Theory and Applications of Gradient Free Optimization in Physics

VIKTOR SELIN

Thesis submitted to the University of Ottawa in partial fulfilment of the requirements for the degree of

Master of Science

in

PHYSICS

Department of Physics
Faculty of Science
University of Ottawa

© Viktor Selin, Ottawa, Canada, 2024

Contents

Acronyms iv

List of Figures v

Acknowledgements viii

Abstract ix

Thesis Outline x

Thesis Introduction x

1 MC-GD Correspondence 1

1.1 Introduction . . . 1

1.1.1 Machine learning . . . 1

1.1.2 Gradient descent . . . 7

1.2 Methods . . . 9

1.2.1 Neuroevolution . . . 9

1.3 Results . . . 10

1.3.1 Finite β correspondence derivation . . . 10

1.3.2 Finite β correspondence . . . 12

1.3.3 Infinite β correspondence derivation . . . 15

1.3.4 Infinite β correspondence . . . 16

1.4 Discussion . . . 25

1.5 Conclusion . . . 25

2 Adaptive Monte Carlo 26

2.1 Introduction . . . 26

2.2 Methods . . . 26

2.2.1 Adaptive gradient optimization . . . 27

2.2.2 Adaptive Monte Carlo . . . 29

2.3 Results . . . 30

2.3.1 Implementation of first version . . . 30

2.3.2 Implementation of second version . . . 34

2.4 Discussion . . . 44

2.5 Conclusion . . . 44

3 **Growth Pipeline for Targeted Design** **45**
3.1 Introduction . . . 45
3.2 Methods . . . 46
3.2.1 Growth pipeline . . . 46
3.2.2 Growth model . . . 46
3.2.3 Machine learning . . . 54
3.2.4 FDTD simulation . . . 55
3.3 Results . . . 59
3.3.1 Growth simulation comparison . . . 59
3.3.2 Mean cluster size target pipeline . . . 61
3.3.3 FDTD test . . . 67
3.3.4 Growth for FDTD simulations . . . 69
3.3.5 3D photonic chips . . . 77
3.3.6 Scoring using FDTD simulations . . . 81
3.3.7 Full pipeline test . . . 85
3.3.8 Implementing spatial control methods. . . . 94
3.4 Discussion . . . 99
3.5 Conclusion . . . 100

Thesis Conclusion **100**

Appendix **106**

A Antenna Growth Objective **106**

B Code **108**
B.1 MC-GD correspondence. . . . 108
B.2 aMC. . . . 113
B.3 Growth Pipeline. . . . 124

Acronyms

aMC adaptive Monte Carlo. 4, 5, 33–35

CPU Central Processing Unit. 17

FCNN Fully Connected Neural Network. 12, 21

GD Gradient Descent. 3–5, 9, 13, 16–29, 31–35

GPU Graphical Processing Unit. 11

MAE Mean Absolute Error. 13

MC Monte Carlo. 3, 4, 9, 13–29, 33, 35

ML Machine Learning. 8–11, 29, 31

MSE Mean Square Error. 13, 17

NN Neural Network. 8, 9, 11, 12, 15–17, 26

ReLU Rectified Linear Unit. 12

RNN Recurrent Neural Network. 25

STD Standard Deviation. 5, 34, 35

List of Figures

1.1 Example of a regression task. 2
1.2 Schematic of a FCNN. 3
1.3 Common activation functions used in ML. 6
1.4 Difference in learning using GD due to different learning rate magnitudes. 8
1.5 Correspondence for finite β. 13
1.6 Finite β absolute mean parameter difference. 14
1.7 Single parameter correspondence evolution with reset. 15
1.8 Correspondence for infinite β. 17
1.9 Prediction for GD-MC infinite beta correspondence. 18
1.10 All parameters in a depth 8, width 32 network after initialization. 19
1.11 All parameters in a depth 8, width 32 network after training. 19
1.12 Acceptance plot for MC trajectories for a depth 8, width 32 network. 20
1.13 Loss plot for GD used after MC. 20
1.14 Alternating GD-MC trained on $sin(\pi x)$. 21
1.15 Alternating GD-MC trained on $sin(4\pi x)$. 22
1.16 GD and MC minimizing a smooth pseudo-loss surface. 23
1.17 GD and MC minimizing a pseudo-loss surface consisting of plateaus. 24
1.18 GD and MC minimizing a near zero gradient pseudo-loss surface. 24
2.1 High and low frequency fitting using adaptive GD. 28
2.2 Mutation adaptivity for first version of aMC. 30
2.3 Behaviour of first version of aMC on a smooth pseudo-loss surface. 32
2.4 Behaviour of first version of aMC on a linear pseudo-loss surface. 33
2.5 Mutation adaptivity for second version of aMC. 34
2.6 Results of aMC, Adam and GD trained on three different functions. 37
2.7 32-bit and 64-bit floating point numbers test for aMC, Adam, and GD trained on $sin(2\pi x)$. 38
2.8 Architectural layout for a mushnet. 39
2.9 Results for mushnet of depth 8 trained on a square wave function. 40
2.10 Results for mushnet of depth 32 trained on a square wave function. 41
2.11 Mushnets of depth 4, 16, 32, 64, 128 trained on the square wave function. 42
2.12 Acceptance based on loss for MC and aMC. 42
2.13 Acceptance as a function of the number of parameters. 43
3.1 Schema for controlled growth pipeline. 46
3.2 Snapshots of the growth simulation for $k_BT = 0.2$, $Frac = 0.2$. Nanoparticles are shown in yellow while solvent/air is shown in black. 47

3.3 Finished growths produced with varying neighbour bonds. 48
3.4 Lattice gas model. 49
3.5 Bonds affected by a nanoparticle movement. 52
3.6 Central difference point for magnetic field propagation of Maxwell's equations in FDTD. . 57
3.7 Central difference point for electric field propagation of Maxwell's equations in FDTD. . . 58
3.8 Replicated growths using the same growth parameter as for the reference growths. 60
3.9 Mean cluster size test results for growth pipeline with target cluster size of 100. 62
3.10 Mean cluster size test results for growth pipeline with target cluster size of 500. 63
3.11 Policies of the mean cluster size 500 test. Green is best score while black is worst. 64
3.12 Constrained network output for temperature and chemical potential. 64
3.13 Sample of network policies after N accepted mutations. 65
3.14 Mean cluster test with penalty for cluster size standard deviation. Target cluster size of 900. 66
3.15 Policies for the standard deviation penalty test. 67
3.16 Lumerical target shape test. 68
3.17 Snapshots of the electric field magnitude for the Lumerical test. 69
3.18 Non-periodic boundary conditions for growth model. 70
3.19 Example of grown structure acting as a photonic chip. 71
3.20 Example of growth used as a photonic chip with extended growth region. 71
3.21 Growth with solvent in equilibrium. 72
3.22 Input flux measurement comparison between a growth and an empty waveguide. 73
3.23 Resolution example for a filled box growth. Resolution is 40 $pixels$/µm 73
3.24 Transmission for filled box chip with varying light source width. Each plot corresponds to one of the outputs. 74
3.25 Resolution test for the filled box chip with different resolutions. 75
3.26 Difference and mean difference of the transmission for the output probes of the filled box chip. 75
3.27 Growth used for resolution test. 76
3.28 Resolution test for grown chip. 76
3.29 Electric field for three different slices during a FDTD simulation. 77
3.30 Example of a 2 dimensional chip transformed to a 3 dimensional photonic chip. 78
3.31 Replicated transmission spectra for chip from [1]. 79
3.32 Implementation of epsilon function to define photonic chip structure. 80
3.33 Comparison between 3-dimensional array and epsilon function implementation for the transmission on a test chip. 81

3.34 Examples of growths with constant growth parameters for different μ. 82
3.35 Scoring method for growth pipeline. 82
3.36 Evolution of electric field for three different growths with different chemical potential μ. . 83
3.37 The best scoring growths for the constant growth parameter scoring test. 84
3.38 Score distributions for all 200 growths with $\mu = -2.5, -2.6$. 84
3.39 Growth heat map for score test with constant parameters. 85
3.40 Full pipeline score evolution. 86
3.41 Random selection of growths produced by network policies for different epochs. 86
3.42 Results of the MC pipeline test with 40 growths. 87
3.43 Random selection of growths from different epochs during training. 88
3.44 Score distributions for 40 and 240 growths. 88
3.45 Results of the MC pipeline test with 520 growths. 89
3.46 Score distributions for 3 sequential epochs for the 520 growth MC pipeline test. 89
3.47 Examples of potential transmission based objectives. 90
3.48 Pipeline results for even split objective. 91
3.49 Bandpass filter test with varying mutation standard deviation σ. 91
3.50 Pipeline for piepeline with MSE bandpass scoring. 92
3.51 Transmission plots for MSE bandpass score test. 93
3.52 Growth for three different epochs from the MSE bandpass test. 93
3.53 Score distributions for the MSE bandpass test. 94
3.54 Point attraction implementation. 95
3.55 Example of line attraction. 96
3.56 Score evolution for pipeline with linear attraction. 96
3.57 Policy evolution for pipeline with linear attraction. 97
3.58 Random selection of growths from the first epoch, best scoring epoch and final epoch. . . 97
3.59 Score distributions for pipeline with linear attraction. 98
3.60 Transmission of best performing policy for a number of growths. 98
3.61 Manually designed structures with the corresponding transmission and score. 99
A.1 Antenna objective with different source light directions. 107

Acknowledgements

I would like to start by thanking my supervisor Dr. Isaac Tamblyn. I would also like to thank Dr. Stephen Whitelam and everyone in the CLEAN research group. In addition, I would like to thank Vector Institute and the Digital Research Alliance (formerly Compute Canada) for access to compute resources. Finally, I would like to thank the National Research Council (NRC) for office space and additional compute resources.

Abstract

Machine Learning (ML) has become a popular field of research in many domains. It has become a flexible option to tackle a large variety of problems. This Thesis examines a fundamental component of ML training to explore how these tools can be further used in physics. The gained knowledge is then used for a physics inspired inverse design problem. This is done in three separate projects, the first explores gradient and non-gradient based learning, the second introduces adaptivity, and the final uses these concepts to learn how to grow photonic chips. My contributions for these projects includes the implementation, producing results and plot creations.

Thesis Outline

This Thesis is divided into three separate sections which describe my contributions to three projects. The necessary background information is explained for each project before the results are presented. The projects are presented in chronological order, which means that the necessary background information is often described in the preceding project(s) as each inspired the next. Because of this, if one is interested in reading the second or third project only, it might be of interest to read the **Introduction** and **Methods** section of the previous project(s).

The overarching goal of these projects was to improve Machine Learning applications in physics. The first two projects therefore seek to investigate the Machine Learning methods and offer an alternative to the popular gradient based optimization. The focus in these project is numerical optimization. We sought to better understand how ML learns so as to be able to apply it more effectively in a physics context. The third project makes used of the gained knowledge for a design problem. This project tackles the challenging problem of inverse design with added complexity through the use of an empirically accurate growth model as well as a physical simulator to score the growths based on some objective.

Thesis Introduction

Gradient based methods are wildly popular for training Neural Networks in Machine Learning, but are not the only way. The first part of this thesis numerically demonstrates an analytical correspondence between two optimization algorithms: Gradient Descent and the non gradient-based MC. The analytical derivation is then numerically tested by training networks using both methods, and comparing the parameter evolutions. Building on this result, correspondence between the non-gradient and gradient optimizer gives the possibility of improving MC by introducing adaptivity, a feature commonly used in gradient based optimizers. This allows the algorithm to compete with a widely used adaptive Machine Learning optimization algorithm Adam. The two adaptive algorithms are compared and tested under different circumstances, with our adaptive MC algorithm showing performance similar to Adam. This result shows the potential of non-gradient optimization in ML.

The final project introduces a Machine Learning pipeline for the generation of desired structures using a growth model to fulfil some objective. The growth is done in an empirically realistic nano-particle lattice gas simulation. A Neural Networks is then trained to manipulate the growth simulation parameters during growth to control the generated structure. This network is trained using MC using a score calculated from the transmission spectrum of light simulated passing through the growth. The objective of the pipeline is to generate structures which perform well as photonic chips, for tasks such as wavelength splitting.

1 MC-GD Correspondence

1.1 Introduction

Machine Learning (ML) has gained significant traction over the years in a wide variety of domains. In the most public facing advancements, it has found particular success in generation tasks. Text generation has been popularized by the widely used OpenAI GPT models [2, 3]. These large models are able to produce context appropriate text with much higher quality than was previously possible. Image generation has similarly entered public perception through the likes of stable diffusion [4] or DALL-E [5], which allows direct text to image generation. In a similar vain, ML has also found a place in scientific research as a useful tool in a multitude of fields, such as the predictions of complex molecules [6, 7], proposal of novel materials [8–10], and as surrogate models for expensive computations [11, 12]. In contrast to the massive models used in industry, usually these smaller models are implemented to either test certain ML architectural advancements or to accomplish a specific task.

In all these examples, an important component sometimes overlooked when looking at the final model is the optimizer - the algorithm in charge of updating the parameters of the Neural Network (NN). Improvements as well as gaining understanding on how these function is evidently incredibly important to improve model performance and reduce the necessary training time.

In this first project, a correspondence is analytically derived between two optimizers: Gradient Descent (GD) as well as a simple Metropolis Monte Carlo (MC) algorithm often used in physics simulations. These algorithms are detailed in the methods section. Numerical simulations are used to show the correspondence in a simple ML regression problem.

Contributions - The contributions are separated as follows: the analytical proof of correspondence was derived by Dr. Stephen Whitelam. All numerical results presented below, as well as the code needed to produce the necessary data and simulations was implemented and run by me.

1.1.1 Machine learning

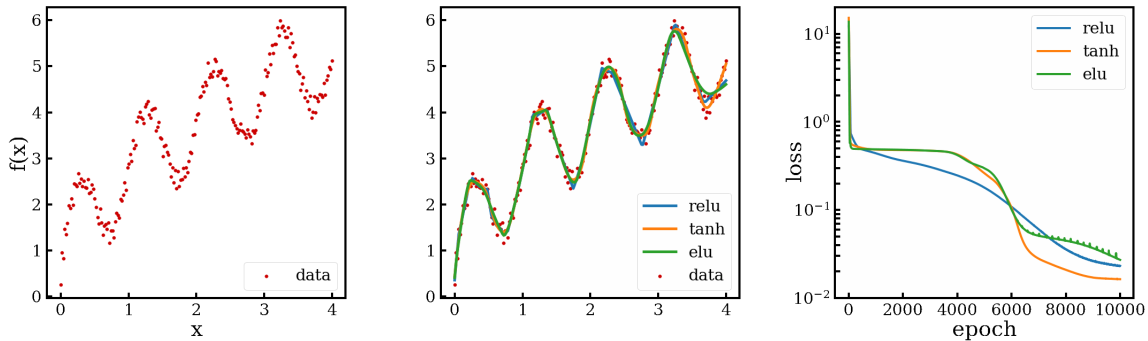
ML is a learning algorithm which learns from data. Learning in this context is defined by the ability to improve its performance with respect to some task [13]. To fully understand ML, we need to understand what the task is, how the performance is calculated, and finally, how it uses data to improve at the task.

Task - ML can be used for a multitude of different tasks. Two important types are that of classification and regression. Classification involves the process of separating inputs into k different categories. More precisely, it needs to learn a function f which maps $\mathbb{R}^n \rightarrow 1, .., k$, where n is the number of parameters

of the function. Usually this is done by outputting a probability of the input belonging to each of the k categories.

Regression tasks involves the learning of a mapping from input to output $\mathbb{R}^n \to \mathbb{R}$, which is usually a single number. In essence, it is learning to fit a function. All projects in this thesis involve this type of task. Figure 1.1 shows an example of a NN learning how to fit a noisy sinusoidal function $f(x) = sin(2\pi x) + sqrt(6x) + N$, where N is random Gaussian noise added to the training samples. The network is given a x value and outputs the corresponding $f(x)$. In this test, three different activation functions are used, these will be explained further in a later section. The quality of the fit is measured using the loss, a value used to quantify the network's performance. In this example the mean squared error is used. A lower loss signifies a better fit. The evolution of the loss shows how the network improved as a function of training step, labelled epoch.

This is an example of a supervised ML task, which uses labelled training data, that is, known input-output samples to learn. In contrast, unsupervised learning does not have labelled training data, and instead wants to learn how to separate the input data, discovering patterns in the data. While all tests in this project were done using supervised learning, correspondence between GD and MC still holds for unsupervised learning.



(a) Noisy training data of a sinusoidal function.

(b) Final fit of trained network with varying activation functions.

(c) Loss evolution of training.

Figure 1.1: Example of a regression task consisting of learning a noisy sinusoidal function. Gradient descent was used to train the network with varying activation functions: ReLU, Tanh, ELU.

Performance - The performance of the ML algorithm is measured by calculating a value known as the loss. What specific type of loss to use is dependent on the task. In general however, it relates to how well the algorithm is predicting the correct output. The exact loss used in this project is further explained in the methods section.

Learning from data - The two most common ways data is used in ML is unsupervised and supervised learning. These methods present the data to the ML algorithm in two different ways, and are thus suitable

for different tasks. In general, the data is organized into a dataset for training. The data samples used as inputs are called features. As learning is an iterative process, the ML algorithm will see the dataset many times. A complete pass of the entire dataset is called an epoch. Controlling the number of epochs therefore determines how long the algorithm will train for.

In unsupervised learning, the features are shown to the ML algorithm so it can learn useful relations relating to the task. For example, learning to divide the dataset into a number of groups (clustering), or learning the distribution that generates the dataset to produce more examples. This is useful for synthesis tasks, such as image generation.

For supervised learning, data in the dataset is instead paired with a target, known as a label. For each feature in the dataset, the paired label corresponds to the desired output. The ML algorithm therefore needs to learn a mapping for all the feature-label pairs. This type of learning is common for tasks such as classifying features into known categories, or learning to fit a known function. This type of learning is used in this project as the target function $f(x)$ is used as the label for the corresponding x input value.

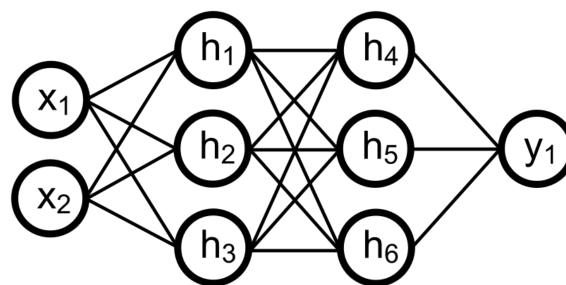


Figure 1.2: FCNN with 2 input nodes, 1 output node and 2 hidden layers each with 3 nodes. The number of nodes in the input, hidden, and output layers can vary from one, to many. An image used as an input could have an input node for every pixel.

Network architecture - In this project, a Neural Network (NN) acts as the model the ML algorithm uses for learning. A NN is defined as a series of layers as shown in Figure 1.2. The first layer is the input layer, corresponding to features of the dataset. The final layer is the output layer, serving as the output of the network. In the basic supervised case, the input will have the same dimensions as the features, and the output that of the labels.

The hidden layers lie between the input and output layers. The term ”hidden” comes from the layers being hidden from the user, as opposed to the input and output. This is where the term ”blackbox” is commonly used, as it is a difficult task to understand how the input signal transforms within the hidden layers to eventually become the desired output. Gaining insight into how the signal transforms within these layers is an important field of research in ML.

The number of hidden layers plus the output layer gives the **depth** of the NN. Shallow learning involves one or two hidden layers and was the standard before more powerful GPUs were developed. With the increase in computation power available, the field of deep learning, involving 2+ hidden layers, was born.

Each layer is comprised of one or more neurons (or nodes), as shown in Figure 1.2. In this Figure, the input has 2 neurons (x_1, x_2), the output has 1 neuron (y_1), and the hidden layers each have 3 neurons (h_n). The number of neurons in the widest hidden layer is commonly refereed to as the **width** of the NN. The connections between neurons represent how the signal propagates through the network.

Network parameters - NNs are comprised of two different types of parameters: weights and biases. These are the tunable parameters we seek to optimize when training a NN. Weights multiply the node's incoming signal while biases are added to it. Based on the architecture of the network, these parameters can take varying dimensions.

For this project, we investigate the simplest form of NNs, a Fully Connected Neural Network (FCNN). An example of a small FCNN is shown in Figure 1.2. The notable feature of a FCNN is that every node is connected to all the nodes in the previous layer, hence the term *fully connected*.

As an example of these elements, the h_1 node in Figure 1.2 has 2 input connections. This connection represent the weights that multiply the signal coming in from the precedent input nodes. The input to the node will be the weights multiplying the corresponding output of the precedent layer, in this case the raw input x_1, x_2. These are summed together to form the full input to the node:

$$w_1^{h_1} x_1 + w_2^{h_1} x_2, \tag{1}$$

where $w_i^{h_1}$ are the weights of the connections between the input layer and h_1.

The bias is then added to this sum:

$$w_1^{h_1} x_1 + w_2^{h_1} x_2 + b^{h_1}, \tag{2}$$

where b^{h_1} is the bias of h_1.

Activation functions - With Equation 2, layers are connected linearly. The implication of this is that the depth of the network has no impact on the network's ability to learn complex relations. This can be seen by looking at the full output of the network:

$$y_1 = w_1^{y_1} h_4 + w_2^{y_1} h_5 + w_3^{y_1} h_6 + b^{y_1}, \tag{3}$$

where the three weights leading into the output y_1 multiply the output of their corresponding hidden node. Expanding the outputs of the hidden nodes in equation 3 yields:

$$y_1 = w_1^{y_1}\left(w_1^{h_4}h_1+w_2^{h_4}h_2+w_3^{h_4}h_3+b^{h_4}\right)+w_2^{y_1}\left(w_1^{h_5}h_1+w_2^{h_5}h_2+w_3^{h_5}h_3+b^{h_5}\right)+w_3^{y_1}\left(w_1^{h_6}h_1+w_2^{h_6}h_2+w_3^{h_6}h_3+b^{h_6}\right)+b^{y_1} \tag{4}$$

$$\begin{aligned} y_1 =& \left(w_1^{y_1}w_1^{h_4} + w_2^{y_1}w_1^{h_5} + w_3^{y_1}w_1^{h_6}\right)h_1 + \left(w_1^{y_1}w_2^{h_4} + w_2^{y_1}w_2^{h_5} + w_3^{y_1}w_2^{h_6}\right)h_2 \\ &+ \left(w_1^{y_1}w_3^{h_4} + w_2^{y_1}w_3^{h_5} + w_3^{y_1}w_3^{h_6}\right)h_3 + w_1^{y_1}b^{h_4} + w_2^{y_1}b^{h_5} + w_3^{y_1}b^{h_6} + b^{y_1} \end{aligned} \tag{5}$$

$$y_1 = w_1'h_1 + w_2'h_2 + w_3'h_3 + b', \tag{6}$$

where the weights of a corresponding hidden output and all the biases have been summed together. Doing the exact same procedure for the hidden layers h_1 to h_3 gives:

$$y_1 = w_1'\left(w_1^{h_1}x_1 + w_2^{h_1}x_2 + b^{h_1}\right) + w_2'\left(w_1^{h_2}x_1 + w_2^{h_2}x_2 + b^{h_2}\right) + w_3'\left(w_1^{h_3}x_1 + w_2^{h_3}x_2 + b^{h_3}\right) + b'. \tag{7}$$

And again, due to the linear nature of these relations, the corresponding terms can be summed together:

$$y_1 = w_1'x_2 + w_2'x_2 + b'. \tag{8}$$

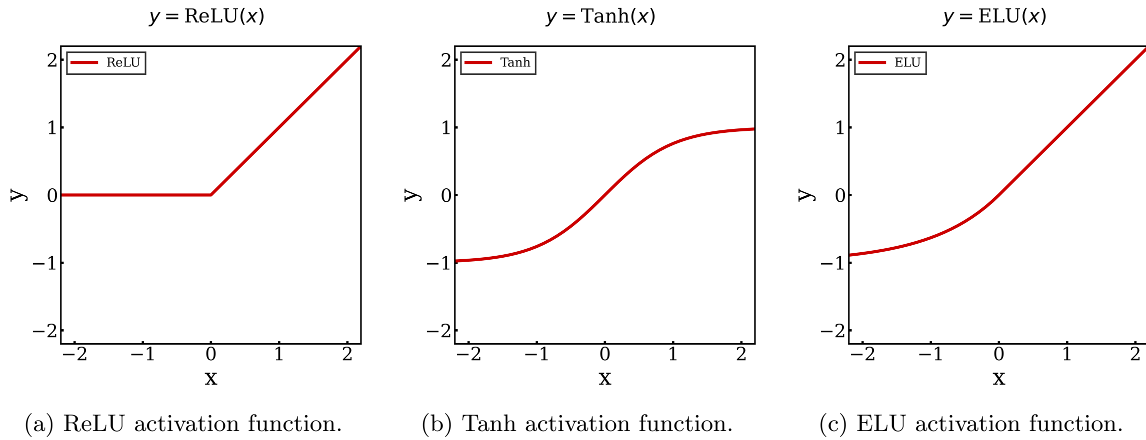
All the parameters contained within the network can be simplified down to only 3. This network can only learn a simple linear relationship as described by Equation 8.

