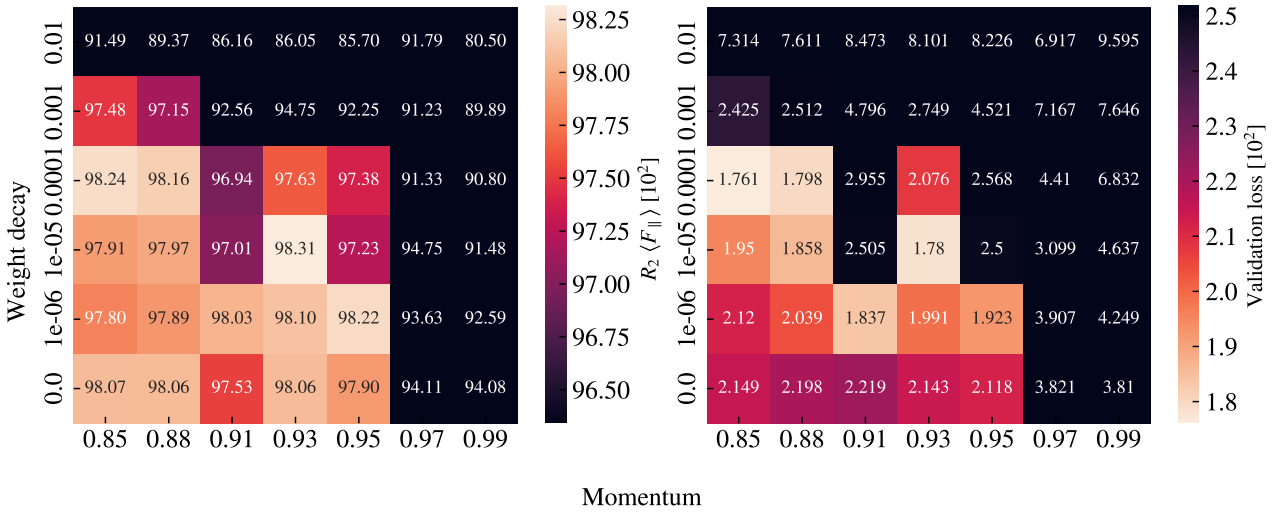


(a) Validation performance.

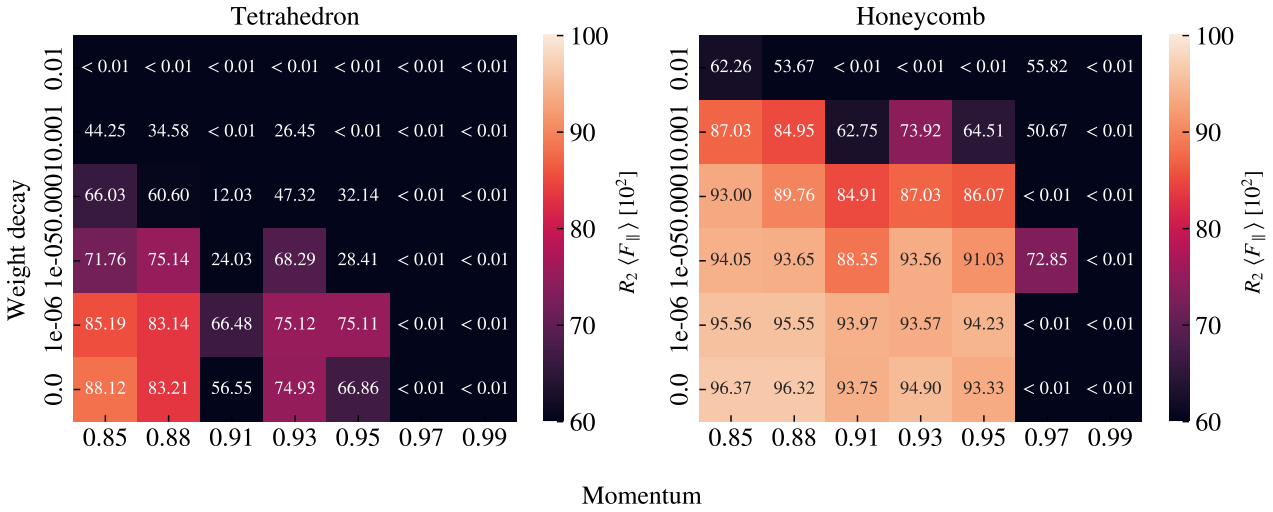


(b) Selected set comparison.

Figure 2.12: Constant learning rate and momentum scheme



(a) Validation performance.



(b) Selected set comparison.

Figure 2.13: Cyclic learning rate and momentum scheme

The original validation scores, before varying momentum and weight decay, were a validation loss of 0.02124 and a mean friction R_2 score of 0.9816. By varying momentum and weight decay, we can improve these scores slightly for the constant learning rate scheme (loss: 0.02038, R_2: 0.9823) and even more for the cyclic scheme (loss: 0.0176, R_2: 0.9831). However, notice that these scores a taking for separate combinations of hyperparameters. The comparison among best scores is summarized in Table 2.3. In general, the constant scheme shows rather stable results for all momentum settings $m \in [0.85, 0.99]$ in combination with a low weight decay $\lambda \leq 10^{-4}$. For the cyclic scheme the performance peaks towards a low momentum $m \leq 0.93$ and low weight decay $\lambda \leq 10^{-4}$. Looking at the summary in Table 2.3 we see that the cyclic scheme can produce a high score among all four performance metrics, but since these scores do not share common hyperparameters we need to choose which of them to prioritize. Due to our interest in capturing the non-linear trends, we prioritize the score from the selected set of Tetrahedron patterns as this provided the greatest challenge for our model to capture. We recognize that this choice introduces a greater risk of overfitting since the data points are partly included in the training set. However, for the purpose of performing an accelerated search, we find it more important to increase the chances of discovering novel designs than to reduce the chance of getting false positive results. Since we have the option to verify the properties of a given design through MD simulations we do not have to rely on the machine learning prediction as a final score but only as a proposal for configurations worth further studing. Thus we choose the

cyclic trained model with low momentum $m = 0.85$ and zero weight decay as our final model. Finally, we note that since our choice of hyperparameters corresponded to the edge of our grid search it would have been natural to perform an extended search in that range. This was omitted due to time prioritization and the belief that the potential gain of doing so was not huge.

