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

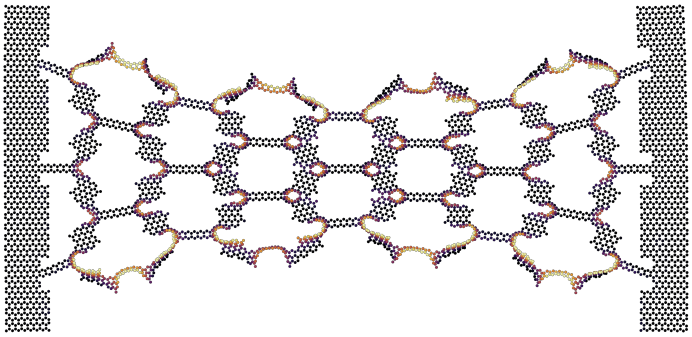
UNIVERSITY OF OSLO

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. 12, 13, 14

EMA Exponetial Moving Average. 10

MD Molecular Dynamics. 1, 2, 3, 7

ML Machine Learning. 2, 3

MSE Mean Squared Error. 8

RMSProp Root Mean Square Propagation. 10

SGD Stochastic gradient descent. 9

Contents

Chapter 1

Introduction

1.1 Motivation

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

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

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

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

In materials science such MD-based numerical studies have been used to explore the concept of so-called *metamaterials* where material compositions are designed meticulously to enhance certain physical properties [6–11]. This is often achieved either by intertwining different material types or removing certain regions completely.

In recent papers by Hanakata et al. [6, 7] numerical studies have showcased that the mechanical properties of a graphene sheet, yield stress and yield strain, can be altered through the introduction of so-called *kirigami* inspired cuts into the sheet. Kirigami is a variation of origami where the paper is cut additionally to being folded. While these methods originate as an art form, aiming to produce various artistic objects, they have proven to be applicable in a wide range of fields such as optics, physics, biology, chemistry and engineering [12]. Various forms of stimuli enable direct 2D to 3D transformations through folding, bending, and twisting of microstructures. While original human designs have contributed to specific scientific applications in the past, the future of this field is highly driven by the question of how to generate new designs optimized for certain physical properties. However, the complexity of such systems and the associated design space make for seemingly intractable problems ruling out analytic solutions.

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

Hanakata et al. attribute the variety in yield properties to the non-linear effects arising from the out-of-plane buckling of the sheet. Since it is generally accepted that the surface roughness is of great importance for frictional properties it can be hypothesized that the kirigami cut and stretch procedure can also be exploited for the design of frictional metamaterials. For certain designs, we might hope to find a relationship between the stretching of the sheet and frictional properties. If significant, this could give rise to a control of friction behavior beyond manufacturing. For instance, the grasping robot might apply such a material as artificial skin for which stretching or relaxing of the surface could result in a changeable friction strength; Slippery and smooth when interacting with people and rough and firmly gripping when moving heavy objects. In addition, a possible coupling between stretch and the normal load through a nanomachine design would allow for an altered friction coefficient. This invites the idea of non-linear friction coefficients which might in theory also take on negative values given the right response from stretching. The latter would constitute a rarely found property. This has (only?) been reported indirectly for bulk graphite by Deng et al. [13] where the friction kept increasing during the unloading phase. Check for other cases and what I can really say here.

To the best of our knowledge, kirigami has not yet been implemented to alter the frictional properties of a nanoscale system. However, in a recent paper by Liefferink et al. [14](2021) it is reported that macroscale kirigami can be used to dynamically control the macroscale roughness of a surface through stretching which was used to change the frictional coefficient by more than one order of magnitude. This supports the idea that kirigami designs can be used to alter friction, but we believe that taking this concept to the nanoscale regime would involve a different set of underlying mechanisms and thus contribute to new insight in this field.

1.2 Goals

In this thesis we investigate the possibility to alter and control the frictional properties of a graphene sheet through application of kirigami inspired cuts and stretching of the sheet. With the use of MD simulations we evaluate the friction properties under different physical conditions in order to get insight into the prospects of this field. By evaluating variations of two kirigami inspired patterns and a series of random walk generated patterns we create a dataset containing information of the frictional properties associated with each design under different load and stretch conditions. We apply ML to the dataset and use an accelerated search approach to

optimize for different properties of interest. The subtask of the thesis are presented more comprehensively in the following.