To properly learn more complex relations that can't be mapped by this simple linear function, non-linearity has to be introduced to the network. This is done using non-linear activation functions. Activation functions act on the output of every node excluding the output node, giving the network the ability to learn more complicated non-linear relations.

$$\text{ReLU}(x) = \max(0, x) \tag{9}$$

$$\text{Tanh}(x) = \frac{\exp(x) - \exp(-x)}{\exp(x) + \exp(-x)} \tag{10}$$

$$\text{ELU}(x) = \begin{cases} x & \text{if } x > 0 \\ \exp(x) - 1 & \text{if } x \leq 0 \end{cases} \tag{11}$$



(a) ReLU activation function. (b) Tanh activation function. (c) ELU activation function.

Figure 1.3: Commonly used activation functions used in ML. These functions act on the output of NN input and hidden layers to introduce non-linearity.

Equations 9-11 define a selection of popular activation functions which can be seen in Figure 1.3. There exists an ever growing number of different activation functions to fulfil niche objectives, but these ones are most commonly used. Stability and speed are the main considerations when choosing what function to use. While a more complicated function suited to the target function would result in faster learning, it relies on knowing the exact shape of the target, which is often not the case. Therefore, simple functions that both prioritize results and speed are used. In this project, $Tanh$ activation is used. We found that the smoothness of the $Tanh$ function helps with fitting smooth functions used as target, which speeds up learning and reduces the necessary training time. This can be seen in Figure 1.1, where the Tanh activation function more smoothly fits the sinusoidal target function. With that said, other activation functions such as Rectified Linear Unit (ReLU) would have worked, but we found that $Tanh$ trained faster even when accounting for the extra computation required. This is likely due to how well suited $Tanh$ is for the target sinusoidal function. As this project involves comparing two optimizers using the same network architecture, the activation function more suited for the task is picked.

The full equation for the output of the h_1 node in Figure 1.2 is thus:

$$h_1 = f\left(w_1^{h_1} x_1 + w_2^{h_1} x_2 + b^{h_1}\right), \tag{12}$$

where y_{h_1} is the output of the node and f is the activation function acting on the node. This node output y^{h_1} will then serve as part of the input of the nodes in the second hidden layer. This process repeats until the output layer. As to not limit the output range of the network, the final output layer does not have an activation function for regression tasks.

Loss function - The loss function is used to score the current set of network parameters. By convention, the loss functions are constructed to make a smaller loss represent a better performance.

Therefore, the objective is to minimize the loss.

What loss function is chosen depends on the task. For regression tasks such as done in this project, the Mean Square Error (MSE) loss, shown in equation 13, is commonly used:

$$U = \frac{1}{N} \sum_{i=1}^{N} \left(y_i^{output} - y_i^{target}\right)^2, \tag{13}$$

where N is the number of feature-label pairs in the dataset, that is, the number of training samples. y_i^{output} is the network output for the training sample feature while y_i^{target} is the true label we desire the output to be.

$$U = \frac{1}{N} \sum_{i=1}^{N} \mathrm{abs}\left(y_i^{output} - y_i^{target}\right) \tag{14}$$

The MSE loss is used over the simpler Mean Absolute Error (MAE) shown in equation 14 due to desirable properties for learning. Notably, MSE is more sensitive to outliers in the dataset, as one sample poorly evaluated will have a greater contribution on the loss as compared to MAE. This is useful as it makes the network more robust to imbalanced datasets wherein some features are less represented. MSE therefore discourages the network from disregarding the rarer samples and only learning the often less important prevalent data. Additionally, the smooth change of the loss at low error makes the network more stable as compared to the linearly increasing error of MAE. For MAE a change in the error when the error is very small will have the same effect as when the error is large, which can lead to spikes in the loss during training. Meanwhile, the squared error means that the effect of small change in error scale as training occurs.

The final piece of learning is the optimizer. This refers to the algorithm used to minimize the loss, driving the learning. In this project, Gradient Descent and a Monte Carlo algorithm are used.

1.1.2 Gradient descent

Learning using gradients is done using back-propagation [14]. This algorithm calculates the gradient of the loss with respect to every parameter (weights and biases), which is then used to update the parameters in the negative gradient direction to lower the loss. Repeated application of gradient descent then allows minimization of the loss. Back-propagation occurs in two stages. The first is a forward pass where the network produces outputs based on input data. These outputs are then used to compute the loss. The backward pass then calculates the gradient of the loss with respect to every parameter in the network using an efficient implementation of the derivative chain rule. By starting with the gradient of the final layer, these values can be used to calculate the gradient of the previous layer, and so on until the

gradient of every parameter has been calculated from the output layer to the input layer. This method means that repeated calculations are avoided and intermediate gradients of a parameter with respect to neighbouring layers are unnecessary, as the gradient of the parameters with respect to the loss is directly computed.

The last step of this process involves shifting every parameter in the network in the direction of the negative gradient as shown in equation 15.

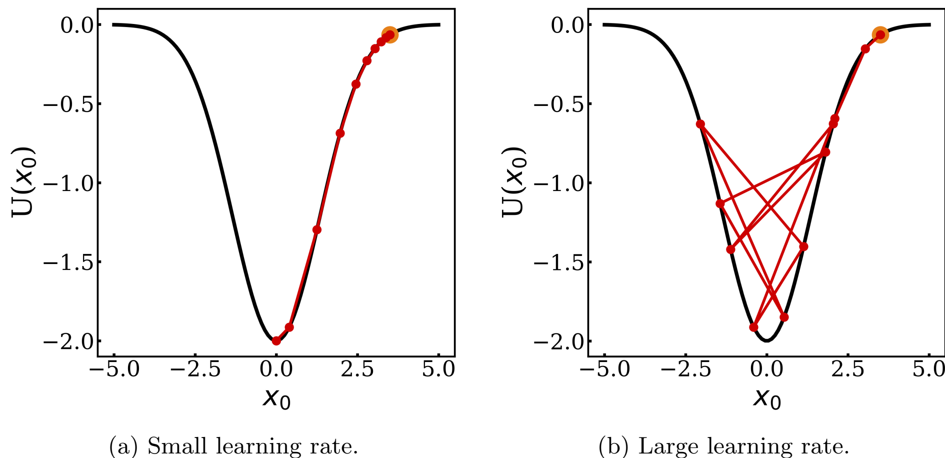
$$x_i^{t+1} = x_i^t - \alpha \frac{\partial U(\boldsymbol{x})}{\partial x_i}, \tag{15}$$

where the parameter x_i is discretely updated by a scalar, called the learning rate α, multiplied by the gradient of the loss U with respect to the parameter x_i. As we seek to minimize the loss, the negative of the gradient is used.

Equation 15 can also be written as:

$$\frac{\partial x_i}{\partial t} = -\alpha \frac{\partial U(\boldsymbol{x})}{\partial x_i}, \tag{16}$$

where the time t represents the discrete epochs.



(a) Small learning rate.

(b) Large learning rate.

Figure 1.4: Mock example of how learnable parameters evolve when using GD to optimize. A small (left) and large (right) learning rate is used to update the parameter x_0 based on the gradient of the pseudo-loss.

The learning rate α controls the magnitude of the gradient's impact on the learnable parameter. The larger this value is, the more effect the gradient will have on the parameters, at the risk of greater instability due to the possibility of overstepping. In contrast, a smaller α will lead to slower but more stable learning. Figure 1.4 shows the consequence of a too large learning rate, as the optimizer fails to reach the minima. In contrast, the smaller learning rate spends many iterations near the starting point. Choosing a learning rate is therefore incredibly important when training a NN. In general, a learning

rate of the order $1e-4$ is used, but can be changed depending a variety of factors, such as what loss is used.

1.2 Methods

1.2.1 Neuroevolution

MC learning - Under Monte Carlo (MC), each parameter θ_i of the neural network is proposed a unique mutation of type $\theta_i \rightarrow \theta_i + \epsilon_i$, where ϵ_i is a random independent Gaussian number with zero mean and variance σ^2. All mutations are accepted with probability $min\big(1, e^{-\beta\Delta U}\big)$, where β is an inverse temperature and ΔU is the change in the loss caused by the proposed mutations ϵ on the entire network. This temperature can be viewed as the likelihood of uphill moves being accepted. The higher the temperature, the more likely uphill moves will be accepted. At zero temperature, only moves which reduce the loss are accepted, while at infinite temperature all moves are accepted. In a physical system this would correspond to the physical temperature of the system. For example, in a simulation of a single particle travelling across a potential landscape, the temperature would be related to the kinetic energy of the particle, dictating its ability to escape from a potential well U as it travels.

$$\Delta U = U(\boldsymbol{x} + \boldsymbol{\epsilon}) - U(\boldsymbol{x}). \tag{17}$$

If rejected, the current mutations are ignored and a new set is generated. The proposal of a set of mutations is called a move, and thus either a move is accepted or rejected every epoch of training. The MC algorithm is shown in Algorithm 1.

There exists two regimes to this algorithm: $\beta = \infty$ and $\beta \neq \infty$. In the case of $\beta = \infty$, or the reciprocal temperature being equal to zero, only moves which do *not* increase the loss, $\Delta U \leq 0$ are accepted. Under these circumstances, the loss can only decrease or stay equal. For finite β, moves that increase the loss are accepted in relation to the magnitude of β and ΔU. As β tends towards zero the more likely uphill moves are to be accepted. In the limit of $\beta = 0$, all moves are accepted and no learning can occur.

This form of MC is commonly used in simulating particles moving across a potential surface, such as nanoparticles being trapped on a surface [15] or the self-assembly of large nanoparticles [16]. This algorithm has been the focus of much research, such as improving its performance in cases with strong attractions or clusters of particles [17]. Similarly to a particle travelling across a potential surface, the network parameters can be viewed as coordinates of a single particle in a high dimensional space, where each network parameter is a dimension. The loss then acts as the potential surface on which this undefined particle travels.

Algorithm 1 MC optimizer.

```
1: Initialize all parameters θ_i
2: for N epochs do
3:     for all i do
4:         θ_i → θ_i + ε_i
5:     end for
6:     Calculate loss difference ΔU
7:     Calculate acceptance p = min(1, e^{-βΔU})
8:     Generate random number r = Uniform[0, 1]
9:     if r < p then
10:        Accept ε
11:    else
12:        Revert ε
13:    end if
14: end for
```

Uphill moves are not permitted when training using gradient descent outside of the often unwanted increases in loss which occur when the gradient update is too large in relation to the local loss surface topography. This often occurs when the learning rate is too large or the loss is very small. It might then come as a surprise that an algorithm which directly allows uphill moves can have an exact correspondence with one that does not, but this is the case shown here. The exact equations for correspondence will be shown for both regimes in the next sections.

1.3 Results

1.3.1 Finite β correspondence derivation

To begin, we examine the regime where β is finite. Here, moves which increase the loss can be accepted with a probability dependent on the change in loss and the chosen temperature. Viewing the learnable parameters as coordinates of a particle and the loss surface as a potential on which the particle travels, a higher temperature simulates a higher energy particle capable of overcoming larger potential barriers. In the limit of infinite temperature, or $\beta = 0$, the particle would have enough energy to disregard the potential landscape altogether, as all moves would be accepted. Relating this to the NN, a higher temperature signifies allowing the network to more often take steps that cause it to have a worse set of parameters for mapping input to output. This may seem counter-intuitive for the purpose of learning, but can be a useful feature in helping the network navigate very hilly and noisy loss surfaces.

In this MC regime, and in the limit of small mutation size $\epsilon_i \to 0$, the parameters of the network evolves following:

$$\frac{\partial x_i}{\partial t} = -\frac{\beta\sigma^2}{2}\frac{\partial U(\boldsymbol{x})}{\partial x_i} + \xi_i(t). \tag{18}$$

derived by Dr. Stephen Whitelam in the paper for this project [18]. Here, σ^2 is the variance of the random Gaussian used for the network parameter mutations ϵ_i. The additional term ξ is a Gaussian white noise with zero mean and the same variance as the mutations σ^2.

The derivation for equation 18 starts by looking at the time evolution of $P(\boldsymbol{x}, t)$, the probability that a network has the set of parameters $\boldsymbol{x}$ at a given time t. This is given by the master equation:

$$\partial_t P(\boldsymbol{x}, t) = \int d\boldsymbol{x}' \left[P(\boldsymbol{x}', t) W_{\boldsymbol{x}-\boldsymbol{x}'}(\boldsymbol{x}') - P(\boldsymbol{x}, t) W_{\boldsymbol{x}'-\boldsymbol{x}}(\boldsymbol{x}) \right], \tag{19}$$

where $\boldsymbol{x}$ is the current set of parameters and $\boldsymbol{x}'$ is the set of parameters moved to with probability $W_{\boldsymbol{x}-\boldsymbol{x}'}$. For MC as described in Algorithm 1, the step $d\boldsymbol{x}'$ is the mutations $\boldsymbol{\epsilon}$, which gives a MC master equation:

$$\partial_t P(\boldsymbol{x}, t) = \int d\boldsymbol{\epsilon} \left[P(\boldsymbol{x} - \boldsymbol{\epsilon}, t) W_{\boldsymbol{\epsilon}}(\boldsymbol{x} - \boldsymbol{\epsilon}) - P(\boldsymbol{x}, t) W_{\boldsymbol{\epsilon}}(\boldsymbol{x}) \right]. \tag{20}$$

The chance of going from a set of parameter $\boldsymbol{x}$ to another $\boldsymbol{x} + \boldsymbol{\epsilon}$ is:

$$W_{\boldsymbol{\epsilon}}(\boldsymbol{x}) = p(\boldsymbol{\epsilon}) min(1, e^{-\beta[U(\boldsymbol{x}+\boldsymbol{\epsilon}) - U(\boldsymbol{x})]}), \tag{21}$$

where $p(\boldsymbol{\epsilon})$ is the probability of proposing a set of Gaussian random numbers $\boldsymbol{\epsilon}$. The derivation then passes from the master equation 20 to a Fokker-Planck equation by assuming a small mutation scale and expanding the two terms $P(\boldsymbol{x} - \boldsymbol{\epsilon}, t)$ and $W_{\boldsymbol{\epsilon}}(\boldsymbol{x} - \boldsymbol{\epsilon})$ to second order in σ. Plugging these back into equation 20 gives:

$$\partial_t P(\boldsymbol{x}, t) \approx - \int d\boldsymbol{\epsilon} (\boldsymbol{\epsilon} \cdot \nabla) P(\boldsymbol{x}, t) W_{\boldsymbol{\epsilon}}(\boldsymbol{x}) + \frac{1}{2} \int d\boldsymbol{\epsilon} (\boldsymbol{\epsilon} \cdot \nabla)^2 P(\boldsymbol{x}, t) W_{\boldsymbol{\epsilon}}(\boldsymbol{x}), \tag{22}$$

where $\boldsymbol{\epsilon} \cdot \nabla = \sum_{i=1}^{N} \epsilon_i \partial_i$ runs over all parameters in the network N. Taking the integrals in 22 inside the sums gives:

$$\partial_t P(\boldsymbol{x}, t) \approx - \sum_{i=1}^{N} \frac{\partial}{\partial x_i} \left(A_i(\boldsymbol{x}) P(\boldsymbol{x}, t) \right) + \frac{1}{2} \sum_{i,j=1}^{N} \frac{\partial^2}{\partial x_i \partial x_j} \left(B_{ij}(\boldsymbol{x}) P(\boldsymbol{x}, t) \right), \tag{23}$$

where

$$A_i(\boldsymbol{x}) = \int d\boldsymbol{\epsilon} \epsilon_i W_{\boldsymbol{\epsilon}}(\boldsymbol{x}), \tag{24}$$

and

$$B_{ij}(\boldsymbol{x}) = \int d\boldsymbol{\epsilon} \epsilon_i \epsilon_j W_{\boldsymbol{\epsilon}}(\boldsymbol{x}). \tag{25}$$

The final step is to calculate these two quantities. This is done by replacing $W_{\boldsymbol{\epsilon}}$ with equation 21 and calculating the integral. For the case with finite β we get:

$$A_i(\boldsymbol{x}) = -\frac{\beta\sigma^2}{2}\partial_i U, \tag{26}$$

and

$$B_ij(\boldsymbol{x}) = \sigma^2\delta_{ij}. \tag{27}$$

Plugging these into equation 23 we get a second order Fokker-Planck equation which is equivalent to the N Langevin equations:

$$\frac{\partial x_i}{\partial t} = -\frac{\beta\sigma^2}{2}\frac{\partial U(\boldsymbol{x})}{\partial x_i} + \xi_i(t). \tag{28}$$

Two observations are drawn from this equation when compared to the GD parameter evolution equation 16. The first being that taking the average of multiple trajectories starting from identical initial conditions will average out the Gaussian white noise, leaving the equation:

$$\frac{\partial x_i}{\partial t} = -\frac{\beta\sigma^2}{2}\frac{\partial U(\boldsymbol{x})}{\partial x_i}. \tag{29}$$

The second observation is the equivalence relation that comes from the learning rate in the GD equation 16 and the terms in equation 29, as shown in equation 30.

$$\alpha = \frac{\beta\sigma^2}{2}. \tag{30}$$

When this relation holds true, and in the limit of small mutation size, the evolution of the network parameters averaged over multiple MC trajectories will be equivalent to that of the GD parameters.

My contribution to this project was to numerically demonstrate this correspondence. That is, train a neural network using both vanilla GD and MC to show the equivalence. In this next section, I will go over how this was done and the main results.

1.3.2 Finite β correspondence

Implementation - All code written for this project was done in Python. PyTorch [19] was used to implement neural networks as well as train them using the in-built GD optimizer. As a popular package widely used in industry, PyTorch was chosen instead of a manual implementation to provide a fair comparison between GD and MC, as the gradient calculations are heavily optimized in the package.

The MC optimizer was implemented to directly act on the PyTorch neural network using in-built PyTorch functions. Finally, all plotting was done using matplotlib.

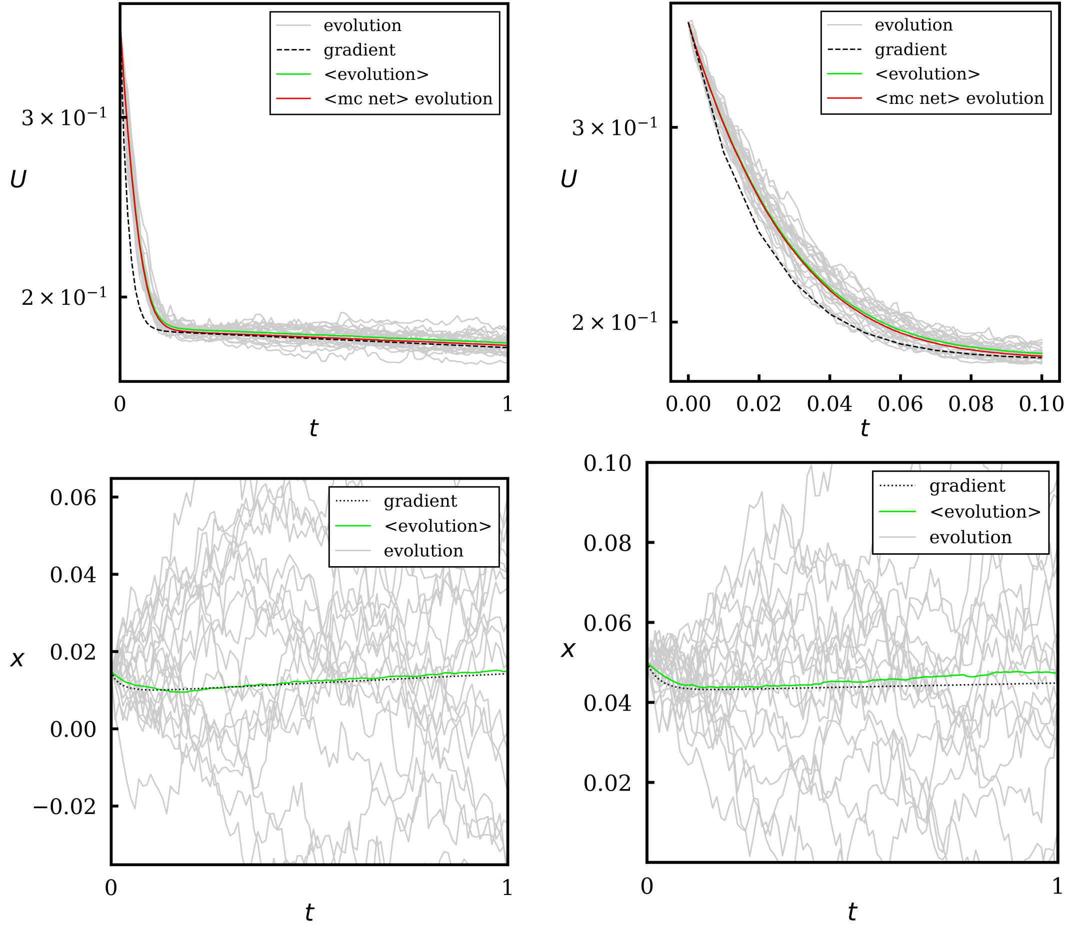


Figure 1.5: Correspondence for a network trained on $y = sin(\pi x)$ on the range $\{-1 < x < 1\}$. Loss is shown for learning rate $\alpha = 10^{-4}$ (left) and $\alpha = 10^{-5}$ (right). In green is the mean loss for all MC evolution trajectories. Red is the loss for the mean network constructed from the mean of all parameters from the MC trajectories. The evolution of two parameters is shown at the bottom.

Results - The correspondence for finite β is shown in Figure 1.5. The input to the network was an x coordinate in the range $\{-1 < x < 1\}$ while the target output was the function $y = sin(\pi x)$. Here t represents time in training epochs $t = epoch \times \alpha$, where α is the learning rate for GD, or the equivalent one for the MC trajectories as given by equation 30. In general, the CPU time taken to train is roughly equivalent using both methods, and therefore the epochs can be directly compared. To start, a simple NN is initialized with a chosen set of parameters using a randomly selected random seed. The network used was a depth 2 network with one hidden layer of width 256. MSE loss was used for training. To compare GD and MC the learning rate α was chosen first, and the corresponding β and σ were then calculated following the relation shown in equation 30.

There are a number of constraints we need to meet:

1. Many MC trajectories are needed to average out the Gaussian noise.

2. Correspondence holds in the limit of small σ.

3. We want β to be as small as possible to stay in the finite regime.

4. The smaller α is, the longer training takes.

For the first point, 100 MC trajectories all starting from the same set of network parameters were run in parallel. Each trajectory was given a unique random seed for reproducibility.

The trouble in selecting α, β, and σ can be seen from the relations of the corresponding terms in equation 30. Foremost, α needs to be large enough for training to occur in a reasonable amount of time. This constraint limits what values β and σ can take, as increasing one necessitates reducing the other. Similarly, for correspondence to hold, σ needs to be as small as possible, but we also want β to be minimal as to not leave the finite β regime.