Table 2.3: Momentum and weight decay grid search using S32D12 model.

		Score [10^2]	Momentum	Weight decay
Validation loss	Original	2.124	0.9	0
	Constant	2.038	0.97	10^{-4}
	Cyclic	1.761	0.85	10^{-4}
Validation R_2	Original	98.16	0.9	0
	Constant	98.23	0.93	0
	Cyclic	98.31	0.93	10^{-5}
Tetrahedron R_2	Original	87.07	0.9	0
	Constant	87.31	0.85	10^{-5}
	Cyclic	88.12	0.85	0
Honeycomb R_2	Original	94.78	0.9	0
	Constant	95.67	0.93	0
	Cyclic	96.37	0.85	0

2.4.5 Final model

From the hypertuning study, we choose the S32D12 model trained by a cyclic training scheme with a momentum of 0.85 and zero weight decay. The main performance metrics are shown in Table 2.4. Since the porosity is a number between 0 and 1 we can interpret the absolute error as the percentage error similar to the relative error for the rupture strain. The rupture strain is generally within a 13 % margin, but we believe that this number is especially high due to some low strain rupture cases in the dataset which contributes to a large relative error. The strain curves for mean friction, max friction and contact is shown in Fig. 2.14 in comparison with the Tetrahedron (7, 5, 1) and Honeycomb (2,2,1,5) used in the pilot study. This gives a visual interpretation of how well the fits are for the given R_2, and we get a visual confirmation that a R_2 score above 0.98 indeed looks like a promising capture for the non-linear trends in the data. We will perform a true test set evaluation later on, based on some of the suggestions from the accelerated search.

Table 2.4: Mean values are used over different configurations.

	Loss [10^2]	R_2 [10^2]			Abs. [10^2]	Rel. [10^2]	Acc. [10^2]
	Total	Mean F_f	Max F_f	Contact	Porosity	Rup. Strain	Rupture
Validation	2.1488	98.067	93.558	94.598	02.325	12.958	96.102
Tetrahedron	4.0328	88.662	85.836	64.683	01.207	05.880	99.762
Honeycomb	8.6867	96.627	89.696	97.171	01.040	01.483	99.111

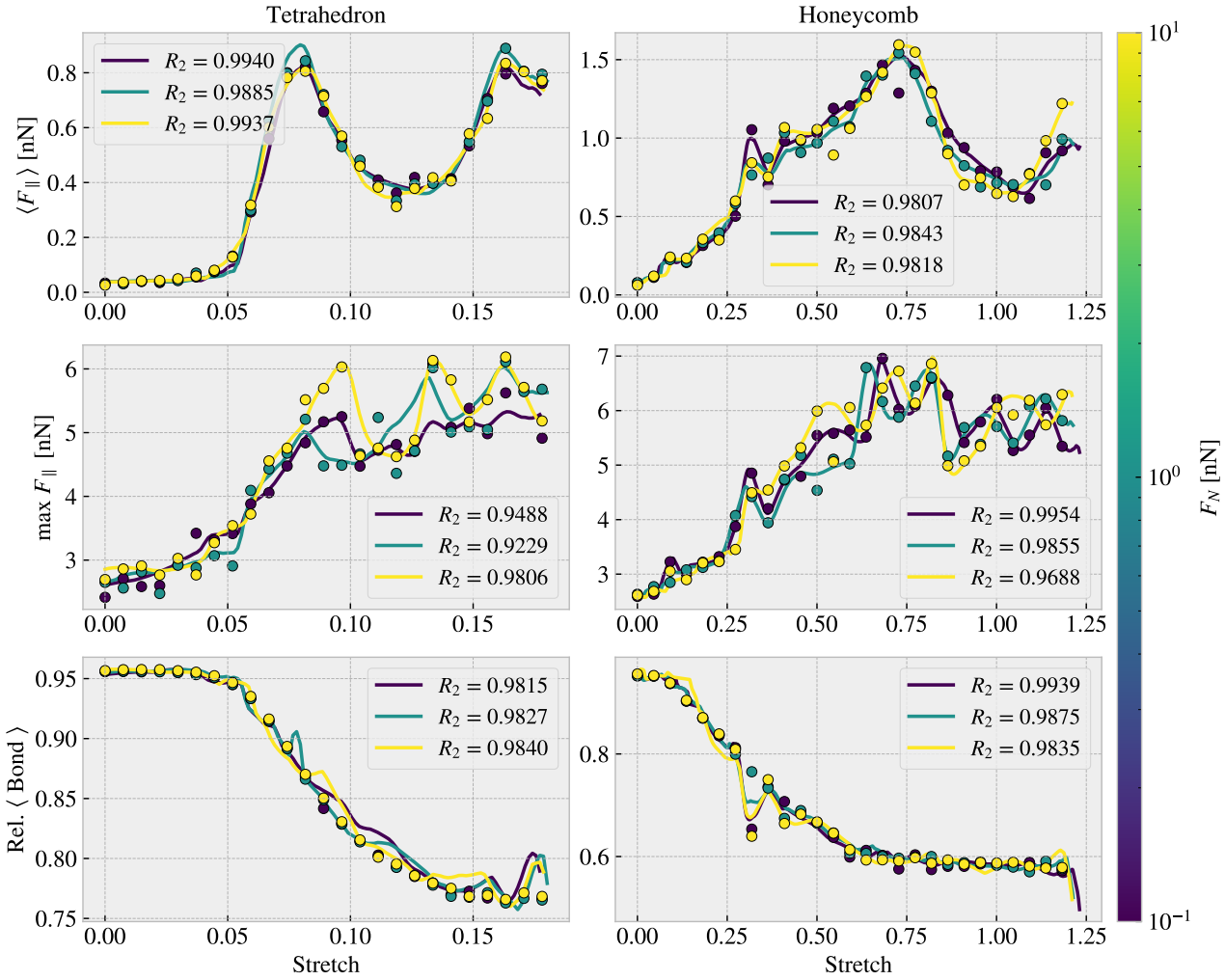


Figure 2.14: With 10^3 points in the strain range [0, 1.5] and stopping after first rupture True prediction.