1. Define a sheet indexing that allows for an unqiue mapping of patterns between a hexagonal graphene lattice representation to a matrix representation suited for numerical analysis.
2. Design a MD simulation procedure to evaluate the frictional properties of a given graphene sheet under specified physical conditions such as load, stretch, temperature etc.
3. Find and implement suitable kirigami patterns which exhibit out-of-plane buckling under tensile load. This includes the creation of a framework for creating variations within each pattern class. Additionally create a procedure for generating different styles of random walk patterns.
4. Perform a pilot study of a representative subset of patterns in order to determine appropriate simulation parameters to use for the further study along with an analysis of the frictional properties shown in the subset.
5. Create a dataset consisting of the chosen kirigami variations and random walk patterns and analyse data trends.
6. Train a neural network to map from the design space to physical properties such as mean friction, maximum friction, contact area etc. and evaluate the performance.
7. Perform an accelerated search optimizing for interesting frictional properties using the ML model. This should be done both through the pattern generation procedures and by following a genetic algorithm approach.
8. Use the most promising candidtes from the accelerated search to investigate the prospects of creating a nanomachine setup which exhibits a negative friction coefficient.
9. Study certain designs of interest with the scope of revealing underling mechanism. This includes simple correlation analysis but also a visualization of feature and gradient maps of the ML network.

Is the list of subtask to specific? Some of the details here might be better suited for the thesis structure section.

1.3 Contributions

What did I actually achieve

1.4 Thesis structure

How is the thesis structured.

Part I

Background Theory

Chapter 2

Machine Learning

In this thesis machine learning will serve as a numerical tool for evaluating and exploring the frictional behavior of various Kirigami designs. We will generate data using MD simulations which serve as a ground truth for the training of a machine learning model. If successful, we can utilize such a model to predict the frictional behavior of unseen configurations. The machine learning predictions will be a lot faster than carrying out a complete MD simulation and thus this can be used to accelerate the search through a new set of configurations. It is not obvious that the machine learning model can readily capture the physical mechanisms in our system. Hence, the attempt to model the system with machine learning also has value in terms of revealing the usefulness of such methods to this problem. We aim to implement a rather traditional machine-learning approach. In this chapter we introduce the key concept behind machine learning and some of the relevant concepts and techniques relevant to our implementation.

2.1 Neural network

The neural network, or more precisely the *feed forward dense neural network*, is one of the original concepts in machine learning arising from the attempt of mimicking the way neurons work in the brain [15, 16]. The neural network can be considered as three major parts: The input layer, the so-called *hidden layers* and finally the output layer as shown in Fig. 2.1. The input is described as a vector $\mathbf{x} = x_0, x_1, \ldots, x_{n_x}$ where each input x_i is usually denoted as a *feature*. For our task we will consider the Kirigami configuration, load and stretch of the system as input features on which we want the model to base its prediction. The input features are densely connected to each of the *nodes* in the first hidden layers as indicated by straight lines in Fig. 2.1. Each line represents a weighted connection that can be adjusted to configure the importance of that feature. Similar dense connections run through all the hidden layers to the final output layer. For a given note $a_j^{[l]}$ in layer l will process the input from layer $l-1$ 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 the choice of denoting this weight to belong to layer l as opposed $l-1$ is simply a notation choice. $b_j^{[l]}$ denotes a bias and $f(\cdot)$ the *activation function*. The activation function is often chosen to give a non-linear mapping of the input to each node. Without this, the network will only be capable of approximate linear functions [15]. Two common activation functions are the *sigmoid*, mapping the input to the range $(0, 1)$, and the ReLU which cuts off negative values and maps positive linearly

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

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 *vêcy*. In order to get useful predictions we must *train* the model which involves tuning the model parameters, the weight and biasses.

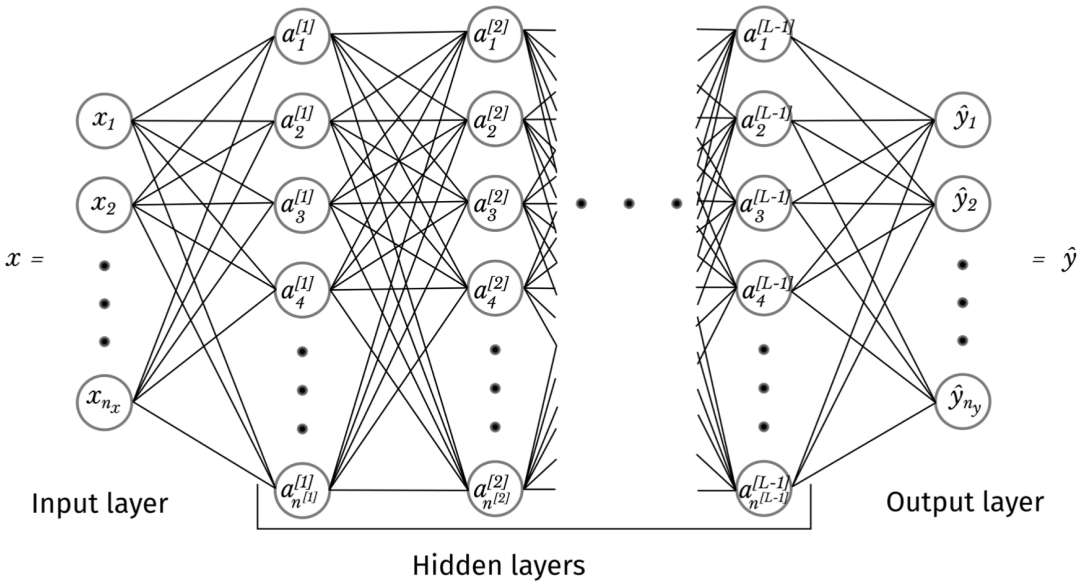


Figure 2.1: From overleaf IN400

The model training relies on two core concepts: *backpropagation* and *gradient descent* optimization. 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 true value $\mathbf{y}$. For a continuous scalar output, we might simply use the mean squared error (MSE)