After multiple tests, the values for β and α were set to $\beta = 1000$, $\alpha = [10^{-4}, 10^{-5}]$ as shown in Figure 1.5. σ is then calculated using:

$$\sigma = \sqrt{\frac{2\alpha}{\beta}}. \tag{31}$$

The effect of averaging over multiple MC trajectories is clearly shown in the bottom two plots of Figure 1.5 which show the evolution of two randomly selected parameters from the network. The green line representing the averaged parameter follows the GD parameter in contrast to the separate MC trajectories shown in grey.

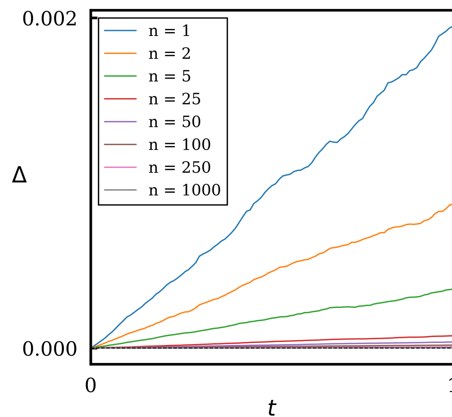


Figure 1.6: Absolute mean difference over all parameters between a GD trained network and the mean parameters of n MC trained networks starting from the same set of initial parameters.

This effect is further shown in Figure 1.6 showing the mean difference over all parameters of n networks trained using MC and a GD trained network. As training occurs, the individual MC trajectories diverge from the GD trajectory, leading to an increase in the difference. This corresponds to the random Gaussian

noise ξ as shown in equation 18. As more MC networks are used for the mean MC parameters, shown by increasing n, the noise is diminished which leads to less difference with the GD parameters.

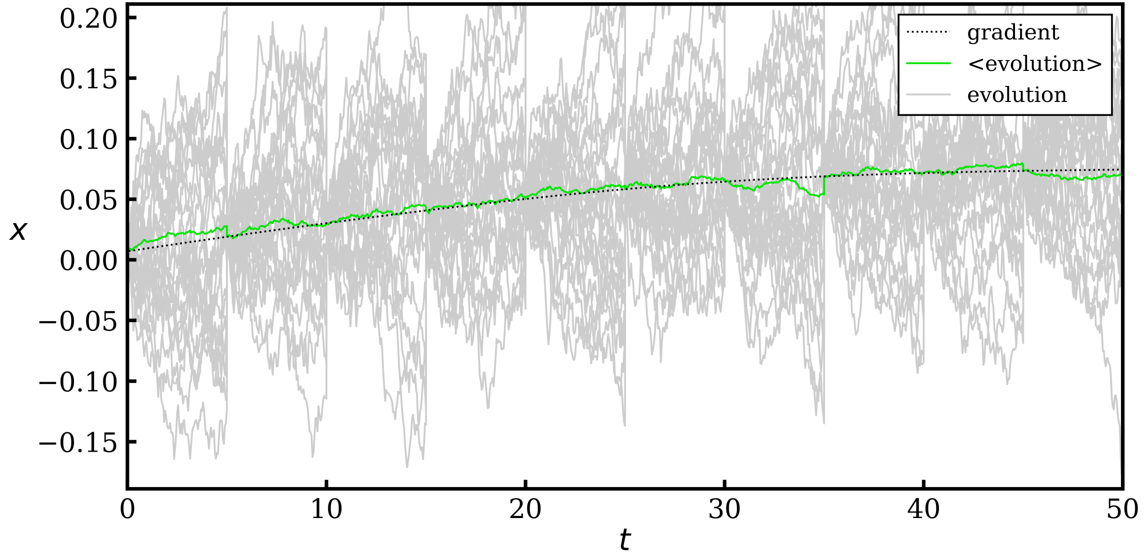


Figure 1.7: Evolution of single weight of a width 100 depth 2 network trained using GD and MC with finite β. 150 MC trajectories are shown wherein the parameters were reset to the GD ones every 5 time units.

To show correspondence beyond the early stages of learning, Figure 1.7 shows the evolution of a single parameter of a network trained using GD in the dotted black line. Every 5 time units, 100 MC trajectories were started from the current set of GD network parameters and trained for 5 time units. This process was then repeated for the full training time. This Figure shows how correspondence still holds later in the training process, and is not exclusive to the early stages of learning where the direction of learning might be the strongest. The resetting of the parameters is necessary since the correspondence breaks after a given time due to the non-infinite number of MC trajectories and non-zero σ.

This Figure shows that correspondence holds for all stages of training as the mean parameter of all MC trajectories matches the GD one.

1.3.3 Infinite β correspondence derivation

In the regime where $\beta = \infty$, only moves which reduce or do not change the loss are accepted. As opposed to the finite β regime, the loss cannot increase. In this regime, the evolution of the parameters evolve following a slightly different equation as derived by Dr. Stephen Whitelam in the project's paper [18]:

$$\frac{\partial x_i}{\partial t} = -\frac{\sigma}{\sqrt{2\pi}} \frac{1}{|\nabla U(\boldsymbol{x})|} \frac{\partial U(\boldsymbol{x})}{\partial x_i} + \eta_i(t), \tag{32}$$

where η is a similar Gaussian noise to ξ in equation 18, but with variance $\sigma^2/2$ instead of σ^2.

Starting from equations 24 and 25, these equations are solved for the case of infinite β by transforming them to standard Gaussian integrals. We end up with:

$$A_i(\boldsymbol{x}) = -\frac{\sigma}{\sqrt{2\pi}} \frac{\partial_i U(\boldsymbol{x})}{|\nabla U(\boldsymbol{x})|}, \tag{33}$$

and

$$B_i j(\boldsymbol{x}) = \frac{1}{2}\sigma^2 \delta_{ij}. \tag{34}$$

Similar to the case for finite β, inserting these terms into equation 23 gives a Fokker-Planck equation equivalent to the N Langevin equations:

$$\frac{\partial x_i}{\partial t} = -\frac{\sigma}{\sqrt{2\pi}} \frac{1}{|\nabla U(\boldsymbol{x})|} \frac{\partial U(\boldsymbol{x})}{\partial x_i} + \eta_i(t), \tag{35}$$

The normalization factor $|\nabla U(\boldsymbol{x})| = \sqrt{\sum_{i=1}^{N} (\partial U(\boldsymbol{x})/\partial x_i)^2}$ acts as a rescaling of the time step based on the magnitude of the gradient.

Averaging over many MC trajectories gives:

$$\frac{\partial x_i}{\partial t} = -\frac{\sigma}{\sqrt{2\pi}} \frac{1}{|\nabla U(\boldsymbol{x})|} \frac{\partial U(\boldsymbol{x})}{\partial x_i}. \tag{36}$$

Comparing equation 36 to the GD equation 16 gives the relation:

$$\alpha = \frac{\sigma}{\sqrt{2}}. \tag{37}$$

The additional normalization term $|\nabla U(\boldsymbol{x})|$ is taken into account by using a slightly altered GD implementation:

$$\frac{\partial x_i}{\partial t} = -\alpha \frac{1}{|\nabla U(\boldsymbol{x})|} \frac{\partial U(\boldsymbol{x})}{\partial x_i}. \tag{38}$$

This is a clipped GD implementation [20]. Specifically, clipped GD using the gradient norm. This normalizes the step taken by the gradient, preventing large gradients from introducing spikes in the loss by limiting the size of steps possible.

1.3.4 Infinite β correspondence

Implementation - Only a few modifications are needed for this regime as compared to the finite β. The normalization term is taken into account by implementing gradient clipping. In addition, as all moves which increase the loss are rejected, the probability $min\big(1, e^{-\beta \Delta U}\big)$ does not need to be calculated and moves can be accepted if the new loss is less than or equal to the previous loss.

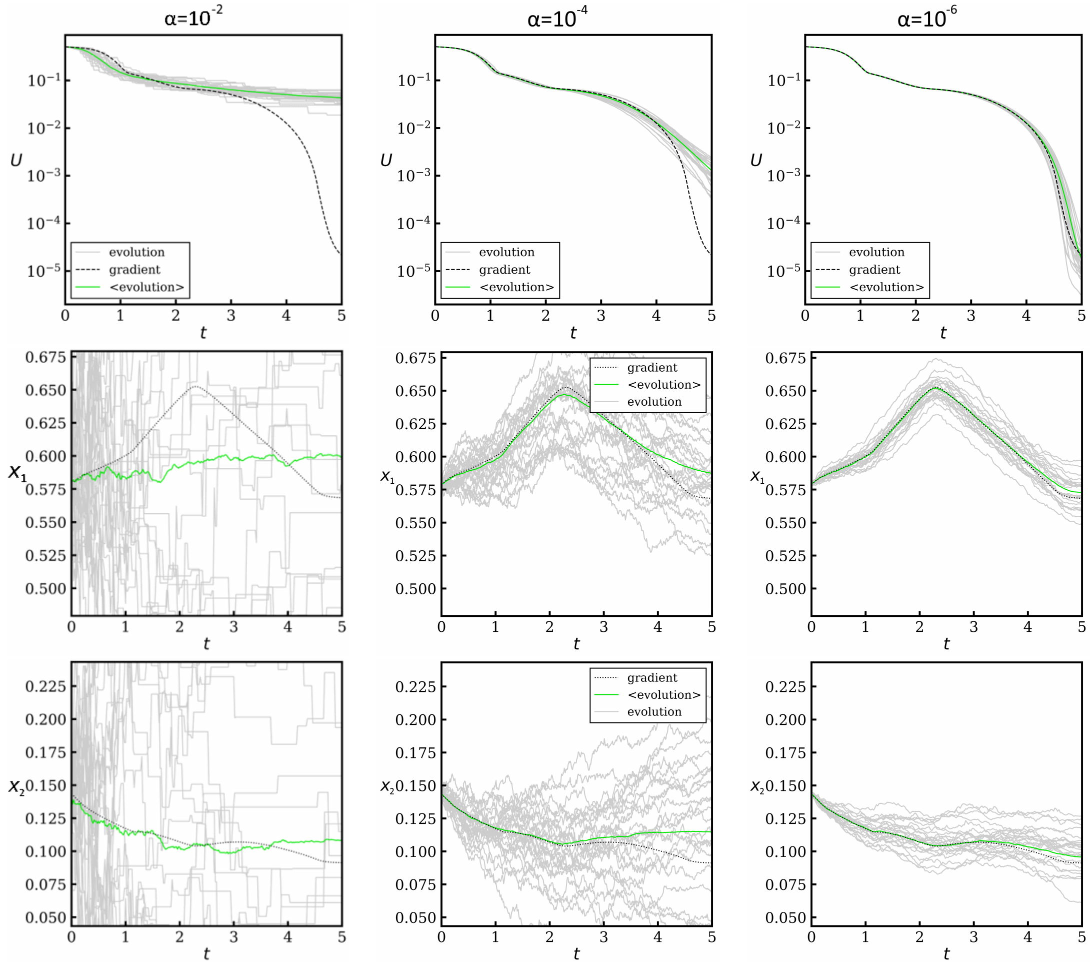


Figure 1.8: Correspondence between GD and MC for $\alpha = [10^{-2}, 10^{-4}, 10^{-5}]$ from left to right. Top plots show the loss as a function of training time. Bottom plots show the evolution of two parameters. All trajectories start from the same initial parameters. Individual MC trajectories are shown in grey.

Results - Correspondence between GD and MC in this regime is shown in Figure 1.8. Three different mutation sizes ϵ are shown from left to right corresponding to an effective learning rate $\alpha = [10^{-2}, 10^{-4}, 10^{-5}]$. The corresponding mutation size variance σ^2 is calculated using equation 37. In all plots, GD is shown as a black dotted line, while the individual MC trajectories are shown in grey. The green line corresponds to the average of all MC trajectories. Correspondence quickly breaks for the largest α as the individual MC trajectories rapidly diverge from the GD baseline due to the large mutations as can be seen in the parameter plots. As α, and therefore the mutation size ϵ, decreases, the random Gaussian noise in equation 35 becomes smaller, causing the MC trajectories to not diverge as quickly. This plot clearly demonstrates the statement that correspondence holds in the limit of small mutation size ϵ.

For this test, a FCNN of depth 8 and width 32 was trained on $y = sin\,(\pi x)$. The number of epochs trained depends on the equivalent learning rate:

$$t_{total} = \text{num_epochs} \times \alpha, \tag{39}$$

which gives a number of epochs of $[5 \times 10^2, 5 \times 10^4, 5 \times 10^5]$ for the shown α in Figure 1.8. For GD, only the smallest learning rate $\alpha = 10^{-5}$ was used as this method of plotting the time makes all learning rates equivalent for GD, with the exception of a too big learning rate introducing artefacts in learning. For this reason, the smallest learning rate was used to most accurately follow the gradient.

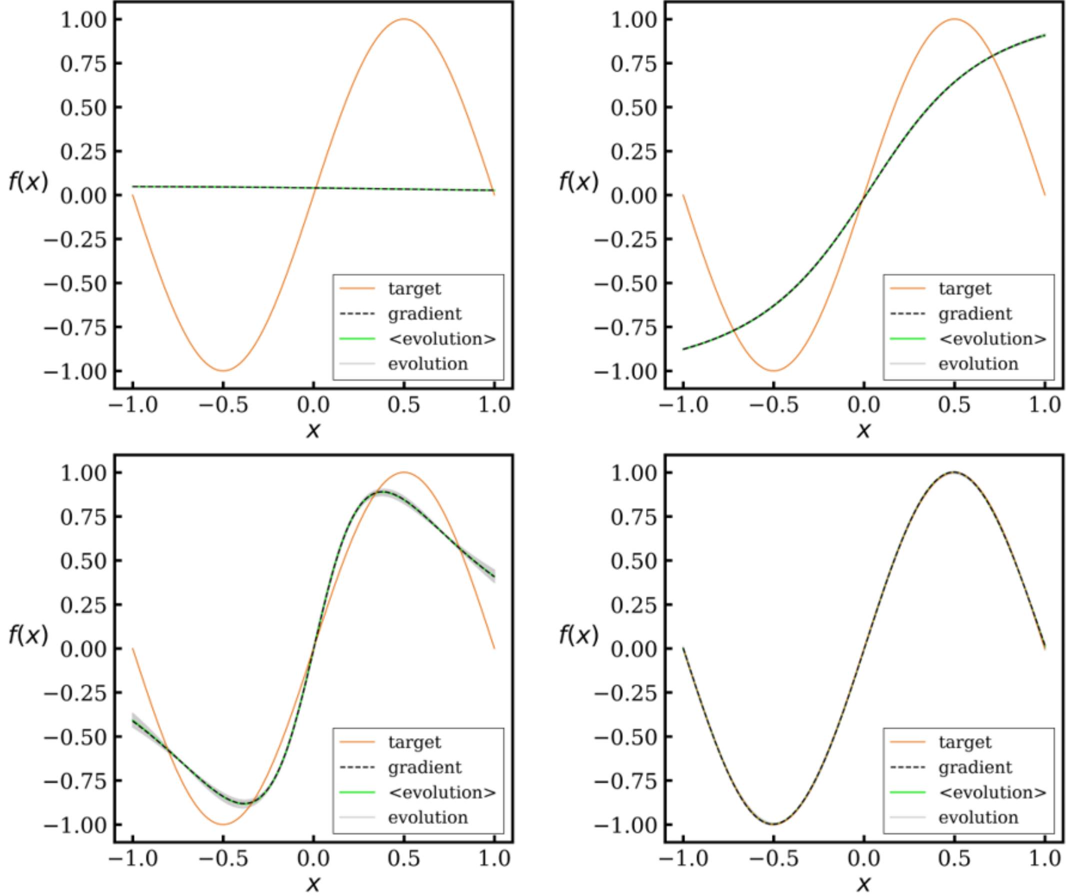


Figure 1.9: Prediction for GD-MC infinite beta correspondence on $sin(\pi x)$ target function with $\alpha = 10^{-5}$. GD and all MC evolution trajectories are shown. The mean prediction of the MC trajectories is shown in green. Prediction is shown for initialization $t = 0$ (top left), finished $t = 5$ (bottom right) as well as two in-between times $t = 1.25, 3.75$.

The associated network predictions during training for $\sigma = 10^{-5}$ are shown in Figure 1.9. This plot shows the output of the network at 4 different times during training $t = [0.00, 1.25, 3.75, 5.00]$. For training, 1001 points evenly spaced across $\{-1 < x < 1.0\}$ were used.

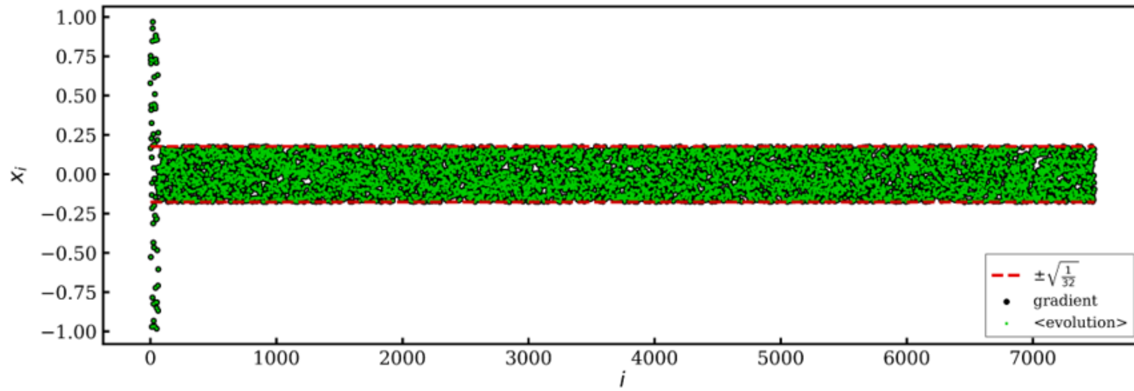


Figure 1.10: All parameters in a depth 8, width 32 network after initialization.

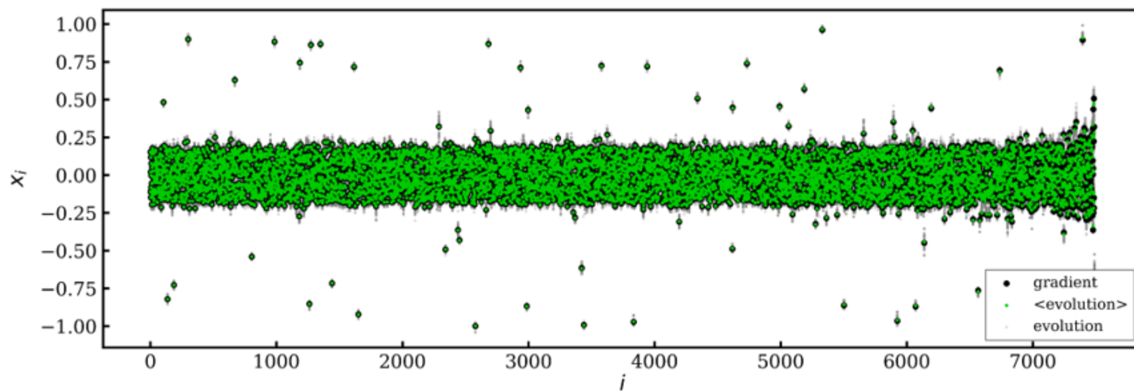


Figure 1.11: All parameters in a depth 8, width 32 network after training.

To visualize the parameters of the network, figures 1.10 - 1.11 show all parameters for the network from figures 1.8 - 1.9 trained with $\alpha = 10^{-5}$. In total, the network has about 7400 parameters. The top plot shows all parameters after initialization. They are sorted from input (left) to output (right). The red dashed lines denotes the range for the initialization in the hidden layers using the default PyTorch initialization. The lower plot shows all parameters after training is complete, sorted from lowest to highest difference between the average of all parameters of the MC trajectories and the corresponding GD parameter. Therefore, the parameters with the largest difference between GD and the mean MC trajectories are shown on the right. As can be seen, even these are near the corresponding GD parameter.

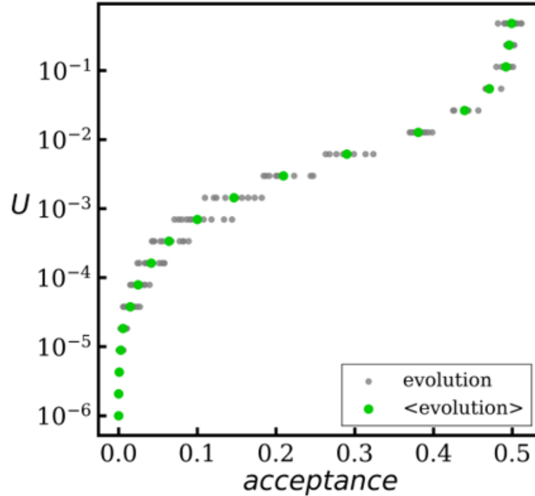


Figure 1.12: Acceptance plot for MC trajectories for a depth 8, width 32 network. Acceptance is binned as a function of the loss. Grey corresponds to individual MC trajectories while green is the mean.

Figure 1.12 shows acceptance as training loss diminishes. Training begins at the top right point in the plot, corresponding to the starting loss. At this stage, the network has approximately a 50% chance of proposing a move which reduces the loss. As training continues, the acceptance falls rapidly to a minuscule chance. This reduction in acceptance means that learning occurs much slower at lower loss, as good moves will be much rarer.

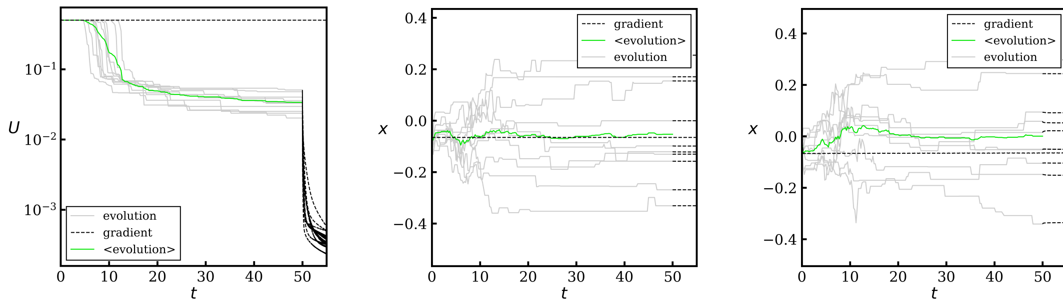


Figure 1.13: Left: Loss for GD and 25 MC trajectories trained on deep network of depth 32 and width 32. After $t = 50$, all MC trajectories are further trained using GD as shown by the black dashed lines. Middle and right: Evolution of two randomly chosen network parameters.

As networks get deeper, the gradient has a harder time passing through the network. This can lead to the gradient becoming small enough that no training occurs using gradient based methods. This *vanishing gradient* problem [21, 22] has to be taken into account when implementing deep networks. As shown in Figure 1.13, the deep network is stuck and fails to learn using GD. A high learning rate of $\alpha = 10^{-2}$ was used to try and give GD a stronger signal to aid learning. In contrast to GD, MC is still able to train without much difficulty as a vanishing gradient does not impact it. Once MC has trained the network and brought it to a more favourable region, the gradient becomes significant enough for GD to further train the network.

In the rightmost two plots of Figure 1.13, the evolution of two parameters is shown. The GD-only parameters do not change, as they do not experience any gradient. This is shown in the long black dashed line. For the MC parameters fine tuned with GD, only minor change to the parameters is made, as can be seen by the short black dashed lines connected to the individual MC trajectories. These minor adjustments are responsible for the significant reduction in loss shown in the first plot, which implies that the MC optimization had already brought the network near an optimal configuration of parameters. The reason MC might have failed to reduce the loss by such a significant amount might be the mutation size variance σ^2 being too large to make the needed fine adjustments done by GD.