Using our final model, we evaluate the performance for the top ranking within the properties of interest. That is, we go through all the configurations in the dataset for the Tetrahedron (Table 2.5), Honeycomb (Table 2.6) and Random Walk (Table 2.7) patterns separately and rank the configurations in each property category. This is then compared to the actual ranking in the dataset. Generally, we find that the ML performs rather well in the ranking of the max, max difference and max drop properies, but it is deviating a bit more for the minimum friction property. This can be attritbuted to the fact that the precision needed for an accurate ranking among the minimum friction cases is a lot greater than for the remaining properties. The ML model gives mostly similar predictions for the max-categories ranking for the Tetrahedron and Honeycomb pattern but struggles a bit more for the Random Walk. Looking at the values for the top candidates in the max-types properties we see that it is generally within a ~ 0.2 nN range which is promising.

Table 2.5: Tetrahedon

ML Rank	Data Config	Data Value [nN]	ML Config	ML Value [nN]	Data Rank
min F_{fric}					
20	(3, 9, 4)	0.0067	(3, 1, 2)	0.0041	5
5	(3, 1, 3)	0.0075	(1, 3, 4)	0.0049	11
6	(5, 3, 4)	0.0084	(1, 3, 3)	0.0066	6
21	(1, 7, 3)	0.0084	(3, 1, 4)	0.0066	8
1	(3, 1, 2)	0.0097	(3, 1, 3)	0.0078	2
max F_{fric}					
1	(5, 3, 1)	1.5875	(5, 3, 1)	1.5920	1
2	(1, 3, 1)	1.4310	(1, 3, 1)	1.2739	2
4	(3, 1, 2)	1.0988	(9, 3, 1)	1.1162	4
3	(9, 3, 1)	1.0936	(3, 1, 2)	0.7819	3
5	(7, 5, 1)	0.7916	(7, 5, 1)	0.7740	5
max ΔF_{fric}					
1	(5, 3, 1)	1.5529	(5, 3, 1)	1.5578	1
2	(1, 3, 1)	1.3916	(1, 3, 1)	1.2331	2
4	(3, 1, 2)	1.0891	(9, 3, 1)	1.0807	4
3	(9, 3, 1)	1.0606	(3, 1, 2)	0.7778	3
5	(7, 5, 1)	0.7536	(7, 5, 1)	0.7399	5
max drop					
1	(5, 3, 1)	0.8841	(5, 3, 1)	0.8603	1
2	(3, 5, 1)	0.4091	(3, 5, 1)	0.3722	2
4	(7, 5, 1)	0.3775	(1, 1, 1)	0.2879	5
5	(9, 7, 1)	0.2238	(7, 5, 1)	0.2478	3
3	(1, 1, 1)	0.1347	(9, 7, 1)	0.1302	4

Table 2.6: Honeycomb

ML Rank	Data Config	Data Value [nN]	ML Config	ML Value [nN]	Data Rank
min F_{fric}					
1	(2, 5, 1, 1)	0.0177	(2, 5, 1, 1)	0.0113	1
9	(2, 4, 5, 1)	0.0187	(2, 5, 5, 3)	0.0149	7
7	(2, 4, 1, 1)	0.0212	(2, 5, 5, 1)	0.0182	4
3	(2, 5, 5, 1)	0.0212	(2, 5, 3, 1)	0.0186	5
4	(2, 5, 3, 1)	0.0226	(2, 4, 1, 3)	0.0198	15
max F_{fric}					
1	(2, 1, 1, 1)	2.8903	(2, 1, 1, 1)	2.9171	1
2	(2, 1, 5, 3)	2.2824	(2, 1, 5, 3)	2.4004	2
6	(2, 1, 3, 1)	2.0818	(2, 1, 5, 1)	2.1060	5
4	(2, 1, 3, 3)	2.0313	(2, 1, 3, 3)	1.9458	4
3	(2, 1, 5, 1)	2.0164	(2, 4, 1, 1)	1.9381	6
max ΔF_{fric}					
1	(2, 1, 5, 3)	2.0234	(2, 1, 5, 3)	2.1675	1
2	(2, 1, 1, 1)	1.9528	(2, 1, 1, 1)	2.0809	2
3	(2, 4, 1, 1)	1.8184	(2, 4, 1, 1)	1.9157	3
4	(2, 1, 3, 3)	1.7645	(2, 1, 3, 3)	1.6968	4
5	(2, 4, 1, 3)	1.4614	(2, 4, 1, 3)	1.5612	5
max drop					
1	(2, 3, 3, 3)	1.2785	(2, 3, 3, 3)	1.3642	1
2	(2, 1, 3, 1)	1.1046	(2, 1, 3, 1)	0.9837	2
3	(2, 3, 3, 5)	0.8947	(2, 3, 3, 5)	0.9803	3
4	(2, 1, 5, 3)	0.8638	(2, 1, 5, 3)	0.9556	4
13	(2, 5, 1, 1)	0.8468	(2, 4, 5, 3)	0.8999	8