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

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

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

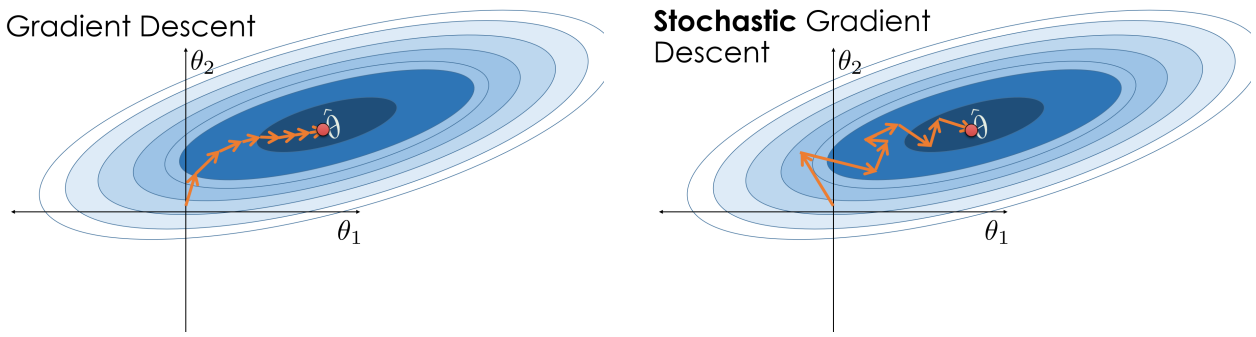
The cross-entropy loss can be derived from a maximum likelihood estimation SOURCE. Without going into details with the derivation we can convince ourselves that the error is minimized for the correct prediction and maximized for the worst prediction. When $y_i = 1$ we get the negative term $-\log(\hat{y}_i)$ 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 gradient expresses how each parameter is connected to the loss. The overall idea is then to "nudge" each parameter in the right direction. We generally denote a full cycle of forward and backpropagation and an update to the parameters as an epoch. We calculate the updated parameter θ_t for epoch t using gradient descent

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

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 analog with only two parameters θ_1 and θ_2 we can think of this as the act

of stepping perpendicular to the contour lines shaped by the loss function as shown in Fig. 2.2a. Notice however that models in general contain on the order of $10^6 - -10^9$ parameters FACT CHECK, but this might get a bit harder to visualize. The length of each step is proportional to the gradient magnitude 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 whole dataset by averaging the contribution from each data point before updating the parameters. This gives the most robust estimate of the gradient and thus the most direct path through parameter space in terms of minimizing the loss function as indicated in Fig. 2.2a. However, for big datasets, this calculation can be computationally heavy as it must carry the entire dataset in memory at once. A solution to this issue is provided by stochastic gradient descent (SGD) which considers only one data point at a time. Each data point is chosen randomly and the parameters are updated based on the corresponding gradient. This leads to more frequent updates of the parameters which will results in a more "noisy" path through parameter space with respect to minimizing the loss as shown in Fig. 2.2b. In some situations, this might compromise the precision but the noisiness makes it more likely to escape local minima in the loss 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 2.2: TMP

2.2 Optimizers

The name *optimizers* covers a variety of gradient descent methods. In our study, we will use the ADAM (adaptive moment estimation). 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

$$\theta_t = \theta_{t-1} - m_t, \qquad m_t = \alpha m_{t-1} + \eta \nabla_\theta L(\theta_t) \tag{2.1}$$

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

$$\begin{aligned} m_1 &= \alpha m_0 + \eta g_1 = \eta g_1 \\ m_2 &= \alpha m_1 + \eta g_2 = \alpha^1 \eta g_1 + \eta g_2 \\ m_3 &= \alpha m_2 + \eta g_3 = \alpha^2 \eta g_1 + \alpha \eta g_2 + \eta g_3 \\ &\vdots \\ m_t &= \eta \left(\sum_{k=1}^{t} \alpha^{t-k} g_k \right). \end{aligned} \tag{2.2}$$

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 loss space. 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. (2.2), but with the explicit weighting by $(1-\alpha)$.