This example shows a situation where GD fails in comparison to MC. While there exist many solutions that solve the vanishing gradient problem for GD to different degrees, they usually involve modifying the structure of the network itself. For example, skip connections [23, 24] which directly connect later layers of the network to earlier ones, allowing the gradient to have a pathway to "skip" layers and ensuring a stronger gradient is present throughout the network. This makes training deep network easier, as the gradient it less prone to becoming zero. That said, having the possibility to use any network architecture and still learn when the gradient is unreliable is a useful tool to have access to.

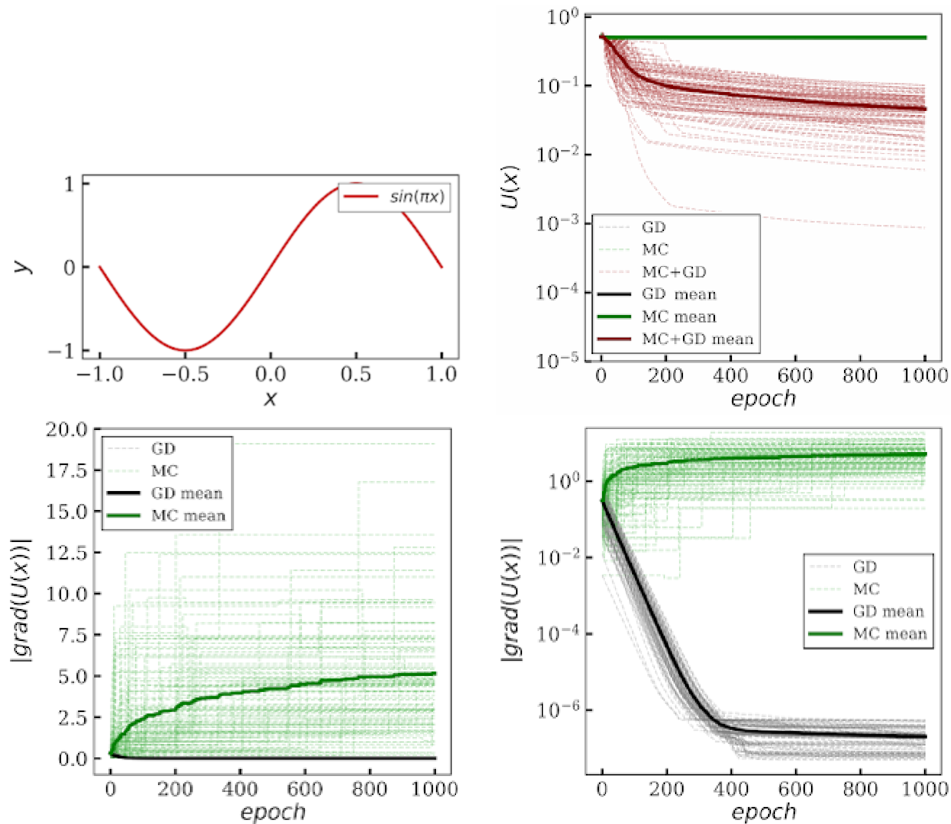


Figure 1.14: Depth 40 network trained on $sin(\pi x)$ using GD, MC, and alternating MC+GD. The magnitude of the gradient for GD and MC is shown in the bottom two plots.

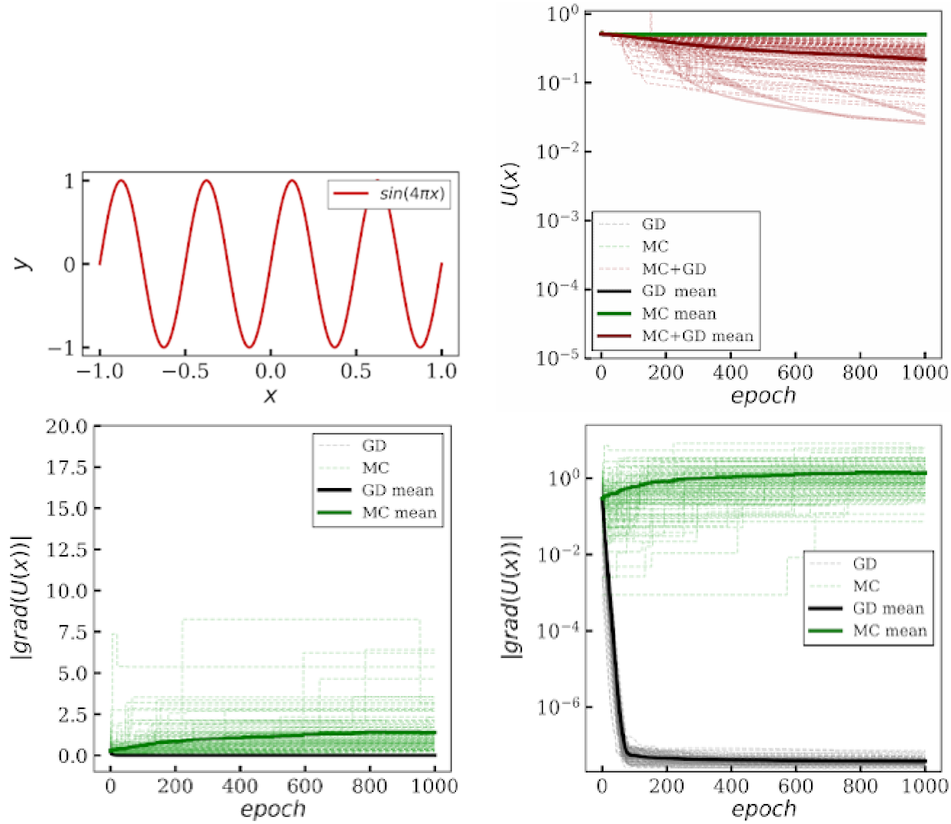


Figure 1.15: Depth 40 network trained on $sin(4\pi x)$ using GD, MC, and alternating MC+GD. The magnitude of the gradient for GD and MC is shown in the bottom two plots.

To further examine how GD is able to train after learning with MC, the magnitude of the gradient, equation 40, was calculated at every epoch for GD and MC. 100 trajectories of GD and MC was run starting from different initial parameters in a depth 40 network. In addition, a third training method utilizing both GD and MC was also used. This method switched which optimizer to use every 200 epochs, alternating between GD and MC. Figures 1.14-1.15 shows the results. The top plots show the target function and the loss evolution. This shows how both MC and GD fail to learn on its own at this depth, while MC+GD is capable. MC likely fails on its own due to either a too large σ, or the inherent difficulty in training very deep networks. The bottom two plots in each Figure show the magnitude of the gradient as described in equation 40. The magnitude of the gradient,

$$|\nabla U(\boldsymbol{x})| = \sqrt{\sum_{i=1}^{N} (\partial U(\boldsymbol{x})/\partial x_i)^2}, \tag{40}$$

rapidly goes to zero for GD, but increases when trained using MC.

To visualize how GD and MC differ, a pseudo-loss surface can be used to glean insight. This is inspired by research into visualising the loss surface of NNs [25–27]. These visualizations are useful for understanding the shortcomings the optimizers need to overcome. The massive number of parameters in

a network makes it difficult to generate intuitive visualizations of the loss surface. In this paper, as the behaviour of the optimizer is of interest, a simple test with a two-parameter pseudo-loss surface is used to see how both GD and MC trains in relation to each other. This loss surface is a simple function of two parameters, which is treated as the loss for the optimizer. Figures 1.16-1.18 show both GD and MC used on 3 different pseudo-loss surfaces. Both methods are shown with one small and one large learning hyperparameter (α for GD, σ for MC). Yellow corresponds to a high loss, while blue corresponds to the low loss. The two parameters are initialized to start at a point of high loss. These figures are purely for visually inspecting the optimizers' behaviour, and thus the scale and numerical values are irrelevant.

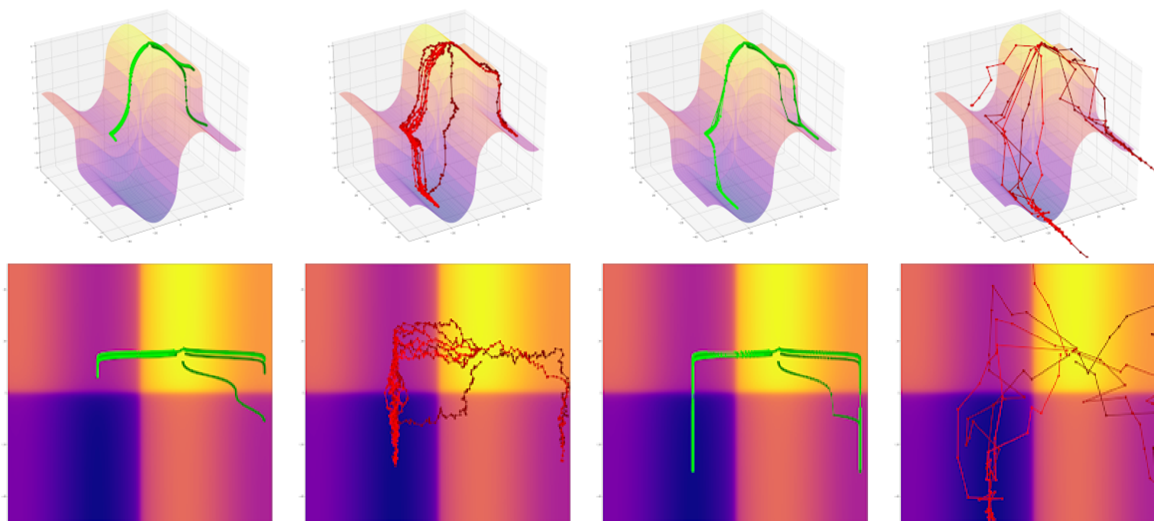


Figure 1.16: Example of GD and MC minimizing the value of the function $f(x,y) = arctan(x) + arctan(y) + sin(x/10)$ with parameters x,y. The yellow corresponds to high loss while the blue corresponds to a low loss. Both a 3-dimensional and top-down view are shown. Green is GD, red is MC. The left plots corresponds to small learning hyperparameters (α, σ), while the right plots corresponds to large ones.

Figure 1.16 shows a loss surface with a large gradient; GD easily follows the gradient down to the minimum. For small MC mutation σ, learning roughly follows the gradient in agreement with correspondence between GD and MC. For larger σ, MC takes large steps disregarding the local gradient. Therein lies the main difference between the two methods: MC allows for equally large steps to be taken in both the direction of the gradient as well as along the gradient contours. A move that keeps the loss constant is accepted in the same way as a move that reduces the loss. This allows MC to explore the local loss landscape, while still roughly following the gradient.

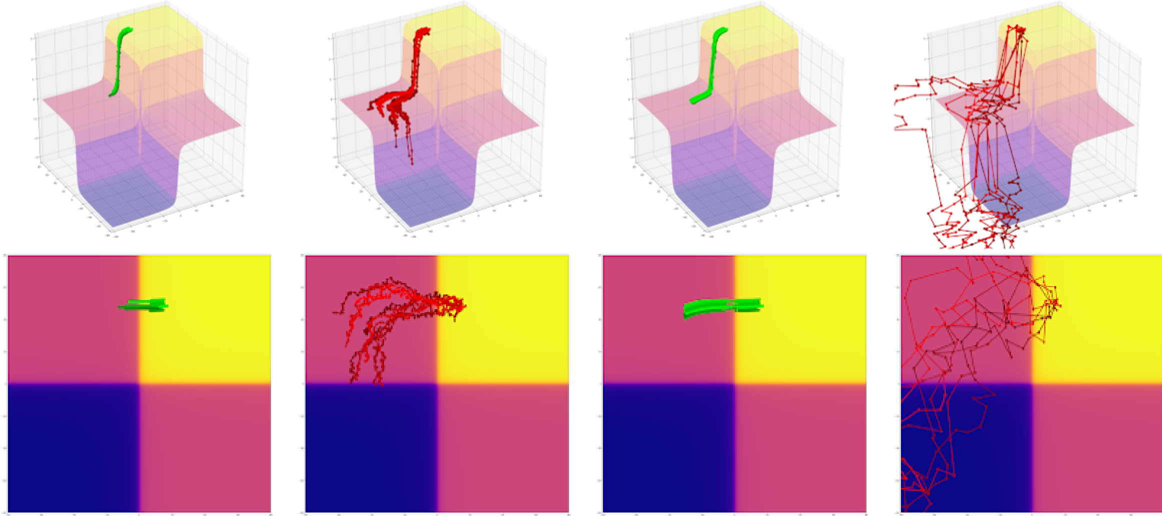


Figure 1.17: Example of GD and MC minimizing the value of the function $f(x, y) = arctan(x)+arctan(y)$ with parameters x,y. The yellow corresponds to high loss while the blue corresponds to a low loss. Both a 3-dimensional and top-down view are shown. Green is GD, red is MC. The left plots corresponds to small learning hyperparameters (α, σ), while the right plots corresponds to large ones.

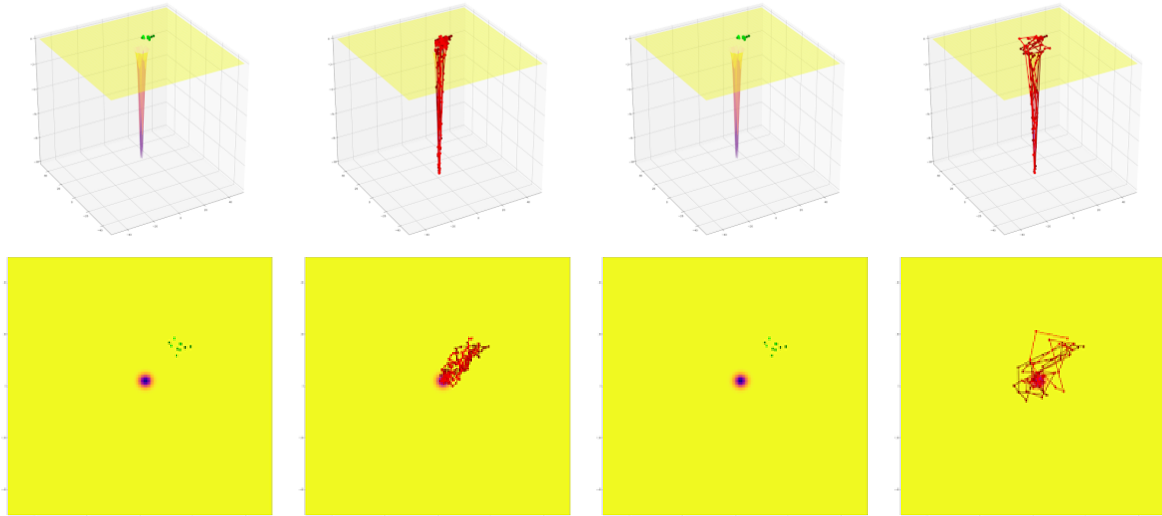


Figure 1.18: Example of GD and MC minimizing the value of the function $f(x, y) = -10 \times exp\left(-1/5 \times [(x-2)^2 + (y-2)^2]\right)$ with parameters x,y. The yellow corresponds to high loss while the blue corresponds to a low loss. Both a 3-dimensional and top-down view are shown. Green is GD, red is MC. The left plots corresponds to small learning hyperparameters (α, σ), while the right plots corresponds to large ones.

Figure 1.17-1.18 show examples of how MC is able to learn when gradients approach zero. Both pseudo-loss surfaces have a region of near-zero gradient. GD quickly finds itself stuck and unable to train while MC is able to explore the region until it stumbles down to the lower loss region. The exploratory nature of MC is evident in the large σ plot on the right of Figure 1.17, where it explores outside the slice of plotted loss. Figure 1.18 shows another example of MC being able to navigate the region of near-zero gradient and find the region of low loss at the center.

While these surfaces do not represent the true landscape found during learning, they manage to show key differences between the two algorithms.

1.4 Discussion

The correspondence between GD and MC show that two algorithm operating on completely different rules can functionally be extremely similar. The importance of this is that gradient based optimizers have received far more attention than the non-gradient based methods, even when baseline performance is similar. As non-gradient based optimizers have cases where they naturally outperform gradient based optimizers, investing research into improving them in a similar fashion to the gradient optimizer is a worthwhile goal.

1.5 Conclusion

These results successfully show numerical examples of the correspondence between GD and MC, as well as when it breaks. By numerically simulating both regimes of β, the analytical proof of correspondence is confirmed. The use of a well known and optimized machine learning package, in the form of PyTorch, ensures that the results shown have a high degree of confidence.

In the tests shown, the network had a single input and output. This was done to simplify the tests, and is not a limitation on the correspondence between GD and MC. Using varying sizes for the input, hidden and output layer, as well as changing the activation functions would still result in correspondence holding.

These results act as the motivation for the second project detailed in the next section. Snippets of the important code can be found in the appendix B.1, or the full code at *https://github.com/reproducible-science/MC-GD-correspondence.*

2 Adaptive Monte Carlo

2.1 Introduction

Throughout the years, many new techniques and tricks have been developed to further augment gradient based optimization. Non-gradient based optimization methods, while still used, have not had nearly as much development.

The primary objective of this project was to implement a version of MC that can compete with these popular improved gradient-based optimizers. The specific benchmarks we aimed for were:

1. Similar capability in training.
2. Train in the same or faster CPU time.
3. Have a similar level of complexity.

1 and 2 are necessary, as the optimizer needs to both have a similar capability to reach low loss and the ability to do so in a similar time. An algorithm that takes much longer to reach similar results is not as interesting. Finally, we desire that the implemented adaptive algorithm is not severely more complicated than the gradient-based competition. By this I mean that the additions made to MC should be logical and intuitive. Concretely, we want to limit the number of additional hyperparameters as much as possible to keep a simple and robust algorithm that does not rely on the user having to adjust a large number of knobs to get a working optimizer.

Of course, MC's ability to be useful in situations where gradients are zero, such as in very deep networks, is a desirable feature that should be kept. Therefore, nothing directly relating to the gradient should be added.

2.2 Methods

An "adaptive" optimizer is one with the ability to adapt to the current training circumstances. For example, a simple adaptive implementation might lower the GD learning rate based on the current loss. This is called an adaptive learning rate scheduler [28], and is an improvement on traditional schedulers which have pre-defined schedules based on factors such as the current epoch. Having an adaptive learning rate during training is a fundamental component of many adaptive algorithms as it makes the algorithm more generalizable to different learning scenarios. The non-adaptive schedulers necessitate fine-tuning on a case-by-case basis as one schedule will not be transferable to other ML problems.

2.2.1 Adaptive gradient optimization

The adaptive gradient based optimizer used as a benchmark for this project is *Adam* [29]. Adam is the prominent optimizer used in both research and industry, and has become the standard optimizer used in most deep learning projects. Therefore, it serves as an excellent benchmark to gauge the effectiveness of our adaptive MC optimizer.

Adam - The name Adam is derived from *adaptive moment estimation*, which also gives a hint as to how it functions. Adam uses first and second moments estimations of the gradient to adapt the learning rate of the network parameters on a per-parameter basis. These estimations are moving averages over past timesteps of the mean and the variance of the gradient with exponential decay and some additional bias-correcting components. Effectively, each parameter will have a different learning rate adapted to its current loss landscape. The update rule for Adam is [29]:

$$\frac{\partial x_i}{\partial t} = -\alpha \frac{\hat{m}}{\sqrt{\hat{v}} + \epsilon} \frac{\partial U(\boldsymbol{x})}{\partial x_i}, \tag{41}$$

where $\hat{m}$, $\hat{v}$ are the bias-corrected first and second moment estimates respectively. ϵ is a small value added for stability. The difference of Equation 41 compared to the equation for GD as shown in Equation 16 is that the learning rate is multiplied by the ratio of $\frac{\hat{m}}{\sqrt{\hat{v}}}$. This ratio is referred to as the *signal-to-noise* ratio as if there is a high certainty in the direction of the gradient (small variance), the step size will be boosted. If the direction of the gradient is unstable, the step size will be reduced. This ratio will also tend towards zero near a loss minimum, leading to better fine-tuning.

To further ensure a fair comparison, the optimized Adam implementation from PyTorch is used. This optimizer has been heavily used and tested, as can be seen in [30] which investigates optimizer popularity in research and compare the most popular ones, including Adam. Another comparison of different optimizer can be found in [31]. Therefore, Adam serves as an excellent benchmark to compare the proposed adaptive MC.

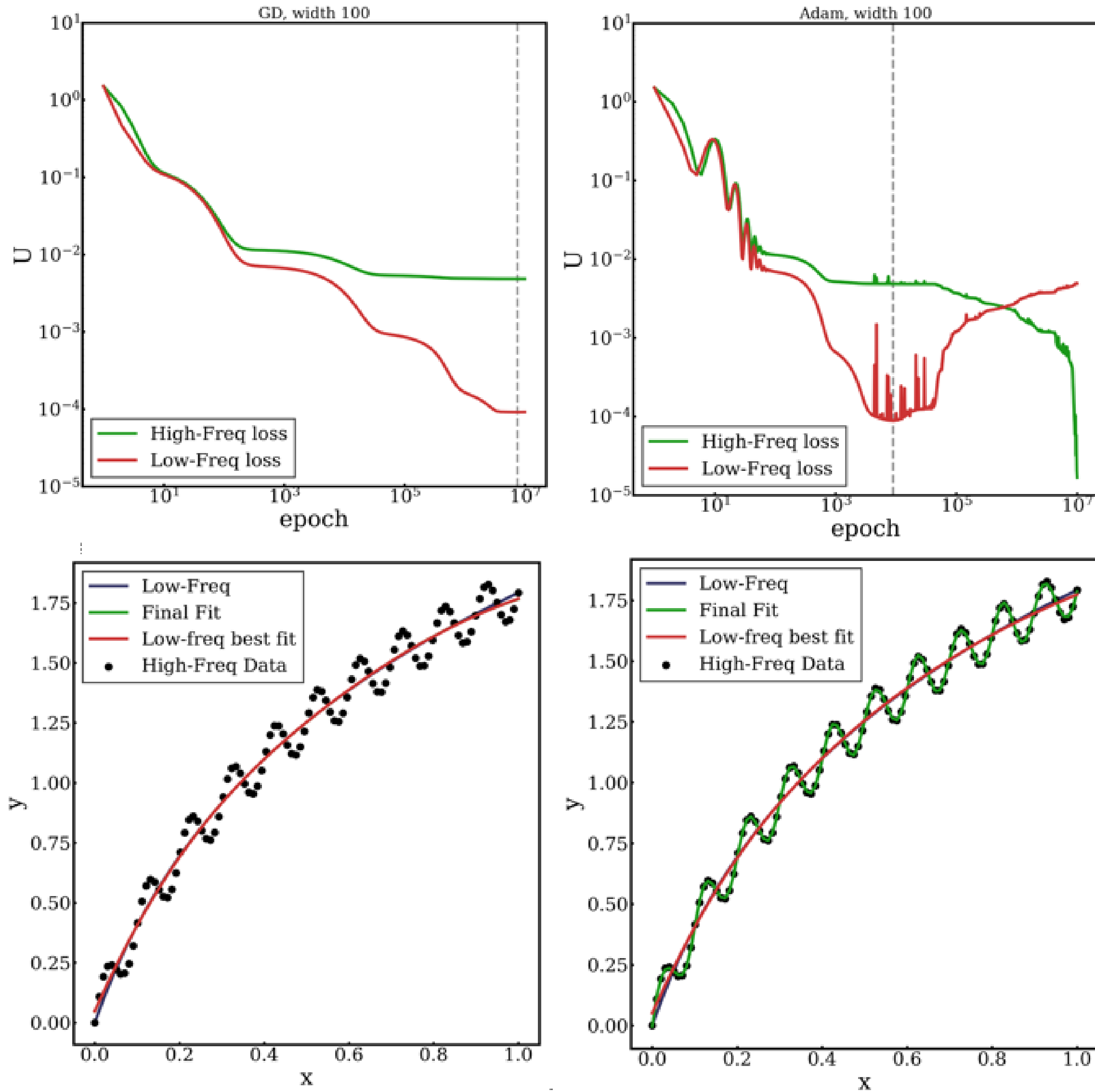


Figure 2.1: High-frequency feature test using GD (left) and Adam (right). Both optimizers were run for the same number of epochs. The dashed grey line denotes the minimal low-frequency loss. The final fit and the best low-frequency fit are shown in the bottom plots.