Table 2.7: RW

ML Rank	Data Config	Data Value [nN]	ML Config	ML Value [nN]	Data Rank
min F_{fric}					
1	12	0.0024	12	-0.0011	1
24	76	0.0040	06	0.0036	27
6	13	0.0055	14	0.0074	23
31	08	0.0065	05	0.0082	19
26	07	0.0069	63	0.0085	57
max F_{fric}					
3	96	0.5758	99	0.5155	2
1	99	0.5316	98	0.4708	3
2	98	0.4478	96	0.4356	1
4	97	0.3624	97	0.3503	4
11	58	0.3410	55	0.2817	7
max ΔF_{fric}					
3	96	0.5448	99	0.4669	2
1	99	0.4769	98	0.4314	3
2	98	0.4085	96	0.4128	1
4	97	0.3268	97	0.3080	4
78	57	0.2978	55	0.2542	7
max drop					
3	01	0.1818	00	0.1883	3
2	96	0.1733	96	0.1654	2
1	00	0.1590	01	0.1532	1
11	37	0.1022	04	0.0591	8
28	34	0.0879	56	0.0552	20

2.5 Accelerated Search

Having trained a machine learning (ML) model we use it for an extended accelerated search for further optimization of the friction properties of interest. We approach this search by two different methods:

1. Using the generative algorithms developed for the creation of the Tetrahedron, Honeycomb and Random walk patterns, we create yet an extended dataset and evaluate the performance using the ML.

2. Using the genetic algorithm method we perpetuate the configurations and optimize for the maximum drop property using the ML model to evaluate the fitness function.

2.5.1 Patteren generation search

We utilize the pattern generators to create an extended dataset for our search. For the Tetrahedron and Honeycomb patterns, the increment of the parameters will eventually lead to the main structures becoming so large they do exceed the size of the sheet. Thus, we can essentially perform a full search "maxing out" the parameters of these patterns. We estimate that this is done with the max parameters, $(60, 60, 30)$ for the Tetrahedron, and $([30, 30, 30, 60])$ for the Honeycomb. We use a random reference position and regenerate each unique parameter 10 times to explore translational effects. This gives in total 135k configurations for the Tetrahedron pattern and 2025k for the Honeycomb pattern. For the Random walk generator, we do a Monte Carlo sampling. In each sample, we draw the scalar values, either from a uniform (U) or logarithmic uniform (LU) distribution as follows.

$$\text{Num. walks} \sim \text{U}[1, 30] \qquad \text{Max. steps} \sim \text{U}[1, 30] \qquad \text{Min. dis.} \sim \text{U}[0, 4]$$
$$\text{Bias direction} \sim \text{U}[0, 2\pi] \qquad \text{Bias. strength} \sim \text{LU}[0, 10] \qquad p_{\text{stay}} \sim \text{U}[0, 1]$$

Notice that we use discrete distribution for the parameters requiring integers. For the binary parameters *Connection*, *Avoid unvalid*, *RN6* and *Grid start* we simply set the values by a 50–50 chance. The remaining parameters are kept constant at *Periodic: True* and *Centering:False* throughout the search. For the handling of clustering, we implement the repair algorithm such that the sheet is repaired by the least modifications approach rather than retrying the generation several times Make sure that this is introduced somewhere. Due to the extra computation time associated with the random walk and the repair algorithm, we only generate 10k configurations within this class. For the evaluation of the configurations we use a normal load of 5 nN and generate a strain curve in the domain 0–200 % using 100 evenly spaced points. Top candidate results for each property are shown in Table 2.8 including a comparison to the dataset top candidates originally shown in Table 2.2. The random walk top five candidates are visualized in Fig. 2.16.

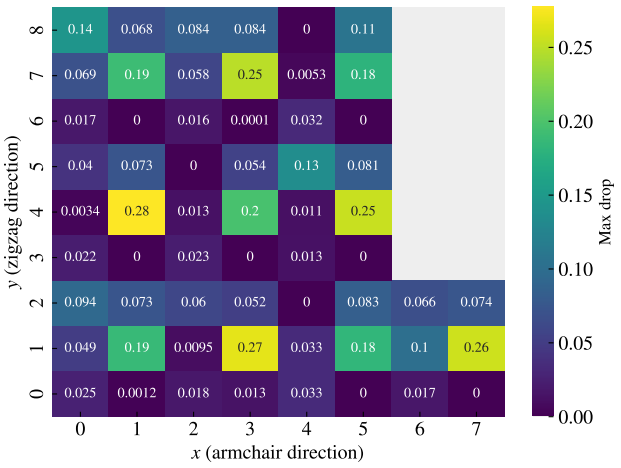
First of all, the search shows a rather consistent result regarding the minimization of friction where the top candidates all share the same feature of being sparsely cut. For the Random walk, we see this visually inFig. 2.16, while for the Tetrahedon and Honeycomb patterns, this is evident from the configuration parameters shown in Table 2.8 where the parameters reveal a high spacing between the cuts. The porosity of the minimum friction top candidates are all rather low being 1.5%, 5.6%, and 1.6% for the Tetrahedron, Honeycomb and Random walk respectively. These results point toward Kirigami modifications not being applicable for a lowering of friction, within the limitations of our numerical setup. This is in agreement with the initial analysis on the MD dataset, and hence the dataset does not provide evident suggestions for friction minimization. Thus we might argue that the model is limited on the dataset for this property. The fact that the top candidates all take negative values also shows that the model is not reliable in this domain. By asking for the lowest friction we effectively take advantage of the sparsely populated areas of the model parameter space which can lead to unphysical predictions. In order to resolve this problem one might attempt an extension of the dataset and possibly also applying a physical constraint for positive friction values.