The second moment of the exponential moving average is utilized in the root mean square propagation (RMSProp) method which is motivated by the issue of passing long plateaus in the loss 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 dividing with the norm $|||g_t|$. However, this does not immediately solve the problem of long plateaus as we need to consider multiple past gradients which is then accommodated by the use of the EMA. When reentering a steep region again we need to "quickly" downscale the gradient steps again which can be achieved by using the squared norm $||g_t||^2$ for the EMA which makes it more sensitive to outliers. The RMSProp update scheme then becomes

$$\theta_t = \theta_{t-1} - \eta \frac{g_t}{\sqrt{\text{EMA}(||g_t||^2)} + \epsilon}, \tag{2.3}$$

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. (2.3)

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

Since these are initially set to zero we can correct a bias towards zero by a scaling term $(1-\beta_1^t)$ and $(1-\beta_2^t)$ respectively such that the ADAM scheme becomes

$$\theta_{t+1} = \theta_t - \eta \frac{\hat{m}_t}{\sqrt{\hat{v}_t} + \epsilon}, \qquad \hat{m}_t = \frac{m_t}{1-\beta_1^t}, \qquad \hat{v}_t = \frac{v_t}{1-\beta_2^t}. \tag{2.4}$$

2.2.1 Weight decay

By adding a so-called *regularization* to the loss function we can penalize high magnitudes of the model parameters. 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 new loss and gradient then become

$$\begin{aligned}
L_{l^2}(\theta) &= L(\theta) + \frac{1}{2}\lambda||\theta||_2^2 \\
\nabla_\theta L_{l2}(\theta) &= \nabla_\theta L(\theta) + \lambda\theta,
\end{aligned} \tag{2.5}$$

where λ is the weight decay parameter $\in [0, 1]$. The name *weight decay* relates to the fact that some practitioners only apply this penalty to the weights in the model, but we will include the biases as well (standard in PyTorch). Following the original gradient descent scheme Eq. (2.5) we get

$$\begin{aligned}
\theta_{t+1} &= \theta_t - \eta g_t - \eta\lambda\theta_t \\
&= \theta_t \underbrace{(1-\eta\lambda)}_{\text{Weight decay}} - \eta g_t.
\end{aligned}$$

Thus we notice that choosing a high weight decay (towards 1) will downscale the model parameters while choosing a low weight decay (towards 0) yields the original gradient descent scheme. Note that this is used in combination with ADAM, but it is easier to show the consequences for the original gradient descent scheme.

2.2.2 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 values we get $f(z \to \pm\infty) \to 0$. However, even a small gradient will vanish throughout a deep network as the calculation of the gradient involves the chain rule. A similar problem can be found with the ReLU activation function which gets a zero gradient for inputs $z < 0$ which can be mitigated by the so-called leaky RelU which gives the $z < 0$ a small slope. 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 and sometimes result in numerical overflow. While there exist techniques to accommodate this, 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 to other layers. This balanced gradient flow is more likely to happen if we initialize the weight by the same set of criteria. The specific actions to achieve this depend on model architecture, including the choice of activation functions. For instance, using the ReLU activation functions it was found that the single standard deviation will depend on the number of input nodes from the previous layer $N^{[l-1]}$ as $\sim \sqrt{N^{[l-1]}}/\sqrt{2}$ and thus we simply scale a zero mean uniform distribution to match this. This is part of the Kaiming initialization scheme which is standard in Pytorch SOURCe. Mention choices for bias initialization.

Batch normalization is another technique that might also help reduce the issue of poor gradient flow. Furthermore, can benefit by speeding up convergence making it 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 result 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 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.

2.2.3 Learning rate decay strategies

Until now we have assumed a constant learning rate, but variations use a changing learning rate beyond the adaptiveness included in the optimizers as described earlier. 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. 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. This can simply be done by a linear increase and decrease, but we choose a cosine function that is shifted to peak for the first 30% of the training length.