First we investigate how adaptiveness affects learning. In Figure 2.1 the effect of adaptivity is shown by training a network using both GD and Adam in a regression task on a summed function: a high frequency sin added to an underlying low frequency $\sqrt{x}$. The optimizer has the objective of learning this summed function, labelled "High-Freq loss". The loss with respect to the low frequency function, labelled "Low-Freq loss", is the loss of the low frequency component $\sqrt{x}$. This loss is only calculated for plotting and is not used during training. The network was of depth 1 and width 100, and had identical initial parameters for both optimizers.

Separating the loss this way gives insight into how the network is learning. One might expect the network to learn parts of the high frequency function in the early stages of learning, notably if the target

function is near the initial network output. However, this plot shows that the network instead learns the underlying low frequency function before learning any high frequency features, likely due to a much stronger signal pushing the network towards this simple function. This behaviour is shown in the Adam trained plot on the right, as the GD did not manage to learn the high frequency component in the chosen number of epochs.

This ability for Adam, and adaptive optimizers in general, to more easily learn high frequency features is however not always a desirable feature. Certainly, if the objective is to learn a complicated function as in this example, adaptivity is a beneficial addition. However, this ability to learn high frequency features also applies to any noise present in the training data. Adaptive optimizers are therefore prone to "overfitting" the training data, reaching very low loss at the detriment of the models ability to generalize. Instead of learning the general features of the data, it instead learns to memorize the training samples.

A number of techniques are commonly used in conjuncture with an adaptive optimizer, such as regularization to mitigate overfitting and the usage of a validation data set which is not used to update the network's parameters. By comparing the loss on the training set to the loss on the validation set, known as the validation loss, the network can be stopped when the network starts overfitting. As the network begins overfitting the training data, the validation loss will increase while the training loss continues to decrease.

2.2.2 Adaptive Monte Carlo

The main observation that prompted this project comes from Figure 1.12 shown in the MC-GD correspondence project. The Figure shows how the ratio of accepted to rejected moves decreases for MC as learning occurs. The optimizer has a harder time proposing moves which improve the network's performance as the loss decreases. We were curious to see if we could implement some simple additions to MC which would improve the chance of a good move being proposed by the optimizer, and how that would affect learning.

In the next section, I will describe two versions of an adaptive MC that I implemented and tested. These are referred to as adaptive Monte Carlo (aMC). The code for these implementations is in appendix B.2.

2.3 Results

2.3.1 Implementation of first version

In the first implementation of aMC, we sought to experiment with different additions to gauge their effectiveness. In this first version, we decided on 3 additions to implement. Two of these were not kept in the second version of aMC, and I will therefore only briefly comment on them, explaining the reasoning behind them and why they were discarded.

Change 1: The first addition was to limit the number of parameters mutated in a single move. To do this, for each parameter in the network, randomly decide if the parameter is chosen with some probability $p = U(0,1) < f_{chosen}$. The hyperparameter f_{chosen} determines how many parameters will be selected. If $f_{chosen} = 0.5$, then roughly half of the parameters will be selected.

The logic behind this change was the belief that mutating a smaller number of parameters each move would increase acceptance. However, through testing, we determined that this was not the case. Seemingly, the number of parameters selected did not affect the overall acceptance of proposed moves. This addition was therefore not worth the increase in training time needed as fewer parameters were updated each move.

Change 2: The second change is more significant. This change relates to how the learning rates are adapted on a per-parameter basis for Adam. The goal is to adapt the mean and σ of the proposed mutations on a per-parameter basis. This change was kept in the second version of aMC, albeit altered.

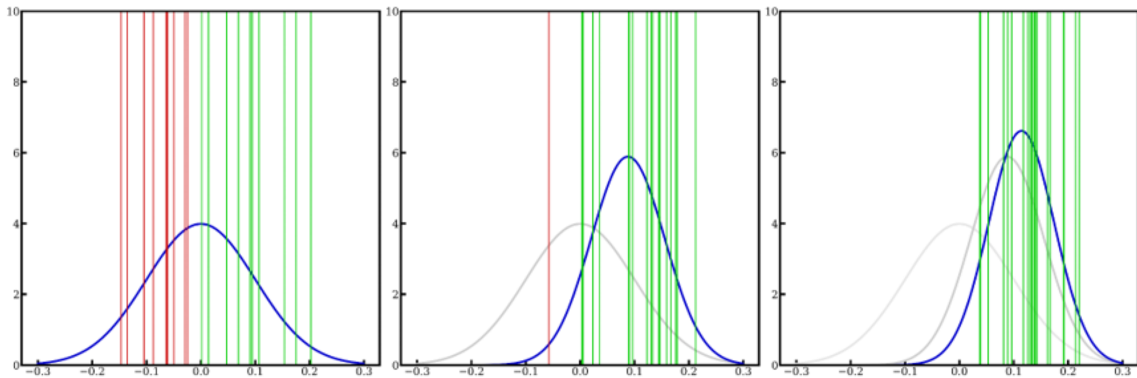


Figure 2.2: Example of the evolution of the mutation distribution for the first version of aMC. Moves are drawn from the current distribution (blue) and are either accepted (green) or rejected (red) based on the loss. A new distribution is calculated every N accepted moves. The distributions in grey shows the previous distribution.

As opposed to MC where mutations are sampled from a single Normal distribution with zero mean and variance σ^2, aMC implements a shifting distribution. Figure 2.2 shows and example of how a parameter's distribution adapts as learning occurs. A list of accepted mutations is kept for each parameter.

After a parameter has experienced N accepted mutations ϵ_i, the mean μ and standard deviation σ are recalculated, with σ using an unbiased estimator.

$$\mu = \frac{1}{N}\sum_{i=1}^{N}\epsilon_i, \tag{42}$$

$$\sigma = \sqrt{\frac{1}{N-1}\sum_{i=1}^{N}(\epsilon_i - \mu)^2}. \tag{43}$$

This new distribution will then be more likely to propose mutations that have previously been accepted, leading to an increase in acceptance. Finally, the mean and σ is reset to the initial values $\mu = 0$, $\sigma = \sigma_{initial}$ if a number of consecutive proposed mutations are rejected. This prevents the network from getting stuck when the direction of accepted mutations changes and the distribution is unable to propose mutations on the other side of zero. Put together, this algorithm tries to maximize the probability of proposing accepted mutations by shifting the distributions mutations are drawn from.

Change 3: The final change is the concept of escape moves. Instead of simply rejecting moves, additional mutations are proposed on top of the rejected one, up to a limit. The idea is to give the optimizer a better ability at escaping from tough features in the loss landscape, such as climbing over a hill. If an escape move is accepted after some number of rejected mutations, it is not used for the purpose of the scaling mean and σ.

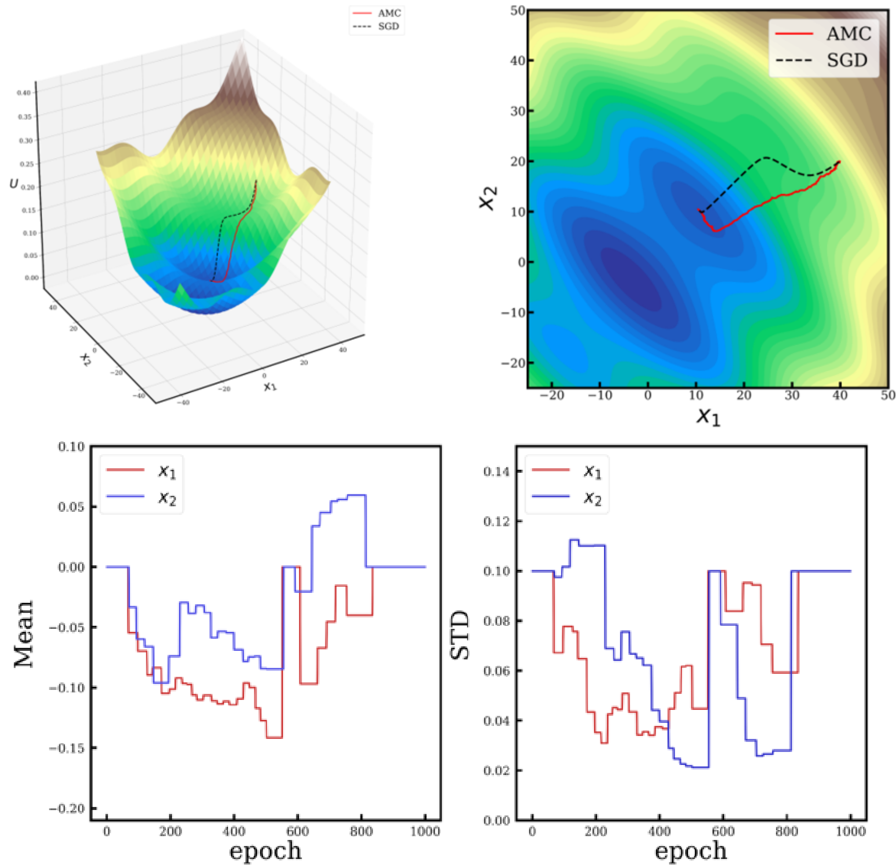


Figure 2.3: Behaviour of version 1 of aMC and GD optimizing a pseudo-loss surface composed of 2 parameters x_1 and x_2. Bottom plots show the evolution of the aMC parameter specific mutation means and σ. The optimizers are minimizing the loss, going from high loss (white,brown) to low loss (blue).

As before, a pseudo-loss surface is a useful way to glean insight into the behaviour of an optimizer. In Figure 2.3, GD and the first version of aMC is shown optimizing a loss surface comprised of 2 parameters x_1 and x_2. As the function is known, GD directly uses the gradient of the function to reach the minima, while aMC operates as usual, proposing mutations and accepting/rejecting them based on the previous loss. In contrast to vanilla MC, aMC does not follow the gradient and instead takes a different path to the minima.

The bottom plots show how the mean and σ evolve for the mutations of the two parameters. Importantly, the individual parameter means tend towards the negatives for both parameters, which is the correct direction for both parameters. As the trajectory continues, the σ becomes smaller and smaller, implying a greater certainty in this chosen direction. Eventually the parameters reset after a number of consecutive rejected moves, which then leads to the x_2 mean switching to a positive direction as shown in the loss trajectory. This example shows how this adaptive mean and σ work together to bias MC to propose moves with a higher acceptance. It also shows how the resetting of the mean and σ is important to give the optimizer the ability to change direction, as otherwise it would get stuck.

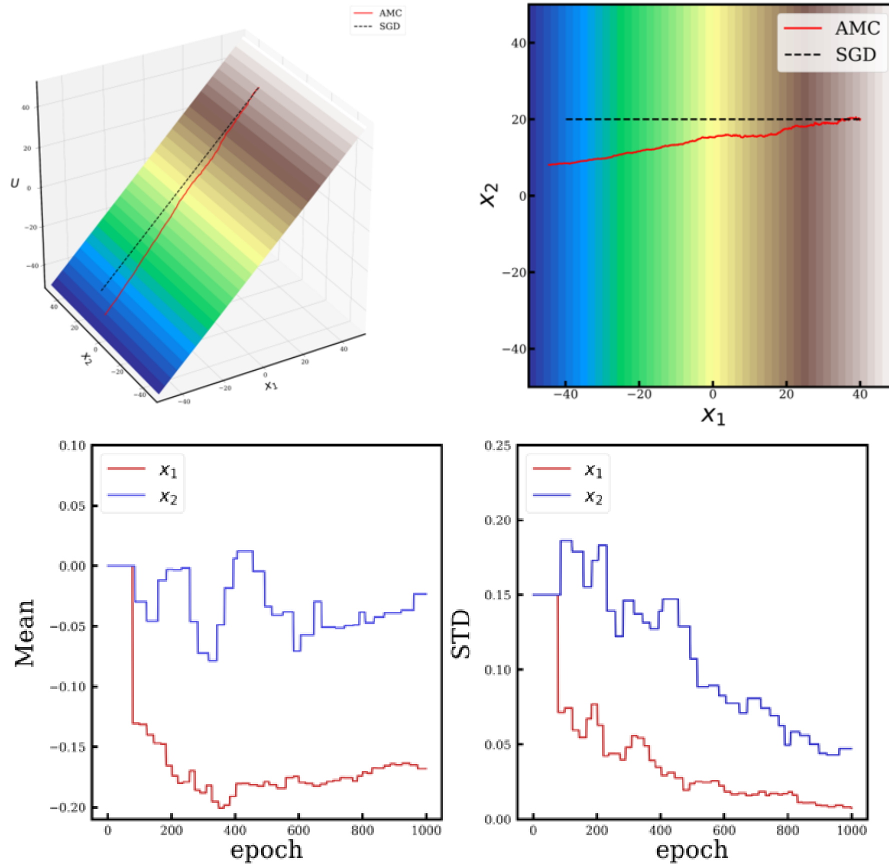


Figure 2.4: Behaviour of version 1 of aMC and GD optimizing a linear pseudo-loss surface composed of 2 parameters x_1 and x_2. Bottom plots show the evolution of the aMC parameter specific mutation means and σ. The optimizers are minimizing the loss, going from high loss (white,brown) to low loss (blue).

This break of the GD-MC correspondence can be further visualized by using a linear surface as shown in Figure 2.4. This pseudo-loss surface simulates a situation where a parameter is currently not relevant for learning, and can therefore vary in either direction without affecting the loss. While this is a perfect theoretical example of this, in reality one might imagine a parameter's impact on the overall loss being much smaller than other parameters, effectively having the same effect. What we observe here is a major difference between this version of aMC and MC. As the x_2 parameter has no preference in its direction, the mean and σ is free to change. From the bottom plots, we can see that the mean of the mutation for x_2 hovers around zero, but due to the fact that these values are calculated from a limited number of samples, the σ diminishes until the mean gets stuck at a non-zero value. This introduces a permanent drift in x_2. In addition, even if x_2 had a slope in a particular direction instead of being flat, if that slope was significantly smaller than the slope of x_1 it could get stuck climbing upwards as the overall loss would still be decreasing.

While this implementation showed potential, there was a few problems we sought to fix for the second version. Primarily, keeping a list of hundreds of mutations for each parameter requires a significant amount of memory, and is therefore less useful as the network parameter count increases. In addition,

the way the σ were scaled resulted in the mean becoming stuck. The mean and σ resetting at the same time also was not useful for learning at low loss. In this situation, we would want the σ to be small as the network is more sensitive to perturbations. For these reasons, the way the mean and σ are scaled was changed. The other additions were removed as they did not significantly impact either the acceptance or the training time. This includes the escape moves as well as the probabilistic selection of parameters to mutate.

2.3.2 Implementation of second version

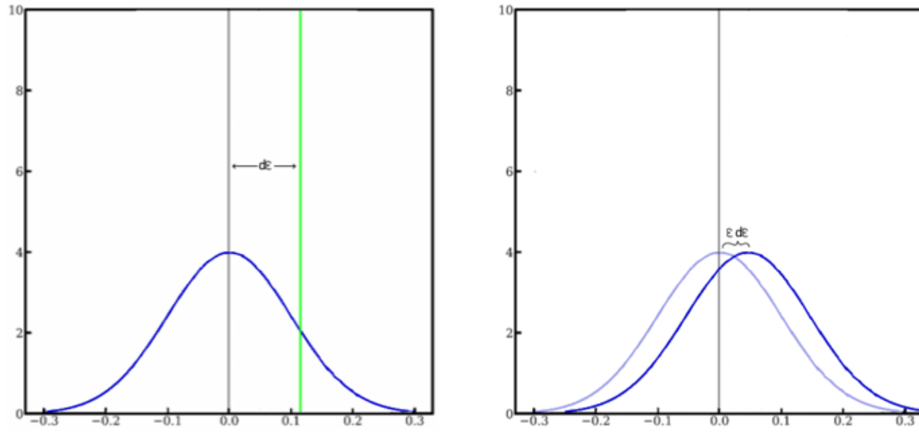


Figure 2.5: Example of the evolution of the mutation distribution for the second version of aMC. Moves are drawn from the distribution (blue) which is shifted every time a move is accepted (green). The shift corresponds to a fraction of the distance of the mutation from the mean.

With most additions removed, the second implementation of aMC also decouples the σ scaling from the mean. In the first version, both of these values were calculated from the list of previously accepted mutations. As explained previously, this had negative consequences relating to the ability to learn. In this version, the method for the scaling of the mean and the σ was changed.

To start, the σ is no longer tied to previous accepted mutations. Instead, it is scaled by a factor after N consecutive rejected moves. We found that using 100 rejected moves before scaling led to good results. After many consecutive rejected moves, the σ is scaled as:

$$\sigma = 0.95 \times \sigma. \tag{44}$$

This means that as training occurs, σ will naturally diminish as the loss decreases and the network becomes more sensitive to perturbations. With a smaller σ, the mutation size will also be smaller as the mean similarly resets when the scaling occurs. This makes the optimizer better at making minute adjustments to the parameters.

Figure 2.5 shows how the mean of the mutation distribution (blue) is scaled. When a move is accepted, the mutation distribution mean is shifted towards the accepted mutation (green). To do this, the difference between the mean and the accepted mutation $d\epsilon$ is calculated, then the mean is shifted by a fraction of this: $\epsilon d\epsilon$. In practice, we used $\epsilon = 0.01$ for the shift amount. This process happens on a per parameter basis, so each parameter will have a unique mutation distribution based on the accepted mutations. To prevent the algorithm from getting stuck, the mean resets to zero if N consecutive rejected moves occurs.

The final addition is the implementation of what we call Signal Norm. The main goal of this addition is to normalize the impact mutations have across the network. As the network parameters have different amounts of input connections, the magnitude of the change in the input of a parameter varies across the network.

For a neuron j, the mutation σ of all weights i feeding its input is scaled as:

$$\sigma_i = \lambda_i \sigma, \tag{45}$$

where λ_i is:

$$\lambda_i = \left(N_{data}^{-1} \sum_{\alpha=1}^{N_{data}} \sum_{i' \to j}^{N_j} \left(S_{i'}^{\alpha} \right)^2 \right)^{-1/2}, \tag{46}$$

where $S_{i'}^{\alpha}$ is the output of neuron i, N_j is the number of neurons that feed j, N_{data} is the number of data samples in a given iteration of training α.

Therefore, we are calculating the magnitude of the outputs of the neurons which feed j and scaling the mutation size σ on these neurons to ensure the input signal of j has the same magnitude for all weights in the network.

The full derivation for the signal norm is explained in more detail in the aMC paper [32]. Signal norm was conceived by Dr. Stephen Whitelam, which I implemented for testing.

Signal norm bears some resemblance to a technique sometimes used called Layer normalization [33] which normalizes the summed input to neurons based on a computed mean and variance. The main difference with our implementation is that it is used only during training instead of being an architectural change in the network, and is constructed for use with MC.

To recap, this second implementation has many benefits in comparison to the previous one:

- Separate scaling of mean and σ of mutations means that the algorithm does not get stuck in unnecessary directions.
- Every move affecting the mean renders the algorithm more responsive to change.
- Not keeping track of accepted moves is more memory efficient.
- The ϵ hyperparameter can be tuned to control the sensitivity of the algorithm. In practice we found keeping $\epsilon = 0.01$ to be sufficient.
- Signal Norm normalizes the impact mutations have across the network, improving learning.

As signal norm was implemented later, the following results of the second aMC version are shown without signal norm.

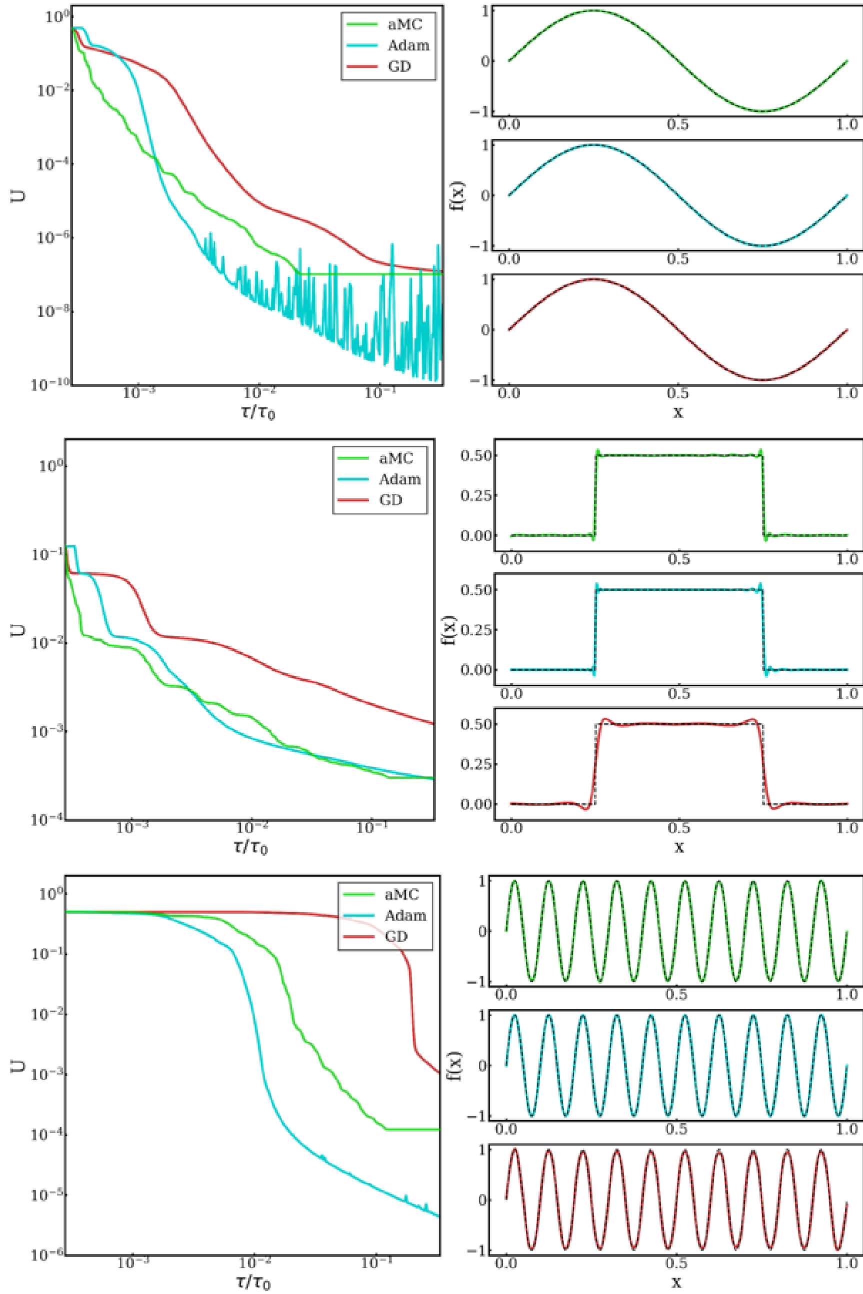


Figure 2.6: Results of aMC, Adam and GD trained on three different functions. Top: $sin(2\pi x)$, middle: *squarewave*, bottom: $sin(20\pi x)$. The corresponding loss evolution is shown on the left.

To test the effectiveness of aMC on a simple problem, 3 target functions are used: a low and high frequency *sin*, and a square function as shown in Figure 2.6. For each optimizer, the same network parameter initialization is used. For GD and Adam, the learning rate was manually chosen to try and minimize training time while still keeping a relatively stable training. The learning rate for Adam was 10^{-4}, while the learning rate for GD was 10^{-2}. For aMC, $\sigma = 10^{-2}$, $\epsilon = 10^{-2}$, $N_{scale} = 100$, where N_{scale} is the number of consecutive rejections before rescaling. All networks were trained for a total of 10^4 epochs. Finally, the chosen network had a single hidden layer of width 40.

From these results we can see that aMC performs similar to Adam on this test problem. These results are purely optimizer-based and do not involve other regularization techniques or architectural changes. This means that aMC is able to train the network with efficiency closer to Adam as opposed to GD. This result shows that aMC has potential as an optimizer, as similar non-optimizer based improvements that are used in conjuncture with Adam could be implemented for aMC.