Among the remaining maximum-type properties, we find competing values for the Honeycomb and Random walk classes. When taking a closer look at the ranking for each property it becomes apparent that the predictions are highly sensitive to the reference position parameters used for the Tetrahedron and Honeycomb pattern. Since we repeated each pattern parameter 10 times for random reference positions, we initially expected to get a ranking in sets of 10. However, the ranking shows contiguous appearing sets in the range 1–5 which points toward a sensitivity for pattern translation. Hence we investigate this further by evaluating the scores for a systematic change of the reference position for selected configurations. We generally finn the max drop parameter to give the highest variation and thus we show scores for the max drop top candidates, Tetrahedron $(1, 7, 1)$, $(5, 3, 1)$ and Honeycomb $(3, 3, 5, 3)$, $(2, 3, 3, 3)$, in Fig. 2.15. It becomes evident that the predictions vary drastically with the translation of these patterns. The emerging question is then whether this is actually grounded in a physical phenomenon or simply a deficiency in the ML. Even though the patterns are periodic in the x-y-plane, with period matching the unique translations shown in Fig. 2.15, the translation will determine the specific configuration of the edge. Previous studies of static friction and stick-slip behavior point to the importance of edge effects look back at theory and maybe source, and thus for a sheet where the atoms sitting on the $\pm x$ free sides constitutes about 2.5% of the inner sheet atom count, it is not unreasonable that the translation might result in a significantly different outcome. In that case, the search through reference positions highlights that the translation can be key to optimizing for certain properties. However, the results might also indicate that the model is either overfitted or that we simply did not provide enough data to reach a generalization of the complex physical behavior of the system. The sensible way forward to unravel this would be to generate additional translational variants of the same configurations to investigate for any physical edge dependencies or otherwise strengthen the model. We earmark this suggestion for another study. When considering some of the strain curves we also find that the prediction of the rupture point plays an important role for the max drop property. As the rupture was often predicted on a descending part of the curve any variation to the rupture point will affect the max drop property quite significantly.

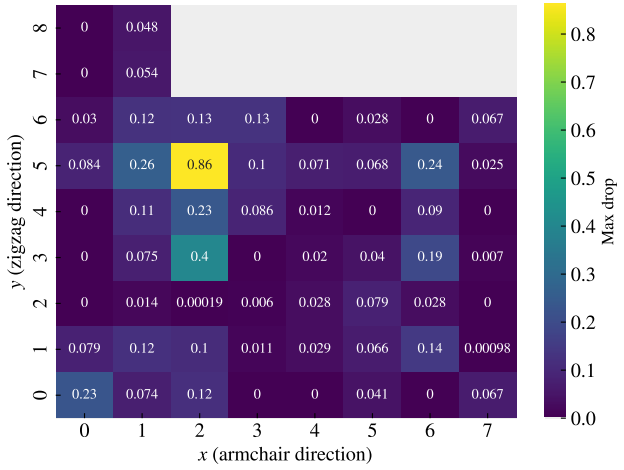
We use the 20 configurations from the top candidates in the Random walk class as a test set. We sample 30 pseudo uniform strain values and use a normal load of 5 nN. The test set gave a loss 2.13 which is two orders of magnitude higher than the validation and an average absolute error for the mean friction of 0.14 and rupture accuracy of 70 %. This corresponded to an average negative R^2 score indicating that we would have been better off guessing on a constant value corresponding to the true data mean. This shows that the model is clearly not generalized. While some parts can be attributed to the overfitting of the model we believe that this is more likely to be a problem of a too small dataset. This was just recently added so I need to incorporate this in the further discussion... It is a bit sad..

Table 2.8: Pattern search. The values are in units nN.

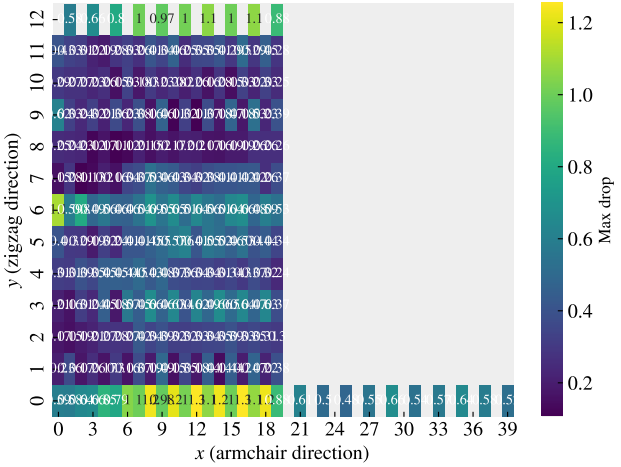
	Search			Data		
Scores	Tetrahedron	Honeycomb	Random walk	Tetrahedron	Honeycomb	Random walk
min F_{fric}	-0.062	-0.109	-0.061	0.0067	0.0177	0.0024
max F_{fric}	1.089	2.917	0.660	1.5875	2.8903	0.5758
max ΔF_{fric}	1.062	2.081	0.629	1.5529	2.0234	0.5448
max drop	0.277	1.250	0.269	0.8841	1.2785	0.1818
Configs.	Tetrahedron	Honeycomb	Random walk	Tetrahedron	Honeycomb	Random walk
min F_{fric}	$(13, 11, 14)$	$(14, 25, 7, 19)$	No naming	$(3, 9, 4)$	$(2, 5, 1, 1)$	12
max F_{fric}	$(1, 3, 1)$	$(2, 1, 1, 1)$	No naming	$(5, 3, 1)$	$(2, 1, 1, 1)$	96
max ΔF_{fric}	$(1, 3, 1)$	$(2, 1, 1, 1)$	No naming	$(5, 3, 1)$	$(2, 1, 5, 3)$	96
max drop	$(1, 7, 1)$	$(3, 3, 5, 3)$	No naming	$(5, 3, 1)$	$(2, 3, 3, 3)$	01



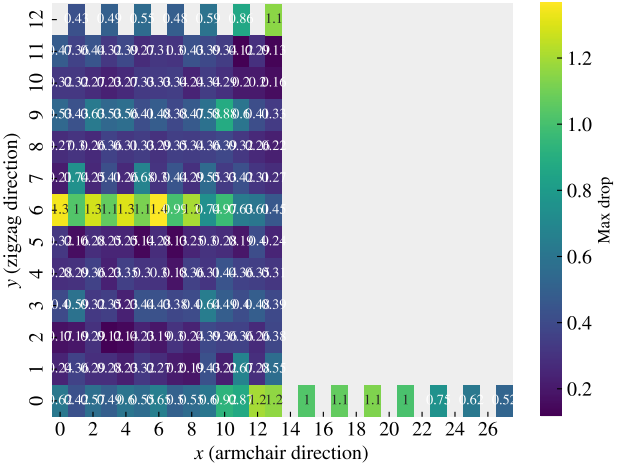
(a) Tetrahedron $(1, 7, 1)$ (60). Std = 0.08, Rel. Std = 1.13