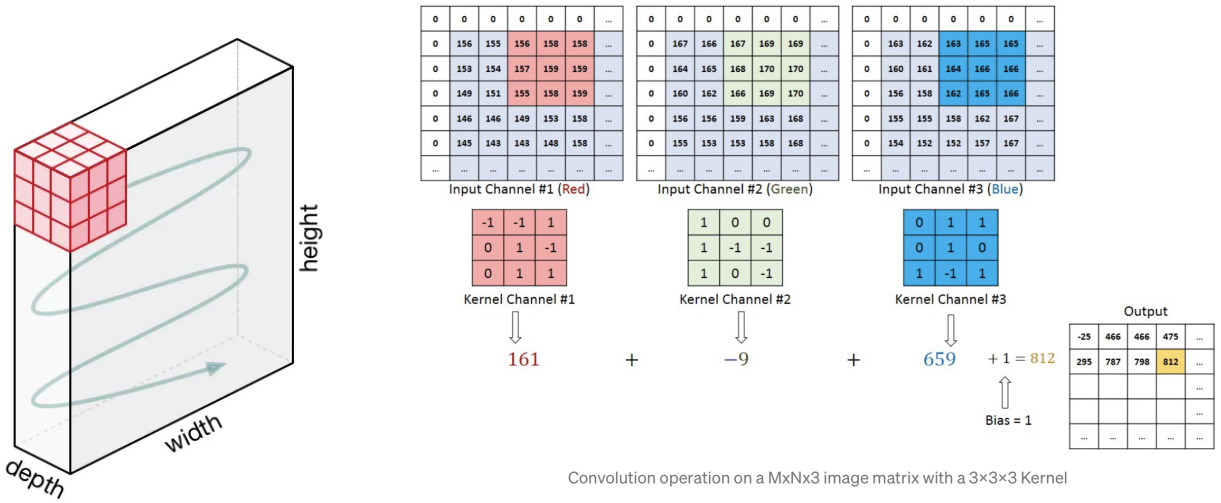
2.3 Convolutional Neural Network

Convolutional Neural Networks (CNNs) build upon some of the same concepts as introduced with the feed-forward neural network. 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. Thus, it does not matter how the input is arranged initially, but it cannot be changed at a later stage. This is impractical if we want the model to recognize images of animals as the animal-related pixels will be in different regions of each image. The CNN can be motivated by the idea of capturing spatial relations, but without being sensitive to the relative placement within the image, 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. 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*. The kernel has a 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. The kernel lines up with the image and calculates the feature map output as a dot product between the weights in the kernel and the aligning subset of the input. This is done for each input channel and summed up with the addition of a bias as illustrated in Fig. 2.3b. The kernel then slides over by a step size given by the *stride* model parameter and repeat the calculation. Choosing a stride of 2 or higher results in a reduction of 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

$$N_d^{[l+1]} = \left\lfloor \frac{N_d^{[l]} - F_d + 2P}{S} + 1 \right\rfloor, \tag{2.6}$$

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 remniscent of a kernel, but instead of calculating the output as a dot product, it calculate the mean (mean pooling) or the max value (max pooling) of the values within its range. 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. (2.6). CNNs will often use repeating series of convolution, pooling and then an activation function. Most architectures will down-sample the spatial input while increasing the number of channels in the model.

[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.



(a) (b)

Figure 2.3: TMP

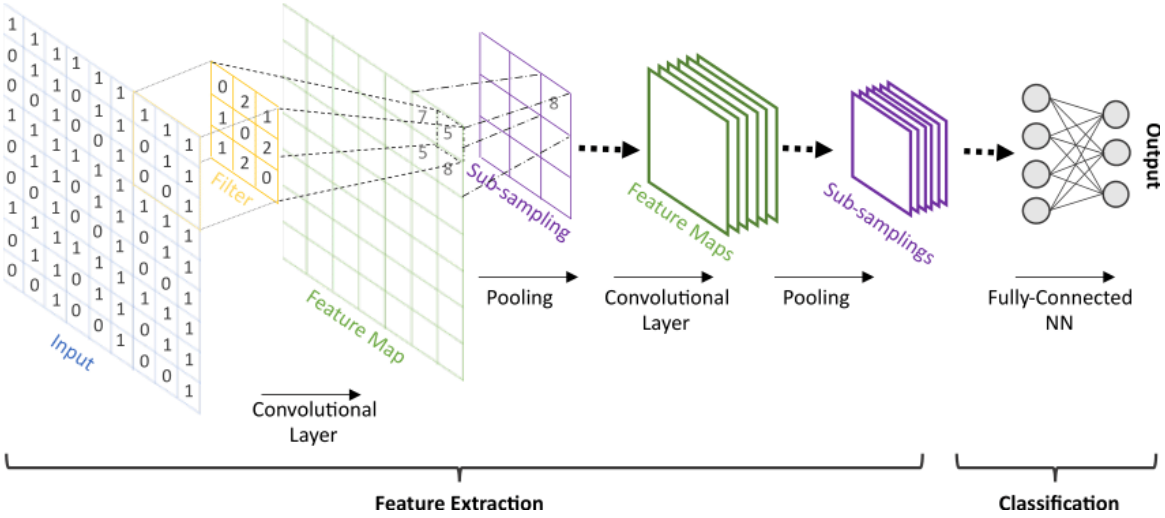


Figure 2.4: TMP Not used at the moment...

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. In Fig. 2.5 the receptive field is illustrated for a 1D representation of a CNN with repative use of a kernel of with 2 and stride 1. Going from the output and backward, we see that the output layers are connected to two nodes in the previous layer. Each of these nodes is connected to two nodes in the layer before that, however with one of them 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 due to the fact that these nodes in the center of the receptive field will have more connections leading to the output, as seen in Fig. 2.5. Thus, in practice the effective receptive field will be smaller than the theoretical Implementations like dilated convolutions, which make the filter expand in circumference and skipping skipping positions within the filter, which can be used to further increase the effective field.