One observation of note is the loss plateau aMC reaches towards the end of training. Seemingly, aMC stops training once the loss reaches very low values. After extensive testing, the cause of this was determined to be σ becoming too small, causing mutations to be too small for the 32-bit floating point numbers used. Once it reached this point, the problem compounded as no moves were accepted and σ therefore quickly tended to zero, freezing training.

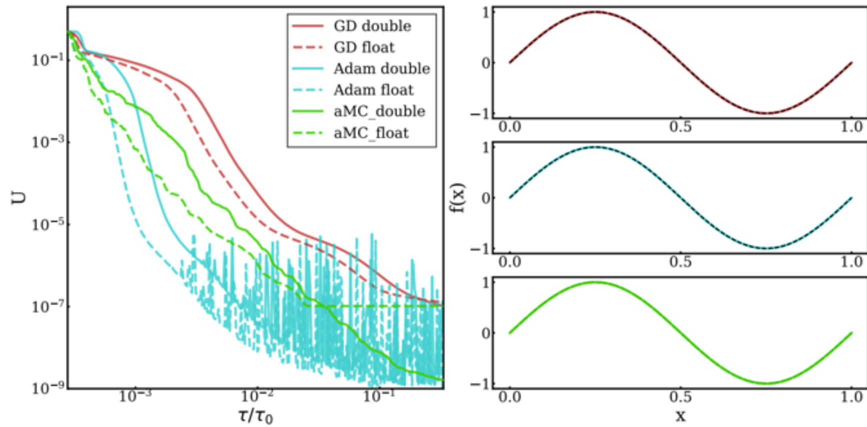


Figure 2.7: 32-bit and 64-bit floating point numbers test for aMC, Adam, and GD trained on $sin(2\pi x)$.

The precision of the floating point numbers being the cause for the loss plateauing comes from the results in Figure 2.7. This Figure shows results for all three optimizers trained using floats (32-bit) and doubles (64-bit) precision floating point numbers. GD and Adam perform similarly with both representations while aMC is able to continue learning when using doubles. Using doubles does not negatively impact training, and only requires more memory to store the increased parameter size. While more memory is required, it is not a drastic downside. Another working option is to limit σ to not decrease beyond some small value which will ensure continued training. This allows floats to be used at the cost of limiting how small σ can reach and potentially introducing a lower bound to the loss.

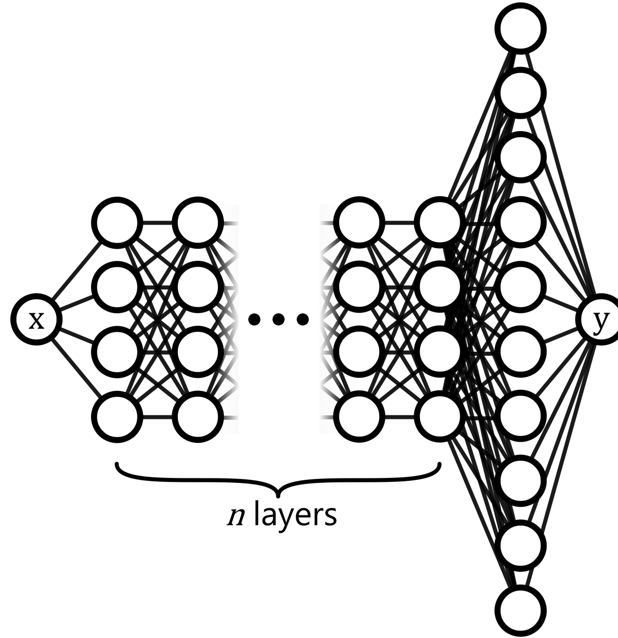


Figure 2.8: Architectural layout for a so-called mushroom network, referenced to as a mushnet. A series of thin hidden layers is followed by a wider hidden layer before the output layer, allowing very deep networks to be trained in a reasonable amount of time.

One of the major benefits of using a gradient-free optimization algorithm is the natural bypass of the vanishing gradient problem, as mentioned previously. To investigate the effect of aMC on very deep networks, a custom fully-connected neural network is used, as shown in Figure 2.8. This network has a number of thin hidden layers followed by a single wider layer before the output. By adding more thin layers, the depth of the network can be significantly increased without significantly increasing training time due to the increased number of parameters. The final wider layer provides more expressibility for the output. For our tests, we chose a thinner width of 4 and a final wider layer of width 10. These values were selected empirically for stability and ease of training. For these networks, the depth refers to the number of thin hidden layers + the single wider hidden layer.

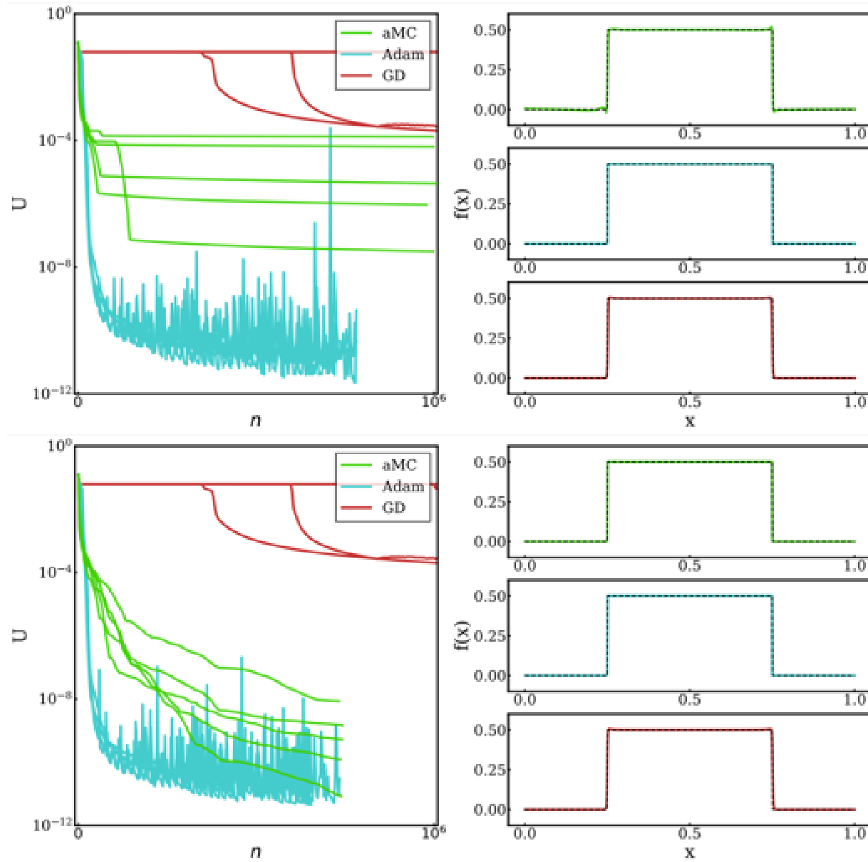


Figure 2.9: Results for aMC, Adam and GD trained on a square wave function using a mushnet of depth 8. Results for floats (top) and doubles (bottom) are shown.

Figure 2.9 shows a depth 8 mushnet trained on the square wave function. Using floats (top plot), the aMC trajectories stop learning around a loss of 10^{-4}. When switching to doubles (bottom plot), aMC continues to learn and stays competitive with Adam. GD is already incapable of reaching low loss at this depth. These networks were trained for a set amount of time rather than a number of epochs. As can be seen, aMC trains for a number of epochs similar to Adam.

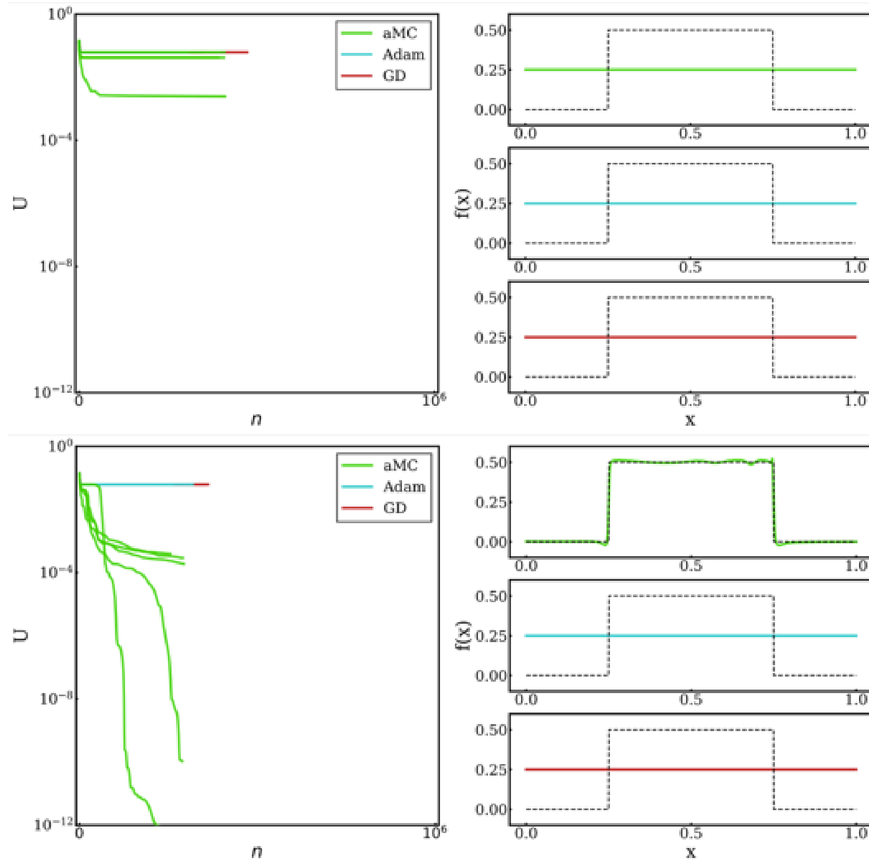
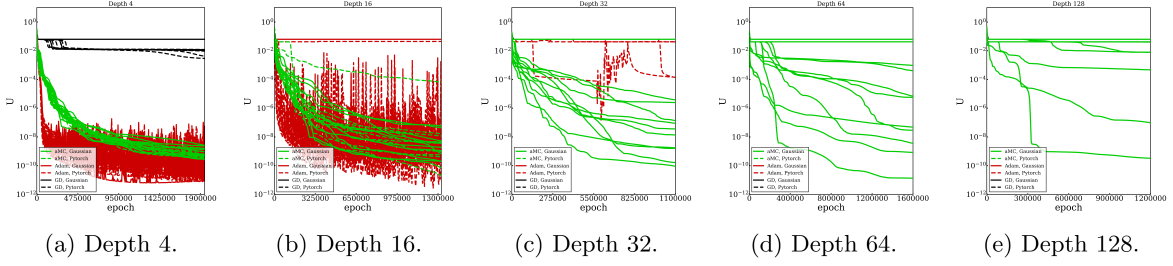


Figure 2.10: Results for aMC, Adam and GD trained on a square wave function using a mushnet of depth 32. Results for floats (top) and doubles (bottom) are shown.

When a much deeper mushnet is used, as in Figure 2.10, only aMC is able to learn. This Figure shows a mushnet of depth 32 which was trained for the same amount of time as the previous Figure. As there are more parameters in the network, training occurred for less epochs. Using only the optimizers, Adam and GD is unable to train this network. While there exists methods to allow these deep networks to be trained using gradients, these involve architectural changes to the network, which we are not investigating here. The property of the optimizer itself is of interest as we can glean insight into the limits of the optimizers and compare the two methodologies.

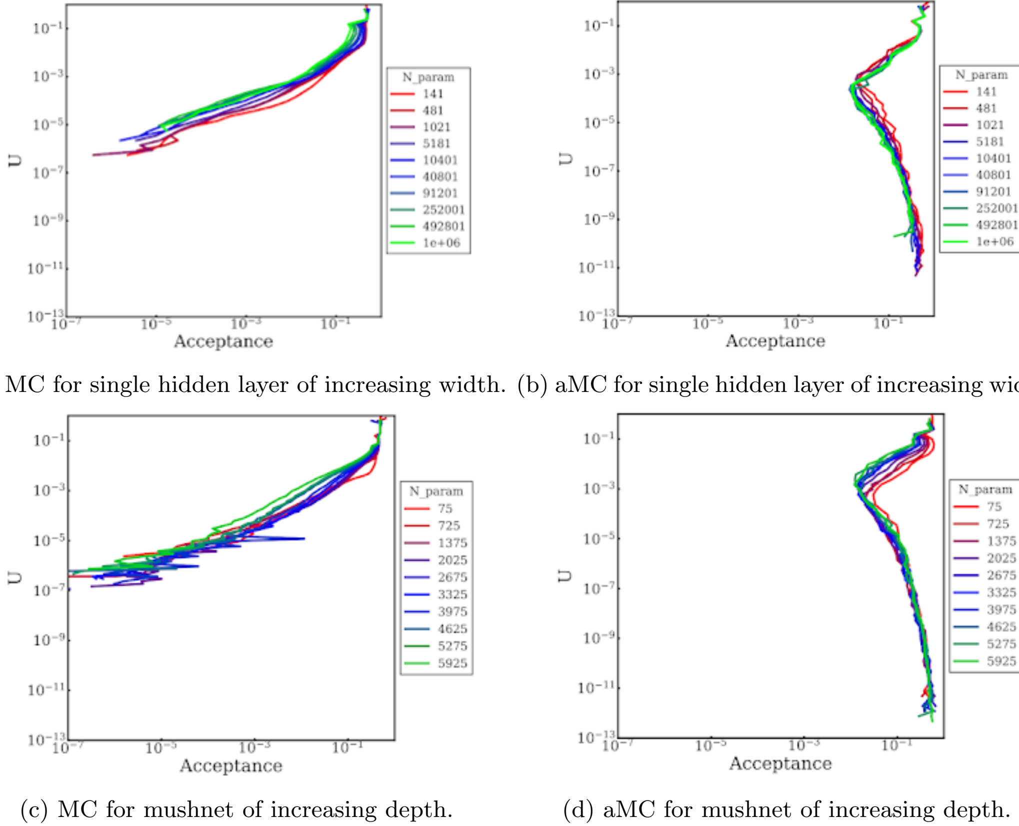
While not as important, a difference we noticed between aMC and Adam is that a Gaussian initialization of the networks parameters helped aMC, while the standard PyTorch uniform initialization was better for Adam. For fairness, we used the standard PyTorch initialization for both, as that is the commonly used method.



(a) Depth 4. (b) Depth 16. (c) Depth 32. (d) Depth 64. (e) Depth 128.

Figure 2.11: Mushnets of depth 4, 16, 32, 64, 128 trained on the square wave function for 10^6 epochs using aMC, Adam and GD. Solid lines corresponds to Gaussian parameter initialization while dashed lines correspond to default PyTorch initialization.

Figure 2.11 shows the results of training mushnets of depth 4, 16, 32, 64, 128 trained on the square wave function for 10^6 epochs. In the results without signal norm, aMC managed to train the network in a few trajectories. With the addition of signal norm however, almost all trajectories for depth 32 and 64 successfully train. Even mushnets of depth 128 are possible to train.



(a) MC for single hidden layer of increasing width. (b) aMC for single hidden layer of increasing width.

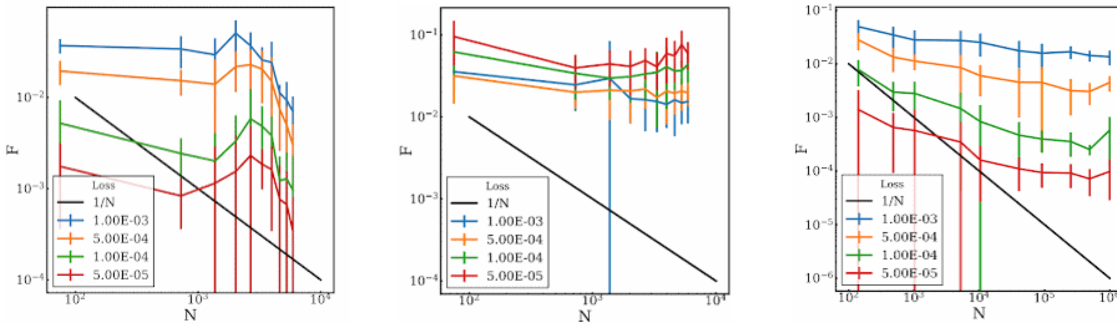
(c) MC for mushnet of increasing depth. (d) aMC for mushnet of increasing depth.

Figure 2.12: Acceptance based on loss for MC and aMC for two cases: increasing width of a single hidden layer (top), and increasing depth of a mushnet (bottom).

While the ability for aMC to train deep network when gradients are unreliable, the main objective of the adaptive implementation was for aMC to compete with Adam in terms of both training speed and ability. While I have shown aMC able to keep up with Adam, Figure 2.12 provides the explanation as to why.

In the top plots, the acceptance for both MC and aMC is shown based on the loss. Each line represents the mean of 100 networks with a single hidden layer of increasing width from 10 to 1000, which is represented by the number of parameters. The first conclusion to draw from these plots is that the number of parameters does not significantly affect the acceptance. This is a surprising result, as one might think that it would be harder to propose mutations which lower the loss as the number of parameters increase. The second observation to note is that for MC the acceptance rapidly drops as training happens. Once the loss reaches a value around 10^{-3}, it becomes very rare for a move to be accepted. In contrast, the adaptive elements of aMC is able to mitigate this decrease in the acceptance to keep the chance of proposing a good move relatively similar as a function of loss. A similar conclusion is drawn from the bottom plots, showing the same test for a mushnet of increasing depth from 1 to 10.

This highlights a difference in how adaptivity differs between gradient and non-gradient based methods. While for gradient based algorithms the main efforts involve making more efficient use of the gradient, non-gradient based algorithms of the type shown here involves increasing the quality of proposed moves.



(a) MC loss slices for mushnet of increasing depth.

(b) aMC loss slices for mushnet of increasing depth.

(c) MC loss slices for single hidden layer width.

Figure 2.13: Acceptance as a function of the number of parameters. Each line corresponds to a loss slice. The black line is the inverse of the number of parameters.

As mentioned, the number of parameters does not severely affect acceptance. This is shown in Figure 2.13. In this Figure, the acceptance for different loss slices, horizontal slices of the plots in Figure 2.12, is plotted. This shows how the acceptance varies for increasing total number of parameters in the network at different loss levels. For the single hidden layer of increasing width shown in the bottom plot, the acceptance stays relatively flat as a function of the number of parameters. The black line corresponding to $1/N$, where N is the number of parameters, represents a prediction of what the acceptance should be were it the case that acceptance was affected by the number of parameters. For the mushnet of increasing depth shown in the top plots, the same initial behaviour as for the width is observed for the small depths, but acceptance rapidly drops for higher depths as MC performs worse. This reduction in performance at higher depth might be a result of an increase in the network's sensitivity to perturbations, as the earlier weights have a direct impact on the rest of the network. This means it becomes increasingly difficult to

propose good mutations without any form of adaptiveness as larger fraction of the weights need receive a good mutation. In contrast, aMC manages to keep acceptance high for all loss slices.

2.4 Discussion

While there is room for improvement in the adaptive MC algorithm shown above, the current implementation clearly shows how these non-gradient based methods can compete with the most widely used gradient optimizer. More complicated ML problems were done by a colleague which demonstrates how aMC compares to Adam on the well documented task of classifying digits in the MNIST dataset [34] . These results are shown in the aMC paper [32].

2.5 Conclusion

In conclusion, we found that this adaptive version of MC is useful for training NNs both as a alternative for popular gradient based methods but also in situations where these methods fail. While there exists improvements to ensure these gradient based optimizers work in these tricky situations, it is still interesting to note the existence of optimization methods which require no further assistance. At the very least, aMC is a useful algorithm to have access to if the need for an easy way to train a network with no gradients and limited memory arises.

Snippets of the important code can be found in the appendix B.2, or the full code at *https://github.com/reproducible-science/aMC*.

3 Growth Pipeline for Targeted Design

3.1 Introduction

With a non-gradient based optimizer with similar performance to gradient based ones, we can now tackle a more difficult inverse design problem that involves a physics-inspired scoring simulator: physical solvers that simulate real processes as accurately as possible. Inverse design is a popular method to developed designs to solve a wide range of problems by defining an objective and using an algorithm to optimize a solution. Instead of directly designing the solution, we are interested in an algorithm that produces solutions. In particular, material science is one field where ML has been found useful in proposing new and interesting materials [35, 36] and metamaterials [37–39], as well as nanoparticles [40] and molecules [41] amongst others. Inverse design is also often used for designing structures, such as in photonics [42,43] for photonic chips [1], waveguides and antennas [44]. Commonly, these implementations care primarily about the end design and therefore directly produce potential candidates irrespective of how easily they can be fabricated. The creation of these designs is then left to expensive machinery. Specific constraints can be added to the inverse design process to impose design limitation on the final design to help with reproducible in real life. We introduce a pipeline that seeks to produce useful and realistic designs without using constraints on the final design. Instead, we seek to control the growth of an empirically accurate growth simulation to produce the finished design. Ensuring the growth parameters stay within realistic ranges is therefore a constraint that we impose. The end result of the pipeline is a schedule on how to alter the growth parameters during the growth to produce designs that accomplish some target objective. This schedule is referred to as a policy. These policies could then theoretically be used in real life to replicate the designs using whatever method most closely matches the growth simulation. As this is a pipeline, the individual components, such as the growth simulation or physical solver, can be swapped out without much effort. The pipeline is used in this thesis to design photonic chips to act as wavelength splitters or bandpass filters. A realistic growth simulation is used to produce structures, after which light is simulated passing through them to produce a score based on the objective.

Implementation - All parts of the pipeline were implemented in Python. Modularity was the primary design objective to ensure that each components could be easily swapped if so desired. Where available, pre-existing implementations were used to ensure accuracy and speed. Of note, PyTorch was used for the NN, while PyMeep was used for the Finite Difference Time Domain (FDTD) light simulations. The growth algorithm was manually implemented as explained in the next section. The code is shown in appendix B.3.

3.2 Methods

We begin by introducing the general pipeline and the role of its components. Following this, the chosen components used for the results are explained in detail.

3.2.1 Growth pipeline

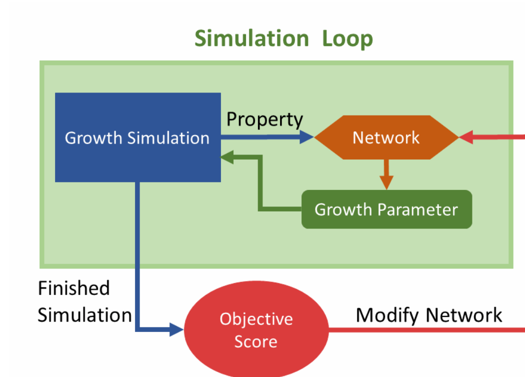
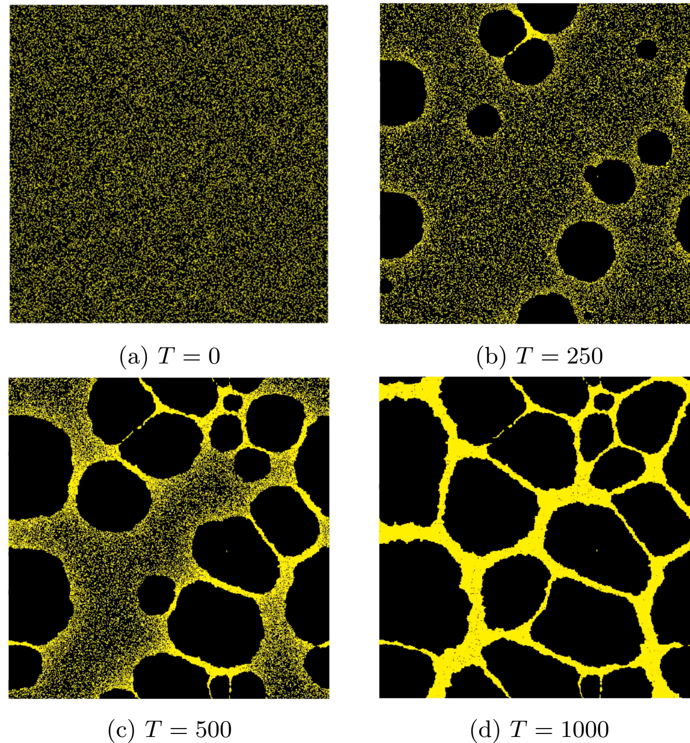


Figure 3.1: Schema for controlled growth pipeline. The simulation loop occurs during the growth process. Scoring and updating the network is done after the growth is finished.