(b) Tetrahedron $(5, 3, 1)$ (60). Std = 0.13, Rel. Std = 1.61



(c) Honeycomb $(3, 3, 5, 3)$ (260). Std = 0.25, Rel. Std = 0.58



(d) Honeycomb $(2, 3, 3, 3)$. (182) Std = 0.27, Rel. Std = 0.60

Figure 2.15: CAPTION

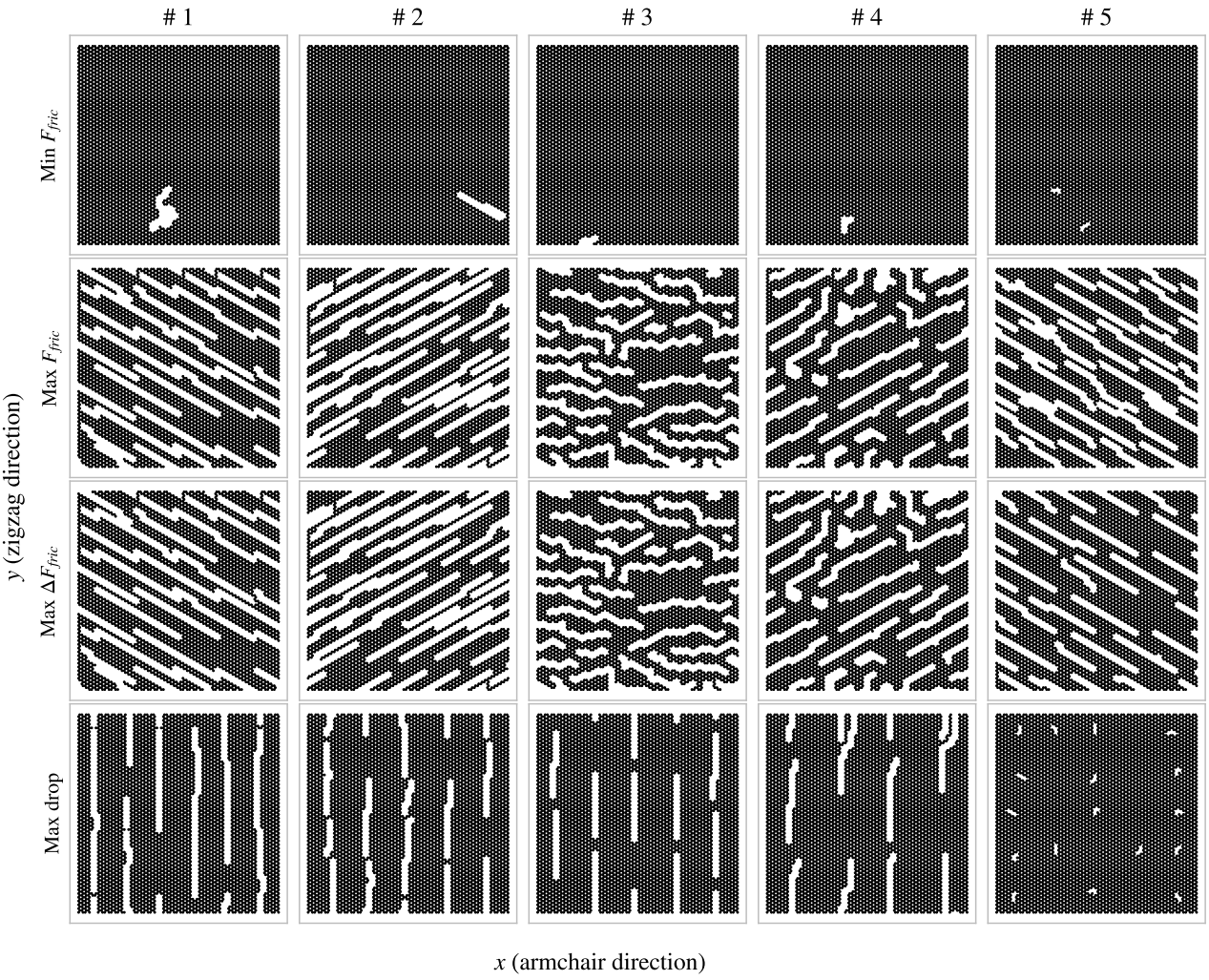


Figure 2.16: RW search top results.

2.5.2 Genetic algorithm search

For the second approach to an accelerated search, we consider the genetic algorithm. So far we have concluded that a minimization of the friction is not promising, and hence we discard this property for further studing. We have also seen that the maximum style properties often share similar top candidates, and thus we choose to only investigate the max drop property, associated with the aim of creating a negative friction coefficient. We are going to use the extended search top candidate as a basis for the genetic algorithm population algorithm search. We generate a population of 100 configurations using the settings associated with the max drop top candidate for the Tetrahedron, Honeycomb and Random walk patterns respectively. We run the search for 50 generations as we did not see much difference for any longer runs. The Tetrahedron and Honeycomb search did immediately not give any improvements as the highest scoring individual from the population was not improved through the search even though the average score was rising initially. For the Random walk, we choose to perform a genetic algorithm search for each of the top 5 candidates. Most of them gave a similar result as seen from the Tetrahedon and Honeycomb pattern with only a single new candidate generated. The score of this candidate was 0.240 nN which is a small improvement from the best score of 0.182 nN. However, from some of the other runs the population initialization provided a top score of 0.345 nN which shows that we have better hopes of optimizing this property by simply generating more configurations from the top candidate parameters. The fact that starting from an existing design did not give any useful results we attempted to start from a population of random noise as well. We made one population from mixed porosities, having even 20 individuals each for porosity $\{0.01, 0.05, 0.1, 0.2, 0.3\}$, and two populations based on a porosity of 0.25 and 0.5 respectively. This