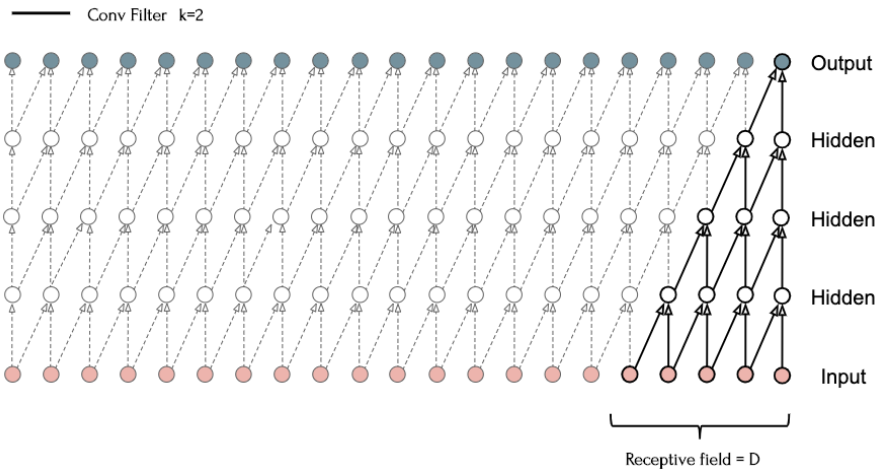


Figure 2.5: TMP

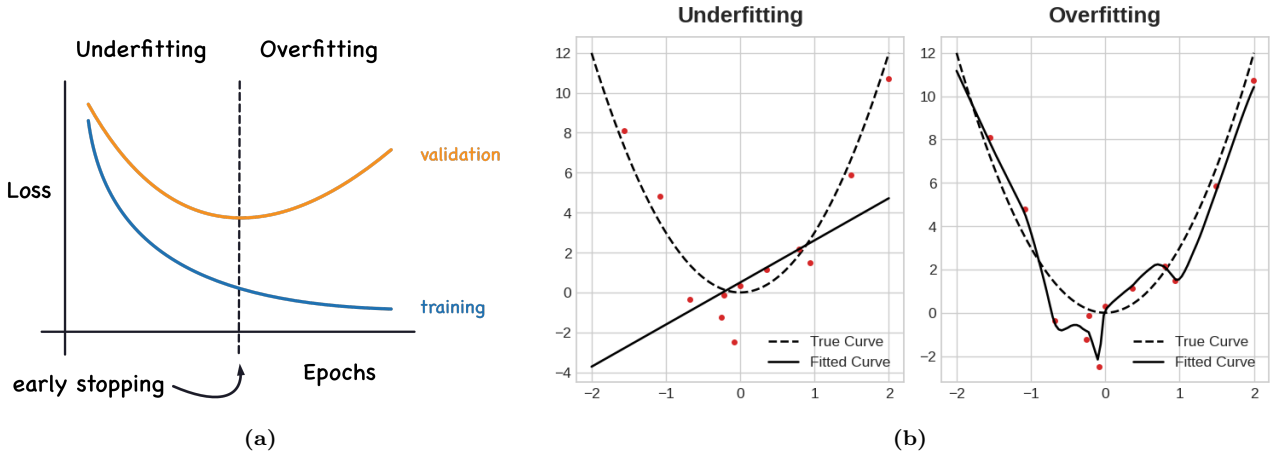
On a final note regarding the CNN we point out that convolution is often used in combination with a dense network, or *fully connected*, at the end. We can then think of the convolution part to handle the translation from a spatial import to some internal features. For the animal detection network, we would perhaps suggest features like, 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. We discuss one approach for interpreting of the model internals in Sec. 2.6

2.3.1 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 is it improves. If a model has enough complexity, i.e. enough layers with a sufficient amount of nodes, it can be proven that it can fit any function SOURCE. Thus for a complex model, it is just a matter of time before the model eventually will learn to give matching predictions for the training data. However, we want the model to learn general trends and not to "memorize" all the data points which is known as overfitting. While the predictions can get arbitrarily good the performance on unseen data within the same task will yield awful results if the model is overfitted. Hence, we put aside some subset of the data which we use as a *validation* throughout and after training. By keeping this *validation* set separate from the training data we can get a more reliable performance estimate for the model. This splitting of data should be done randomly in order to ensure a similar distribution of data features in each set. For the size ratio we want to strike a balance between training quality and validation quality, usually a 20%–80% split favoring the training set is a good figure. Other techniques exist which aim to optimize the data use for sparse data situations, like cross-validation and bootstrap fact check, but we will not consider such methods for this thesis. A third training set which is often forgoten 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 like learning rate, momentum and weight decay, we will use the performance on the validation set as a metric to steer by. This represents a slower pace parameter fitting which can also led to overfitting on a higher level. Hence, we should denote a test set for the final evaluation of our model which have not be considered before the end. Formally, this is the only reliable performance metric for the model.