The general growth pipeline is shown in Figure 3.1. In the simulation loop, some property of the growth model is periodically exposed and passed to a network which outputs what the desired growth parameters should be changed to. This cycle repeats until the simulation is finished. The finished growth is then evaluated based on some objective score and the performance is used to tweak the model in a direction to improve the objective score.

3.2.2 Growth model

The growth model used in this project was introduced in [45]. The purpose of the model is to simulate nanoparticles suspended in an evaporating solvent. The system is shown to generate structures which empirically match experiments. For this reason, it is an interesting system to simulate for the purposes of targeted growth as the grown structures could be replicated using the generated policies. In addition, we implement some alterations proposed in [46] which include the next-nearest neighbours in the energy calculations to further improve realism.

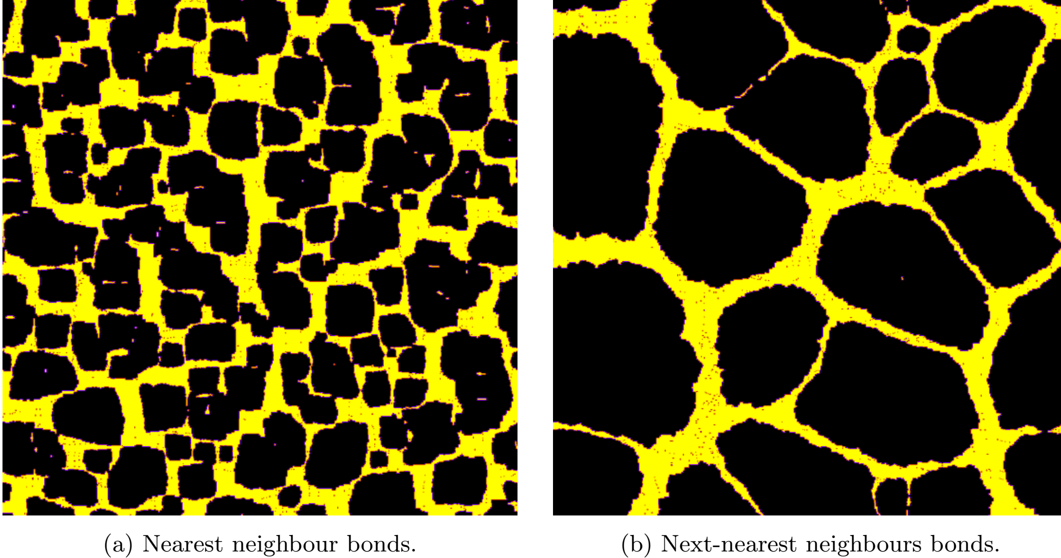


(a) $T = 0$ (b) $T = 250$ (c) $T = 500$ (d) $T = 1000$

Figure 3.2: Snapshots of the growth simulation for $k_B T = 0.2$, $Frac = 0.2$. Nanoparticles are shown in yellow while solvent/air is shown in black.

An example of a growth simulation is shown in Figure 3.2. The initial state (top left) and finished growth (bottom right) are shown with two times taken between. The main mechanism behind growth can be seen in this plot. The nanoparticles (yellow) are initially randomly scattered inside the solvent. Eventually, a solvent cell evaporates which energetically encourages the neighbouring cells to do the same while pushing away any adjacent nanoparticles. This leads to air bubbles growing while nanoparticles migrate away from the newly created air-solvent interface. The rate at which new nucleation occurs in relation to how fast the nanoparticles move determines the structure the finished growth will form. In this Figure, both solvent and air are shown in black, as to emphasize the nanoparticle movement.

An example of growths produced with nearest and next-nearest bonds is shown in Figure 3.3. Without the next-nearest bonds, the grown structures tend towards squares and form less organic structures due to the lattice grid having a significant effect on the growths.



(a) Nearest neighbour bonds. (b) Next-nearest neighbours bonds.

Figure 3.3: Finished growths produced using only nearest neighbour bonds (left) and additionally with next-nearest neighbours bonds (right). Nanoparticles are shown in yellow, air in black.

The system is represented by a pair of two-dimensional arrays, a solvent array l and a nanoparticle array n. A cell i either contains solvent ($l_i = 1, n_i = 0$), vapour ($l_i = 0, n_i = 0$) or nanoparticle ($l_i = 0, n_i = 1$) as shown in Figure 3.4. Solvent and nanoparticle cannot coexist in the same cell. Solvent occupies single cells while nanoparticles can occupy multiple cells. In this project, they span 3×3 cells. During the simulation, two possible moves can occur: a solvent step wherein a solvent cell transitions from one phase to another $l_i \rightarrow 1 - l_i$ or a nanoparticle step wherein a nanoparticle travels a single cell in one of the four directions along the array. Nanoparticles can only travel if all cells it is attempting to enter are filled with solvent. The displaced solvent is then regenerated in the wake of the travelling nanoparticle, simulating the solvent flowing around the nanoparticle. Either of these moves are accepted or rejected with a random Metropolis probability given by:

$$p_{acc} = min\big(1, e^{-\Delta E/k_B T}\big), \tag{47}$$

where ΔE is the change in energy caused by either the solvent phase change or the travelling nanoparticle and T is the temperature of the system. Every simulation step is composed of N solvent steps and M nanoparticle steps. N equals the number of cells, $x_{dim} \times y_{dim}$ while M equals the number of nanoparticles times the nanoparticle mobility (`NM`). We use `NM` $= 30$ in this work. The nanoparticle mobility controls how fast the nanoparticles wander with respect to the evaporation of the solvent. In contrast to original implementation where each solvent cell are visted once and nanoparticles are visited $N \times M$ times per simulation step, we allow randomness in the selection. Solvent cells and nanoparticles are randomly chosen one at a time.

To derive the equations for ΔE, we begin by looking at the Hamiltonian for the nearest neighbour only system as done in the original paper [45]:

$$E = -\epsilon_l \sum_{<ij>} l_i l_j - \epsilon_n \sum_{<ij>} n_i n_j - \epsilon_{nl} \sum_{<ij>} n_i l_j - \mu \sum_i l_i, \quad (48)$$

where ij corresponds to a nearest neighbour bond. The interactions energies are chosen as in original paper, and are defined in relation to ϵ_l. We set $\epsilon_n = 2\epsilon_l$ and $\epsilon_{nl} = 2.5\epsilon_l$. μ and $k_B T$ are also defined in relation to ϵ_l, but can vary over the course of the simulation. With all terms defined in relation to ϵ_l, we can see that the ϵ_l will cancel out in the exponent of equation 47, effectively setting ϵ_l to 1 and simplifying calculations. For this purpose, ϵ_l will be omitted from parameter values in the result section.

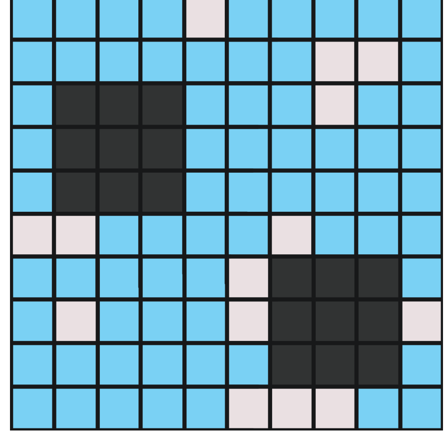


Figure 3.4: Lattice gas model, solvent is shown in blue while gas in white. 2 nanoparticles are shown in black. The lower nanoparticle is stuck, and cannot move as there is gas adjacent to it in all directions.

To be able to compare the nearest neighbour only case with the addition of next-nearest neighbours, scaling is added in as in [46]. Nearest neighbours are given a weight of 1 while next-nearest neighbours are given a weight of $1/\sqrt{2}$. Finally, a normalization factor of $1/(1/\sqrt{2}+1)$ then multiplies the interaction energies. With this normalization, the total energies are comparable.

Adding the normalization factor and next-nearest neighbours to equation 48 gives an expanded Hamiltonian:

$$\begin{aligned} E = \frac{1}{1+1/\sqrt{2}} \Bigg[& -\epsilon_l \Big(\sum_{<ij>} l_i l_j + \frac{1}{\sqrt{2}} \sum_{<ik>} l_i l_k \Big) \\ & -\epsilon_n \Big(\sum_{<ij>} n_i n_j + \frac{1}{\sqrt{2}} \sum_{<ik>} n_i n_k \Big) \\ & -\epsilon_{nl} \Big(\sum_{<ij>} l_i n_j + \frac{1}{\sqrt{2}} \sum_{<ik>} l_i n_k \Big) \Bigg] - \mu \sum_i l_i, \end{aligned} \quad (49)$$

where ij corresponds to the same as in equation 48 and ik corresponds to the next-nearest neighbour bonds. We can see that if the next-nearest neighbours were the same as the nearest neighbours, $l_k = l_j, n_k = n_j$, the two sums in each term would be equal and the normalization factor would then cancel, giving back equation 48.

The energy of a single lattice cell i and all its possible bonds is:

$$E_i = \frac{1}{1+1/\sqrt{2}}\Bigg[-\epsilon_l\Big(l_i \sum_j l_j + \frac{1}{\sqrt{2}} l_i \sum_k l_k \Big) - \epsilon_n \Big(n_i \sum_j n_j + \frac{1}{\sqrt{2}} n_i \sum_k n_k \Big) \\ - \epsilon_{nl}\Big(l_i \sum_j n_j + n_i \sum_j l_j + \frac{1}{\sqrt{2}} \Big(l_i \sum_k n_k + n_i \sum_k l_k \Big)\Big)\Bigg] - \mu l_i, \tag{50}$$

where the ϵ_{nl} term now has two sets of terms to account for when there is a nanoparticle and no solvent ($n = 1, l = 0$) or the opposite ($n = 0, l = 1$). j corresponds to the nearest neighbour and k the next-nearest.

For the purpose of running the simulation, we are not interested in the total energy of the system or that of a single cell, but rather *the change in energy* caused by either a phase change of a solvent cell or the movement of a nanoparticle, which involves multiple cells. For this, we need the energy difference ΔE for both a solvent phase change and a nanoparticle movement.

First we derive the ΔE for a solvent phase change. For a phase change to occur, the cell necessarily needs to have no nanoparticle ($n_i = 0$). Any term in equation 50 with n_i is therefore zero. The equation reduces to:

$$E_i = l_i \Bigg(\frac{1}{1+1/\sqrt{2}}\Bigg[-\epsilon_l\Big(\sum_j l_j + \frac{1}{\sqrt{2}} \sum_k l_k \Big) - \epsilon_{nl}\Big(\sum_j n_j + \frac{1}{\sqrt{2}} \sum_k n_k \Big)\Bigg] - \mu \Bigg). \tag{51}$$

The change in energy $\Delta E = E_f - E_i$ for a solvent phase change $l_i \to l_f$ is:

$$\begin{aligned} \Delta E &= l_f \Bigg(\frac{1}{1+1/\sqrt{2}}\Bigg[-\epsilon_l\Big(\sum_j l_j + \frac{1}{\sqrt{2}} \sum_k l_k \Big) - \epsilon_{nl}\Big(\sum_j n_j + \frac{1}{\sqrt{2}} \sum_k n_k \Big)\Bigg] - \mu \Bigg) \\ &\quad - l_i \Bigg(\frac{1}{1+1/\sqrt{2}}\Bigg[-\epsilon_l\Big(\sum_j l_j + \frac{1}{\sqrt{2}} \sum_k l_k \Big) - \epsilon_{nl}\Big(\sum_j n_j + \frac{1}{\sqrt{2}} \sum_k n_k \Big)\Bigg] - \mu \Bigg) \\ &= \Big(l_f - l_i \Big) \Bigg(\frac{1}{1+1/\sqrt{2}}\Bigg[-\epsilon_l\Big(\sum_j l_j + \frac{1}{\sqrt{2}} \sum_k l_k \Big) - \epsilon_{nl}\Big(\sum_j n_j + \frac{1}{\sqrt{2}} \sum_k n_k \Big)\Bigg] - \mu \Bigg). \end{aligned}$$

We can set $l_f = 1 - l_i$ to represent the phase change. The final equation is:

$$\Delta E = \Big(1 - 2l_i \Big) \Bigg(\frac{1}{1+1/\sqrt{2}}\Bigg[-\epsilon_l\Big(\sum_j l_j + \frac{1}{\sqrt{2}} \sum_k l_k \Big) - \epsilon_{nl}\Big(\sum_j n_j + \frac{1}{\sqrt{2}} \sum_k n_k \Big)\Bigg] - \mu \Bigg). \tag{52}$$

Equation 52 involves 8 bonds corresponding to the 8 neighbouring cells.

For the movement of a nanoparticle we consider equation 50 again. Since the solvent is regenerated in the wake of the moving nanoparticle, the lost energy from the μl_i term of the cell the nanoparticle enters will be offset by the same gain in the cell the nanoparticle leaves. This term can thus be omitted from the ΔE calculation.

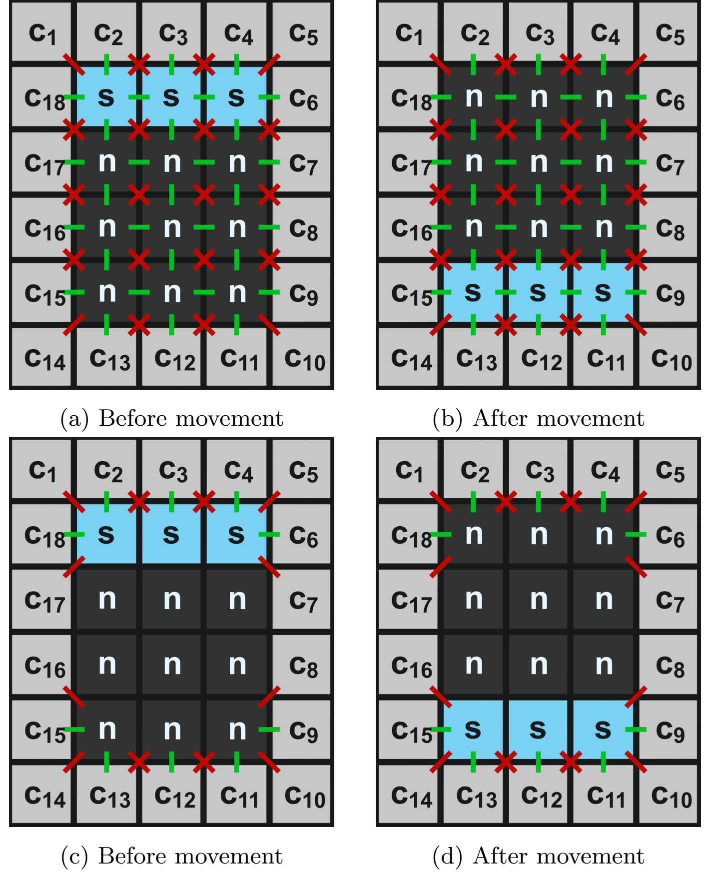
The energy difference for a nanoparticle movement therefore is:

$$\Delta E = \frac{1}{1+1/\sqrt{2}}\Bigg(\Bigg[-\epsilon_l\Big(l_f\sum_j l_j + \frac{1}{\sqrt{2}}l_f\sum_k l_k\Big) - \epsilon_n\Big(n_f\sum_j n_j + \frac{1}{\sqrt{2}}n_f\sum_k n_k\Big)$$
$$-\epsilon_{nl}\Big(l_f\sum_j n_j + n_f\sum_j l_j + \frac{1}{\sqrt{2}}\big(l_f\sum_k n_k + n_f\sum_k l_k\big)\Big)\Bigg]$$
$$-\Bigg[-\epsilon_l\Big(l_i\sum_j l_j + \frac{1}{\sqrt{2}}l_i\sum_k l_k\Big) - \epsilon_n\Big(n_i\sum_j n_j + \frac{1}{\sqrt{2}}n_i\sum_k n_k\Big)$$
$$-\epsilon_{nl}\Big(l_i\sum_j n_j + n_i\sum_j l_j + \frac{1}{\sqrt{2}}\big(l_i\sum_k n_k + n_i\sum_k l_k\big)\Big)\Bigg]\Bigg).$$

Grouping terms yields:

$$\Delta E = \frac{1}{1+1/\sqrt{2}}\Bigg[-(l_f - l_i)\Big(\epsilon_l\big(\sum_j l_j + \frac{1}{\sqrt{2}}\sum_k l_k\big) + \epsilon_{nl}\big(\sum_j n_j + \frac{1}{\sqrt{2}}\sum_k n_k\big)\Big)$$
$$-(n_f - n_i)\Big(\epsilon_n\big(\sum_j n_j + \frac{1}{\sqrt{2}}\sum_k n_k\big) + \epsilon_{nl}\big(\sum_j l_j + \frac{1}{\sqrt{2}}\sum_k l_k\big)\Big)\Bigg]. \tag{53}$$

To simplify this further, we examine what bonds do not change when a nanoparticle moves. The top plots of Figure 3.5 shows all bonds impacted by a 3×3 nanoparticle movement. The surrounding cells are labelled c_{1-18} as we do not know what are inside these cells. The cells which contain nanoparticles or solvent are labelled n and s respectively. These do not have a subscript as their contents are known. If for the before and after case, figures 3.5a and 3.5b, the same bond appears in both, we can ignore it in the energy difference calculation as it does not contribute to a change in energy. The key here is the fact that all nanoparticle cells and solvent cells are equivalent. For example, the $c_1 - s$ bond becomes a $c_1 - n$ bond, therefore we cannot ignore it. On the other hand, $c_{17} - n$ stays the same and can thus be ignored.



(a) Before movement

(b) After movement

(c) Before movement

(d) After movement

Figure 3.5: Nearest bonds (green) and next-nearest bonds (red) impacted by a 3×3 nanoparticle movement. The two top images show all bonds while the bottom two show needed bonds; those that cannot be ignored.

The bottom plots of Figure 3.5 shows all bonds that cannot be ignored. The initial 67 total bonds is reduced to only 26. The bonds at the solvent-nanoparticle interface are moved to the other side of the particle. Equation 53 then only needs to be calculated across the 3 cells the nanoparticle moves into and the 3 the nanoparticle moves from in the case of a 3×3 nanoparticle. Since we know what l_i, l_f, n_i, n_f are, we can write equation 53 for both cases.

For the cells which the nanoparticle will move into - the solvent cells in Figure 3.5c - we have $l_i = 1, l_f = 0, n_i = 0, n_f = 1$. Equation 53 becomes:

$$\Delta E = \frac{1}{1+1/\sqrt{2}}\Bigg[\Big(\epsilon_l\big(\sum_j l_j + \frac{1}{\sqrt{2}}\sum_k l_k\big) + \epsilon_{nl}\big(\sum_j n_j + \frac{1}{\sqrt{2}}\sum_k n_k\big)\Big) - \Big(\epsilon_n\big(\sum_j n_j + \frac{1}{\sqrt{2}}\sum_k n_k\big) + \epsilon_{nl}\big(\sum_j l_j + \frac{1}{\sqrt{2}}\sum_k l_k\big)\Big)\Bigg]. \tag{54}$$

Meanwhile, the equation for the cells in the wake of the nanoparticle with $l_i = 0, l_f = 1, n_i = 1, n_f = 0$ becomes:

$$\Delta E = \frac{1}{1+1/\sqrt{2}}\Bigg[-\Big(\epsilon_l\big(\sum_j l_j + \frac{1}{\sqrt{2}}\sum_k l_k\big) + \epsilon_{nl}\big(\sum_j n_j + \frac{1}{\sqrt{2}}\sum_k n_k\big)\Big) \Big(\epsilon_n\big(\sum_j n_j + \frac{1}{\sqrt{2}}\sum_k n_k\big) + \epsilon_{nl}\big(\sum_j l_j + \frac{1}{\sqrt{2}}\sum_k l_k\big)\Big)\Bigg], \tag{55}$$

where in both these equations j corresponds to the needed nearest neighbours and k to the next-nearest needed neighbours as shown in Figure 3.5. Adding both of these equations yields the energy difference for the nanoparticle movement. Omitting these unneeded bonds has the benefit of speeding up the algorithm as time is not wasted calculating terms that will simply cancel each other.

Implementation - The growth algorithm was implemented in Python following the general algorithm detailed in [45,46]. This algorithm simulates how the nanoparticles move as the solvent evaporates. This is done with two actions: the evaporation of a solvent cell and the displacement of a nanoparticle. Both of these actions are accepted with some probability based on the change in energy. Repeated applications of these two actions results in a simulation of how the system evolves. The accuracy of the simulation was roughly determined by visually comparing the results in [46] with the implemented version as shown in the result section. The goal was to have an implementation that produces the same variety in structures under a set of growth parameters.

Implementing an algorithm which is computationally expensive in Python does pose the question of optimization. By the nature of the algorithm being heavily sequential, a naive implementation in Python would be very slow. This also means that vectorizing the code would not be very efficient. To ensure that the algorithm would be fast enough to be used in the growth pipeline, Numba [47] was used to speed up computation. Numba is a compiler for Python that allows all, or parts of the code to bypass the python interpreter and instead be compiled to machine code. This means that once the code has been run once, it will run at native machine code speeds in subsequent calls. Using Numba for the growth simulation provides a roughly $60x$ speedup as compared to the python implementation. This optimization was deemed sufficient for the purpose of this project.

3.2.3 Machine learning

Algorithm 2 Population genetic algorithm.

1: **for all** networks **do**
2: Initialize parameters θ_i
3: **end for**
4: **for** N epochs **do**
5: **for all** networks **do**
6: Produce growth using network policy
7: Score growth
8: **end for**
9: Sort network based on score
10: Delete N worst networks
11: Duplicate top networks to replace deleted networks
12: **for all** networks **do**
13: Mutate parameters $\theta_i = \theta_i + \mathcal{N}(0, 0.01)$
14: **end for**
15: **end for**

Algorithm 3 MC pipeline.

1: Initialize all parameters θ_i
2: **for** N epochs **do**
3: **for all** i **do**
4: $\theta_i \rightarrow \theta_i + \epsilon_i$
5: **end for**
6: Generate M random seeds
7: **for** M growths **do**
8: Produce growth using network policy and random seed
9: Score growth using FDTD simulator
10: **end for**
11: Calculate mean score S
12: **if** S less than previous **then**
13: Accept ϵ
14: **else**
15: Revert ϵ
16: **end if**
17: **end for**

For the policy generation portion of the pipeline, a FCNN was used. This network has the role of altering one or more growth parameters, such as the temperature or chemical potential, during the simulation to control the final structure. There are many other growth parameters possible, such as the different interaction strengths between components, the ratio of nanoparticle displacements to solvent cell evoparations, or even the number of nanoparticles. However as we wanted this pipeline to be semi-realistic and potentially replicable in real life, we constrained the input and output of the network to what could be logically be controlled in real life. For the input, the current simulation time is the simplest option as that directly links to the real growth time. For the output, the temperature and chemical potential are growth parameters that can quite easily be modified during a real experiment and are thus good candidates.

Training of the network(s) was done in two ways. Initially, a population based genetic algorithm was used. This involves initializing N networks, where each network then grows a structure following their generated policy. After each growth is finished and scored, the networks are sorted based on their

growth's score. How the score is calculated depends on the objective, the first example presented in section 3.3.2 uses the size of clusters of nanoparticles, while the full pipeline uses a score calculated from the transmission of simulated light through the growth. The top performing networks are then kept while the rest is deleted. To fill these empty spots, the top performing networks are cloned. Finally, each network is mutated. For this project, a random Gaussian number $\mathcal{N}(0, 0.01)$ with zero mean and $\sigma = 0.01$ is generated for each network parameter. The process is then repeated with the new population. This algorithm is described in Algorithm 2.