time the algorithm improved the top candidate throughout, but the actual scores are still not impressive. The mixed porosity start gave the highest score, being 0.299 nN. When considering the corresponding patterns from the top five candidates in this search, they were all visually quite similar to the starting configurations; they still looked like random noise. Thus, we do not find signs of a generation of any noticeable patterns worth further investigating. By the use of the Grad-CAM method, we examined for any noticeable patterns for the model predictions on this mixed porosity top candidate as shown Fig. 2.17. For comparison, we included a similar examination for the top candidates in the pattern generation search with respect to the max drop category for the Tetrahedron, Honeycomb and Random walk shown Fig. 2.18 to 2.19. For the mixed porosity top candidate, the Grad-CAM method highlights some areas in the noise configuration as contributing more positively than others. However, we do not see any obvious patterns. For the more structured patterns of the Tetrahedron, Honeycomb and Random walk configurations we see in many cases throughout the straining that the cut parts are highlighted. This gives some confidence to the idea that the model does consider some of the relevant features in the pattern, but it still varies too much to make any strong conclusion. However, we notice that for certain strain values, the Grad-CAM reveals an "attention" towards the edge of the configuration. This especially relates to the top and bottom edge, which we for instance see for the Honeycomb pattern at strain 0.396 regarding the bottom edge in Fig. 2.19. Considering that the top and bottom of the configuration are not a true edge, since these are connected to the pull blocks in the simulation, this is a bit surprising. One interpretation is that the dissipation associated with the thermostat in the pull blocks might be of importance. This is not strongly supported and we simply note that as an interesting topic for further studies.

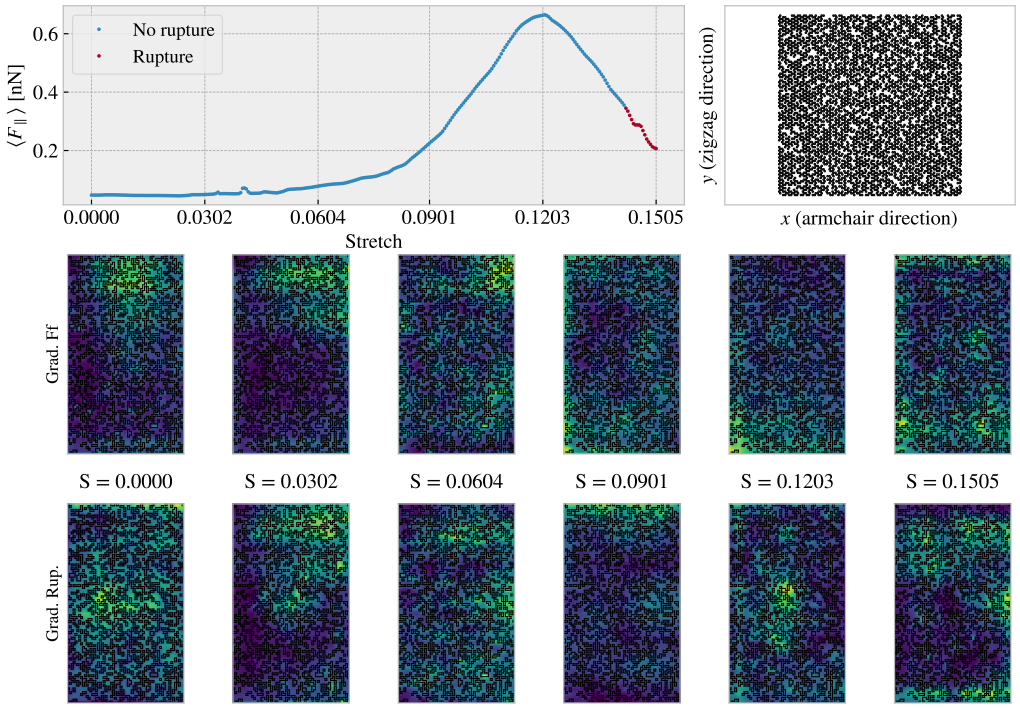


Figure 2.17: $p \in \{0.01, 0.05, 0.1, 0.2, 0.3\}$.