2.4 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 such as learning rate, momentum and weight decay. The textbook visualization of underfitting and overfitting is shown in Fig. 2.6a. 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 to specific trends which marks the transition into overfitting where the validation loss will start to increase again. Notice that the performance on the training data will keep increasing. *Early stopping* can be utilized to detect this transition and stop the training. We will use a variation of this which is based on storing the model parameters at the best validation performance but letting the model training go on. We can imagine a similar curve as seen in Fig. 2.6a with the replacement of *model complexity* on the x-axis. For a certain amount of epochs a simple model will yield underfitting and an overly complex model will yield overfitting. In figure Fig. 2.6b we see how under- and overfitting can be represented by a fitting to a 2D curve. We see how a simple, not complex, underfitted model will make a crude approximation for the true curve. An overly complex overfitted model will follow pick up the noise in the training data and miss the general trend. In practice the diagnosticating of under. and overfitting is not as simple as the figures in Fig. 2.6 imply. First of all, the Fig. 2.6a is simplifying the idea, and the training and validation loss will rarely showcase such a clear indication of the transition to overfitting. The problem with Fig. 2.6b lies in the fact that we do not know the true curve. If we did, we would not need machine learning to estimate 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.



(a) (b)

Figure 2.6: TMP

2.5 Hypertuning

Define hyper-parameter probably like done in [17]

Training a machine learning model revolves around tuning the model parameters such as weights and biases. However, a handful of so-called *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 an a functional model more complicated that simply starting the traiing. As N. Smith [18] 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-parameter based mainly on the findings of [18]. The traditional approach is to perform a grid-search of hyper-parameter combinations for multiple 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.

The learning rate is often regarded as the most important hyper-parameter to tune [17]. Typical values are in the range $[10^{-6}, 1]$. Instead of simply running a grid search, i.e. multiple full training sessions using different learning rates, we can perform a so-called *learning rate range test* (LR range test). One 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 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 [19]. We choose 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 behaviour can be understood for the simplified 1D example of finding the minima of a second-order polynomial as shown in Fig. 2.7. Small learning rates will step in the right direction, but for overly small values this will simply result in a slow convergence. If the learning rate becomes too large, we will effectively step past the minemim. For each step we will overshoot the minimum more and more 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.

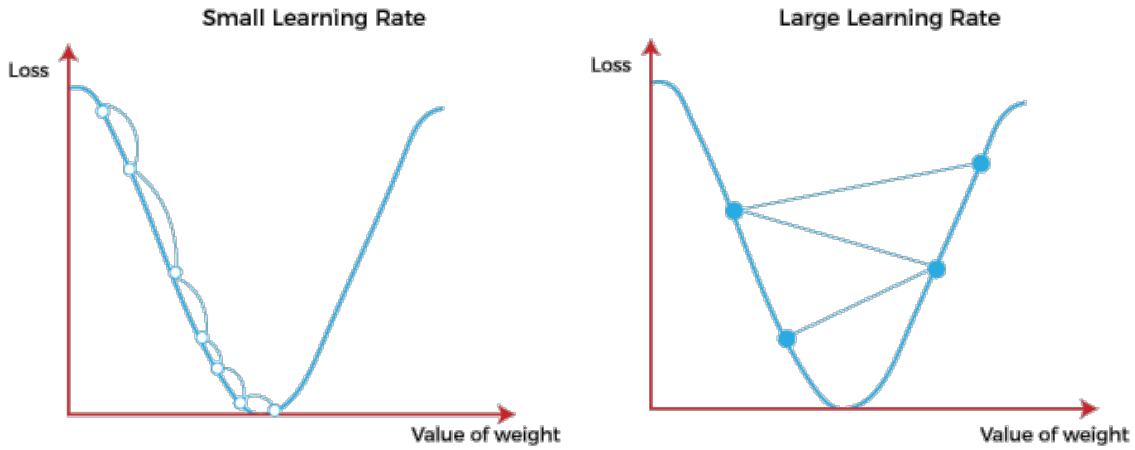


Figure 2.7: TMP

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

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

as the 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 [18] 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. 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.

Finally, we address the weight decay. N. Smith 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 learning rate and momentum choice and this can often be dialed in after setting thoose. Fuirthermore, the weight decay will be dependent on the model complexity. 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.

2.6 Prediction explanation

With the rise of constantly improving machine learning models and computational power, the capabilities of machine learning systems is improving as well. However, with deep learning models, we have essentially no insight into the considerations that went into a prediction other than 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 graident based method called *Grad-CAM* [20]. The idea is to 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 target output. For a classification task, one would often choose the predicted class as the target output, the class with the highest score. The gradients will then highlight which part of the feature maps is most important for the prediction loss. a ReLU activation layer is then applied to only highlight the positive contributions. Since the convolutional layers preserve relative spacing we can rescale the heatmap provided by the feature map gradient to make an input-sized heatmap we can overlay with the input image. This provides a visual clue of what the prediction is based on. We can do this for different depths of the model and even combine the results for multiple layers. In Fig. 2.8 we see how this can be used to reveal the basis for a prediction. In this case, the Grad-CAM analysis shows the difference between a biased and unbiased model for the predictions of professions. The biased model shows to be considering the person more than the actually objective clues given by relevant equipment and work-related clothing

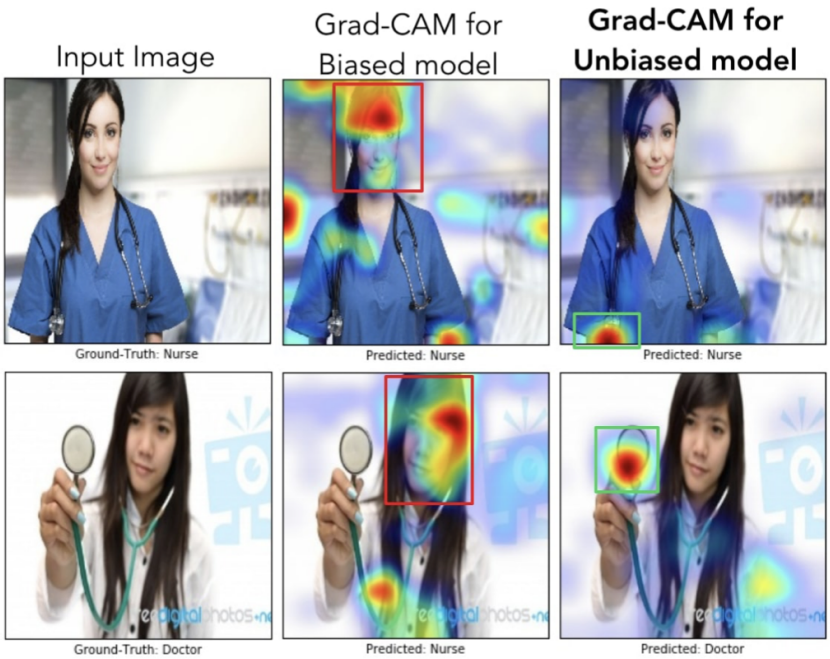


Fig. 8: In the first row, we can see that even though both models made the right decision, the biased model (model1) was looking at the face of the person to decide if the person was a nurse, whereas the unbiased model was looking at the short sleeves to make the decision. For the example image in the 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 2.8: TMP [20]

2.7 Accelerated search using genetic algorithm

2.7.1 Markov-Chain Accelerated Genetic Algorithms

2.7.1.1 Talk about traditional method also?

2.7.1.2 Implementing for 1D chromosone (following article closely)

We have the binary population matrix $A(t)$ at time (generation) t consisting of N rows denoting chromosones and with L columns denoting the so-called locus (fixed position on a chromosome where a particular gene or genetic marker is located, wiki). We sort the matrix rowwise by the fitness of each chromosono evaluated by a fitness function f such that $f_i(t) \leq f_k(t)$ for $i \geq k$. We assume that there are a transistion probability between the current state $A(t)$ and the next state $A(t+1)$. We consider this transistion probability only to take into account mutation process (mutation only updating scheme). During each generation chromosones are sorted from most to least fitted. The chromosone at the i-th fitted place is assigned a row mutation probability $a_i(t)$ by some monotonic increasing function. This is taken to be

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

for some limit N' (refer to first part of article talking about this). We use $N' = N/2$. We also defines the survival probability $s_i = 1 - a_i$. In thus wau a_i and s_i decide together whether to mutate to the other state (flip binary)

or to remain in the current state. We use s_i as the statistical weight for the i-th chromosone given it a weight $w_i = s_i$.
Now the column mutation. For each locus j we define the count of 0's and 1's as $C_0(j)$ and $C_1(j)$ resepctively. These are normalized as

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

These are gathered into the vector $\mathbf{n}(j,t) = (n_0(j,t), n_1(j,t))$ which characterizes the state distribution of j-th locus. In order to direct the current population to a preferred state for locus j we look at the highest weight of row i for locus j taking the value 0 and 1 respectively.

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

which is normalized again

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

The vector $\mathbf{n}(j,t+1) = (n_0(j,t+1), n_1(j,t+1))$ then provides a direction for the population to evolve against. This characterizes the target state distribution of the locus j among all the chromosones in the next generation. We have

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

Since the probability must sum to one for the rows in the P-matrix we have

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

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

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

We just need to know $P_{00}(j,t)$. We start from $P_{00}(j,t=0) = 0.5$ and then choose $P_{00}(j,t) = n_0(j,t)$

2.7.1.3 Repair function

Talk about it here or in random walk section?

Part II

Simulations

Appendices

Appendix A

Appendix B

Appendix C

Bibliography

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

[17]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.

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

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

[20]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).