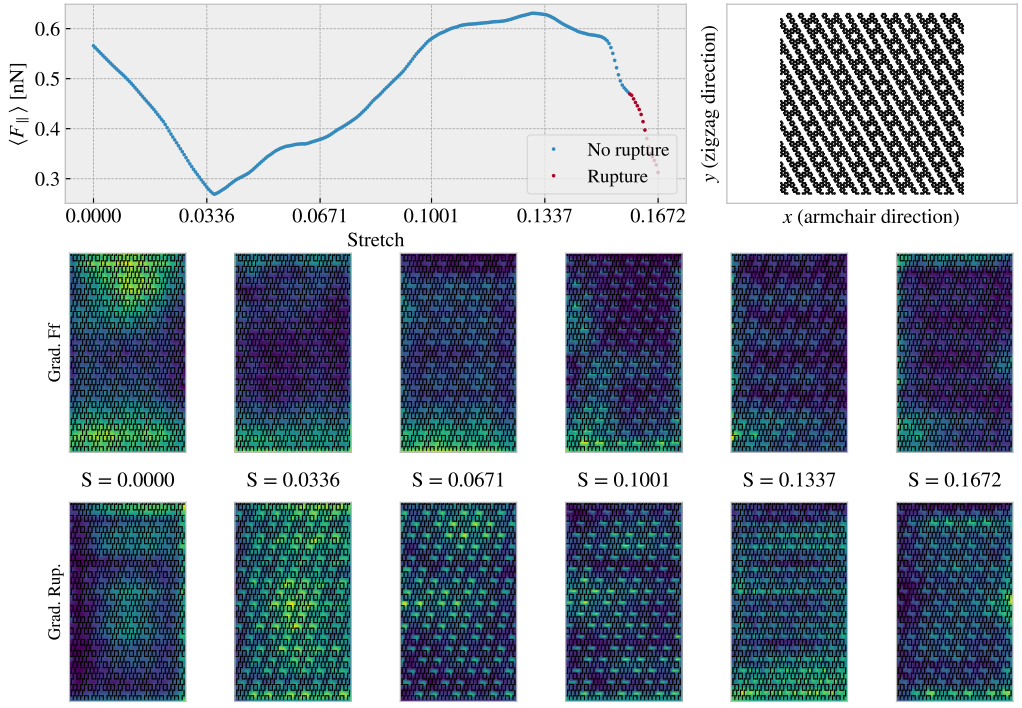


Figure 2.18: Tetrahedron $(1, 7, 1)$, ref $= (1, 4)$

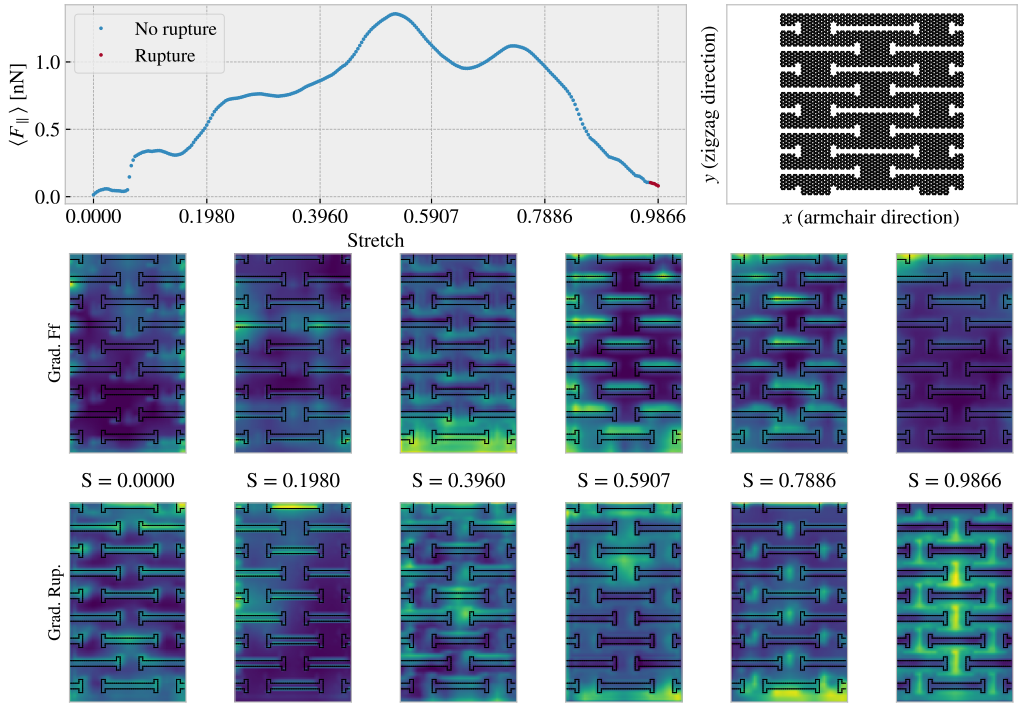


Figure 2.19: Honeycomb $(3, 3, 5, 3)$, ref $= (12, 0)$

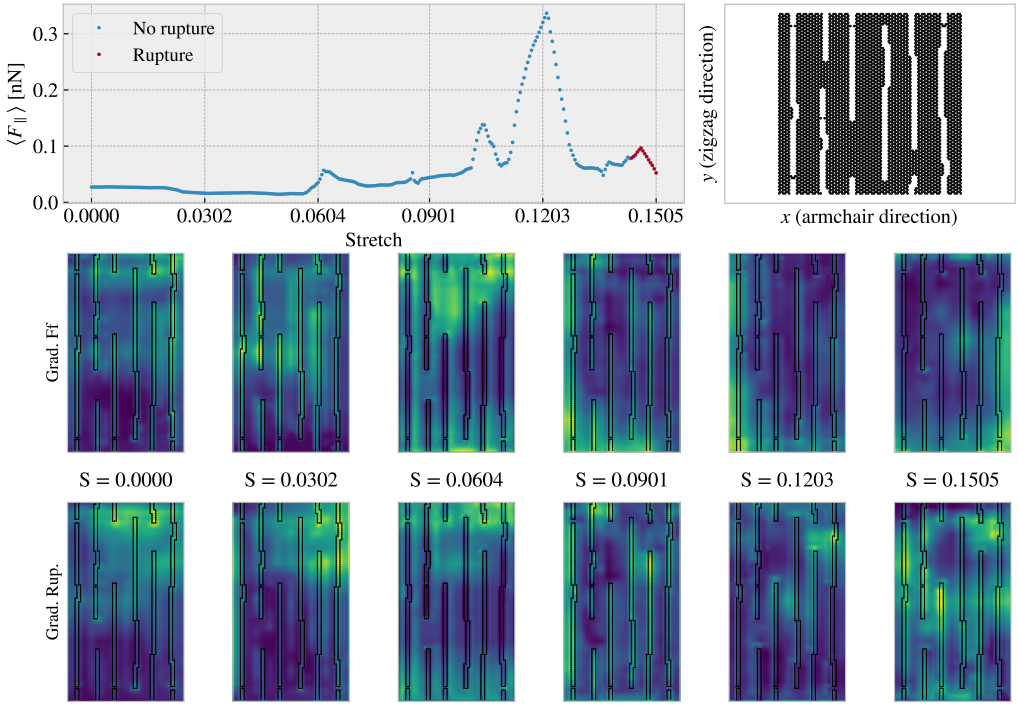


Figure 2.20: RW.

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.

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

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

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