The second method is similar to the MC optimizer. Instead of a population of networks, a single network is used. The policy created by this network is used to produce N growths, which are then scored. The mean score is then used for the purpose of accepting or rejecting mutations. This algorithm is shown in Algorithm 3.

Implementation

3.2.4 FDTD simulation

As the main objective of this demonstration of the growth pipeline is the design of photonic chips, a method to simulate light propagating through the grown structures is needed. This simulation is then used to calculate the transmission of light as a function of wavelength after light has travelled through the chip. This transmission spectrum can then be used to score how well the growth acts as a photonic chip to accomplish some goal, such as a wavelength splitter or bandpass filter. Finite Difference Time Domain (FDTD) [48] is a popular algorithm used in research for this purpose. In this project, an optimized implementation is used. Two different simulation packages were tested: Lumerical [49] and the python implementation of Meep [50]. Lumerical is used in a simple test case while Meep was used for the rest of the project.

Here I will describe the basics of how FDTD functions as per [48], in the context of simulating light. While I did not implement the algorithm, it is still important to understand fundamentally how the algorithm works, and in particular, the limitations that need to be accounted for.

FDTD - The goal of FDTD is to simulate how light will propagate in some medium. This is done by numerically solving Maxwell's curl equations:

$$\frac{\partial H}{\partial t} = -\frac{1}{\mu}\nabla \times E, \tag{56}$$

$$\frac{\partial E}{\partial t} = \frac{1}{\epsilon}\nabla \times H, \tag{57}$$

where E is the electric field and $H = \frac{1}{\mu}B$ is the magnetic field. To propagate these equations in time, the derivatives are discretized using the central difference. We start by looking at central difference equation:

$$\left.\frac{\partial f(x)}{\partial x}\right|_{x=x_0} \approx \frac{\Delta f(x)}{\Delta x} = \frac{f(x_0 + \frac{\Delta x}{2}) - f(x_0 - \frac{\Delta x}{2})}{\Delta x}. \tag{58}$$

This approximation has second order precision at the point $x = x_0$. This precision can be seen when doing the subtraction of the two terms using their Taylor series expansion at the point x_0. The even powers will cancel, leaving behind the odd powers. Dividing δ to solve for the derivative yields additional terms of order $O(\delta^2)$ and above. In the limit of $\delta \to 0$, this equation becomes exact.

In our case, the variable x will be either a spatial coordinate or time. For simplicity case, we will consider the 1D case for the derivation of the update equations. Having the light wave propagating in the x-direction with the electric field polarized in the z-direction implies that the magnetic field is polarized in the y-direction. Equations 56 and 57 then becomes:

$$\frac{\partial H_y}{\partial t} = \frac{1}{\mu}\frac{\partial E_z}{\partial x}, \tag{59}$$

$$\frac{\partial E_z}{\partial t} = \frac{1}{\epsilon}\frac{\partial H_y}{\partial x}. \tag{60}$$

The partial derivatives in these equations are then converted to the central difference shown in Equation 58. As the central difference is accurate to second order at the point between the two offsets, we want the point where we write the central difference to be located between two spatial points as well as two time points. This is shown in Figure 3.6. Starting with Equation 59, the partial differences are exchanged with the central differences around the point marked with an x:

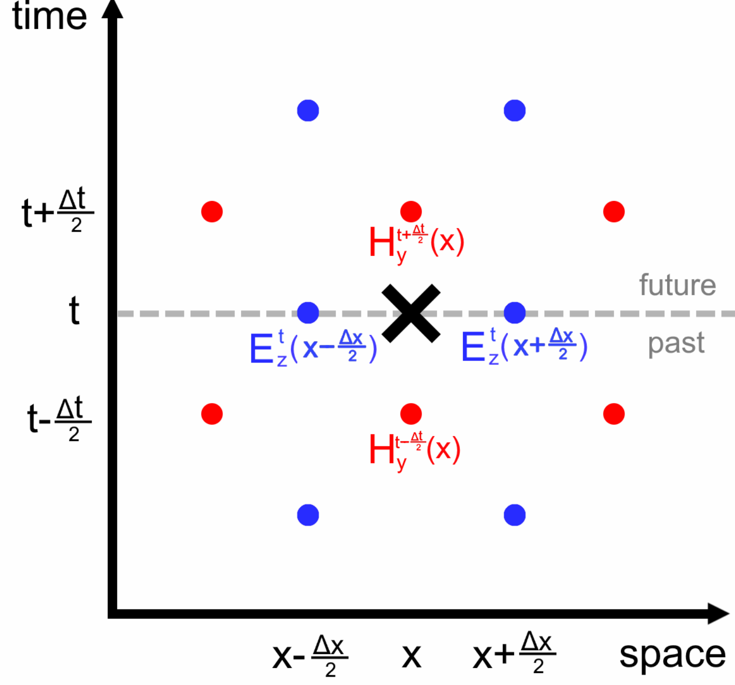


Figure 3.6: Central difference point for magnetic field propagation of Maxwell's equations in FDTD.

The offset lattice as shown in Figure 3.6 is known as the Yee lattice. By offsetting the magnetic field lattice by half in both time and space, the central difference can be written for the center points to maintain second order accuracy. In addition, this configuration allows alternating the field update equations to step forward in time. Replacing the partial derivations in equation 59 with the corresponding central difference at the marked point gives:

$$\frac{H_y^{t+\frac{\Delta t}{2}}(x) - H_y^{t-\frac{\Delta t}{2}}(x)}{\Delta t} = \frac{1}{\mu}\frac{E_z^t(x+\frac{\Delta x}{2}) - E_z^t(x-\frac{\Delta x}{2})}{\Delta x}. \tag{61}$$

Solving for the future time gives:

$$H_y^{t+\frac{\Delta t}{2}}(x) = H_y^{t-\frac{\Delta t}{2}}(x) + \frac{\Delta t}{\mu \Delta x}\left(E_z^t(x+\frac{\Delta x}{2}) - E_z^t(x-\frac{\Delta x}{2})\right). \tag{62}$$

This equation is the update rule for the magnetic field. The magnetic field at the future time $t+\frac{\Delta t}{2}$ is dependent on the magnetic field at that point in space a full time increment in the past $t-\frac{\Delta t}{2}$, as well as the electric field a half spatial step on either side at half a time increment in the past. Once this process has been done for all magnetic field lattice cells, the simulation has moved half a time increment in the future. Once the magnetic field has been stepped forward, the same is done for the electric field at the new time. Doing the same process with Equation 60 gives the update equation:

$$E_z^{t+\frac{\Delta t}{2}}(x) = E_z^{t-\frac{\Delta t}{2}}(x) + \frac{\Delta t}{\epsilon \Delta x}\left(H_y^t(x+\frac{\Delta x}{2}) - H_y^t(x-\frac{\Delta x}{2})\right). \tag{63}$$

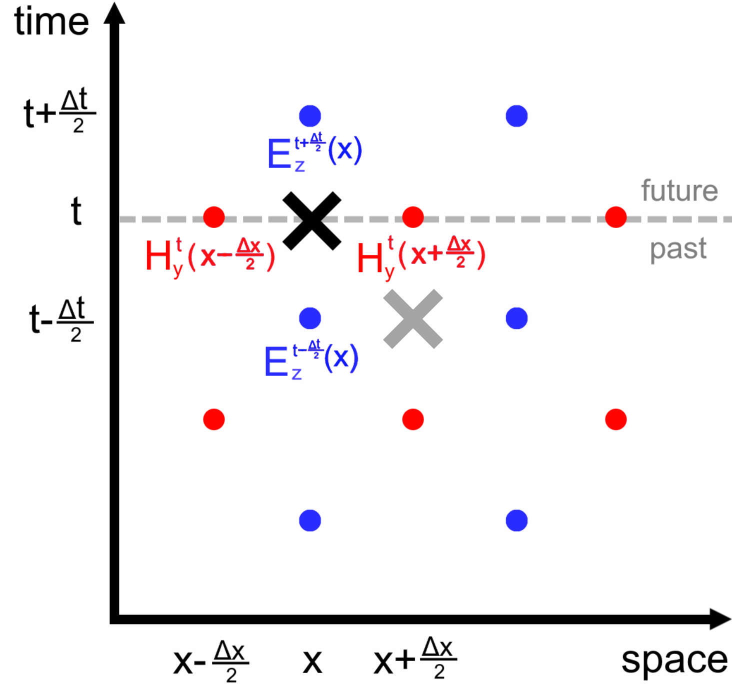


Figure 3.7: Central difference point for electric field propagation of Maxwell's equations in FDTD.

As seen in Figure 3.7, this update equation corresponds to a point half a time step in the future of the previous one and offset by half a space unit, as shown by the previous spot marked with a grey x. Similar to the magnetic field, the future electric field depends on the previous electric field value as well as the two magnetic field points half a space unit away and half a time increment in the past. Once this update rule has been applied to all future electric field cells, the simulation has moved a full time unit forward taking into account the magnetic field update. The process then repeats with the magnetic field update equation in an infinite cycle.

This alternating process bears a strong resemblance to the physical process in play, as the electric field drives the magnetic field and vice versa in light.

Implementation - Using the FDTD packages involves calling the defined methods to construct a simulation environment. For our purposes, we need a way of converting the grown structure into a format that can be used to describe the material light is propagating through. After, it is a simple task

of properly specifying the simulation parameters:

1. Light source
2. Boundary conditions
3. Simulation dimensions
4. Spatial resolution
5. Total simulation time
6. Other necessary monitors

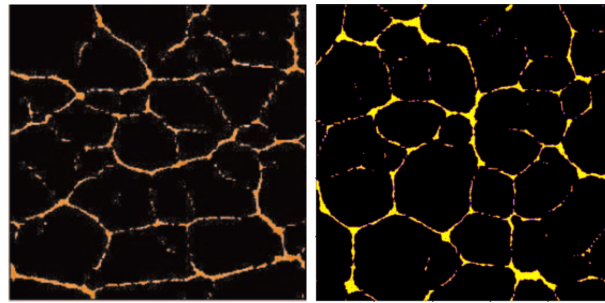
In general, a Gaussian light pulse is used as a light source, as that allows the calculation of transmission across a wide spectra of wavelengths with a single simulation. The width of the light source is set to extend past the waveguide by a small amount, to ensure the waveguide is properly filled with light. For the boundary conditions, Perfectly Matched Layers (PMLs) [51] are used to fully absorb without reflection all outgoing light. The dimensions, resolution and simulation time are selected to ensure accurate results. Finally, monitors are specified to measure the light at the output of the growth.

3.3 Results

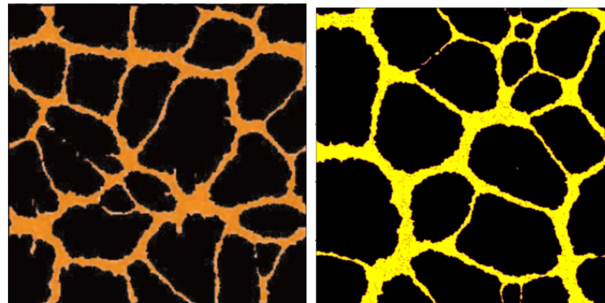
The results are presented in a semi-chronological order. First the implementation of the growth algorithm is tested in section 3.3.1, followed by a simplified pipeline test in 3.3.2. The link between the FDTD simulation and the growth algorithm is then established in sections 3.3.3 and 3.3.6. The main results for the full pipeline with 3D FDTD simulations follows in 3.3.7. Finally, some experimentations with alternate scoring objectives are shown in 3.3.8.

3.3.1 Growth simulation comparison

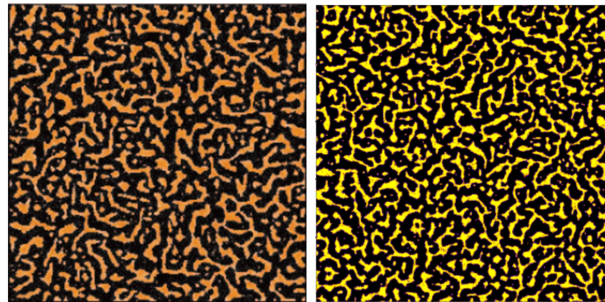
To begin, the accuracy of the growth simulation I implemented is compared to the ones from [45,46]. To do this, growths are simulated using the same growth parameters as done in the papers as shown in Figure 3.8. These are then visually compared to ensure the implementation is accurate. More comprehensive tests to determine an exact match is not needed as the accuracy of the simulation is not the primary focus, but rather having a functional growth model.



(a) $k_BT = 0.2$, $Frac = 0.05$

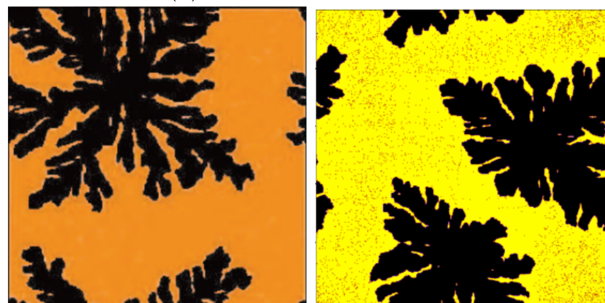


(b) $k_BT = 0.2$, $Frac = 0.2$



(c) $k_BT = 0.4$, $Frac = 0.3$

(d) $k_BT = 0.4$, $Frac = 0.6$



(e) $k_BT = 0.2$, $Frac = 0.6$

Figure 3.8: Replicated growths (right) using the same growth parameter as for the reference growths (left).

Figure 3.8 shows a comparison between the growth algorithm as implemented in [46] and the replicated algorithm for this project. The reference growths are on the left while the replicated growths using the same growth parameters are on the right. k_BT corresponds to the temperature of the simulation, while $Frac$ corresponds to the fraction of space occupied by nanoparticles. All simulations had dimensions 1024×1024 and were run for 1000 simulation steps. The nanoparticle mobility `NM` was 30, which means that each simulation steps included 1 solvent cycle and 30 nanoparticle cycles. As can be seen in Figure 3.8, the replicated results are near-identical to the original. For the bottom image, the replicated results had nucleation begin in two places as opposed to the single in the original. This lead to the two branching air pockets competing for space and therefore the squished appearance. This Figure therefore shows that the implemented algorithm is in the necessary state for this project.

3.3.2 Mean cluster size target pipeline

With a working growth simulation, the next step is to do a simple pipeline test using a simple objective. For this test, the objective is a target mean cluster size. The score is calculated as:

$$Score = -abs(targetSize - meanSize), \tag{64}$$

which means the networks seek to minimize this score. To do so, the networks are able to control the temperature k_BT during the growth. Every 50 growth simulation steps, the current time step is used as input to the networks, which output what k_BT should be changed to. This process repeats for 1000 simulation steps in total. The output of the network over all time steps represents the policy of the network; how the temperature should change over the duration of the growth. This can be viewed as the recipe used by the growth simulation to produce the grown structures.

A population based genetic algorithm is used to train the networks. In total, 25 networks are used. Once all growths are finished, the networks are scored based on their output temperature schedules. The 4 top scoring networks are kept, while the rest are deleted. The kept networks are then cloned and mutated following algorithm 2. A random perturbation is generated from a Gaussian with zero mean and $\sigma = 0.01$ for each parameter in the network. This new set of networks are then used for the next epoch. For this test 50 epochs were done in this way.

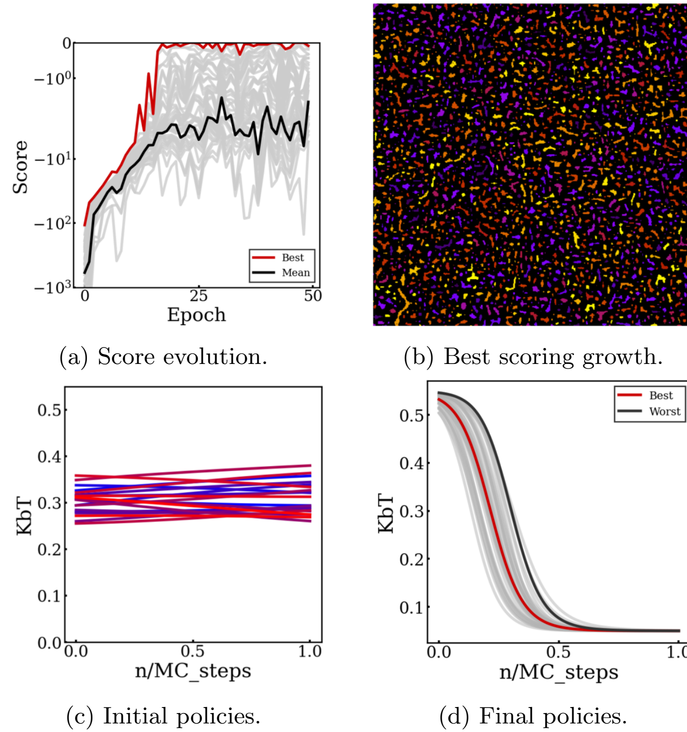
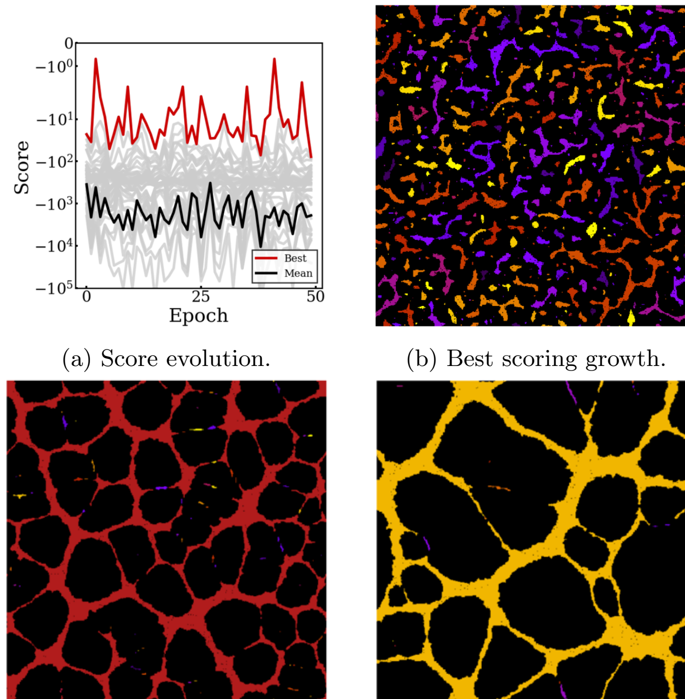


Figure 3.9: Mean cluster size test results for growth pipeline with target cluster size of 100.

The results of this test is shown in Figure 3.9. From the score evolution shown in the top left, we can see how this population based approach improves. Within 10 epochs the best network is capable of producing growths with clusters of the desired size target size. The mean policy score slowly increases beyond this, signifying a slow improvement of the general member of the population. The evolution of the policies can be seen in the bottom two plots. From the initial policy produced by the starting network parameters, the population of networks settled on a decreasing temperature policy. This makes sense as freezing the growth is an effective way of stopping any further movement once a desired amount of clustering has occurred. The best growth is shown in the top right, where clusters are coloured randomly for visualization purposes.

This example demonstrates how the pipeline can learn policies to direct growth. While this is a simple example, the same method is applicable for more complicated objectives.



(a) Score evolution.

(b) Best scoring growth.

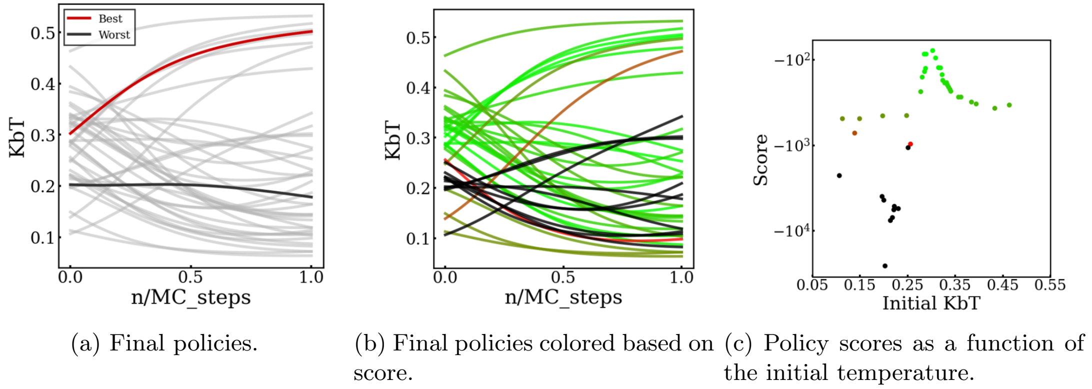
Figure 3.10: Mean cluster size test results for growth pipeline with target cluster size of 500. Bottom plots show two examples of an undesirable growth with good score.

A somewhat unexpected result occurs when the target cluster size is increased to 500. As opposed to the successful policy generation for the smaller target cluster in Figure 3.9, Figure 3.10 show how score does not improve during training.

The first observation is that the pipeline is able to generate a policy that seemingly has a very good score in the first few epochs. However, the best score drastically lowers the following epochs. This is quite strange as we would expect a policy to produce similar growths, and therefore have a similar score. After investigation, we determined that the culprit was the variance in the score for a given policy. As can be seen in the bottom two plots, many of the policies ended up growing a very large cluster with a few smaller ones scattered around. Since we are interested in the mean cluster size, these policies could, by chance, score extremely well. This introduces friction in the learning process, as these unwanted policies are kept and other more consistent policies are removed.

The variance of the score for a policy will be important in the more complicated photonic chip objective. Therefore, it is worth briefly talking about why variance is a significant problem. Primarily, the issue with a given policy having a large variance in the score is that a single growth from that policy will likely not accurately represent the general behaviour of the policy. This means that it is nigh impossible to compare two policies to determine which is better, rendering training unstable as seen with this test. While difficult, it is not impossible to train, as can be seen in the top right plot which shows a growth that can be considered closer to what we desire for a policy.

One possibility to improve training is to change the score to one with a smaller variance. For example, instead of having a score based on the mean cluster size, use a score that seeks to maximize the number of clusters within a certain size. While possible for this test, this option is not always available and, sometimes, the best we can hope for is to reduce the variance of the score. This can be done by averaging the score of multiple independent growths produced with the same policy. Unfortunately, for a population based algorithm as used here, this would need significant more computation time.



(a) Final policies.

(b) Final policies colored based on score.

(c) Policy scores as a function of the initial temperature.

Figure 3.11: Policies of the mean cluster size 500 test. Green is best score while black is worst.

Figure 3.11 provides some further insight into the results. The top plot shows the policies for the final epoch. The bottom left plot has the policies coloured based on their corresponding score, where green is a better score. When these policies are plotted based solely on the initial temperature (bottom right), it can be seen that there is a heavy correlation between initial temperature and final score. This implies that the initial temperature plays a very important part in determining the score. The later time steps are mostly irrelevant for the growth. This, combined with the starting epoch already having a score close to the target renders further learning difficult.

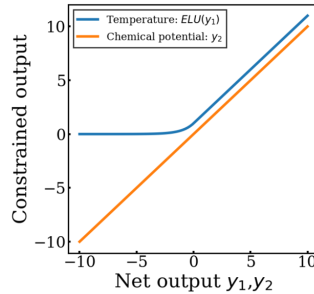


Figure 3.12: Constrained network output for temperature and chemical potential.

To give more control to the network over the growth, the chemical potential μ in the Hamiltonian shown in equation 48 is added as a second output. The network can now vary both the temperature as well as the chemical potential of the growth simulation. Figure 3.12 shows the possible output of the network. To prevent temperatures from reaching non-physical negative values, the temperature output

is constrained to be positive by applying a final *Elu* function [52]. Examples of possible policies for different number of consecutive mutations are shown in Figure 3.13. From this, the breath of possible policies can be seen as it would be trivial to add even more controllable parameters.

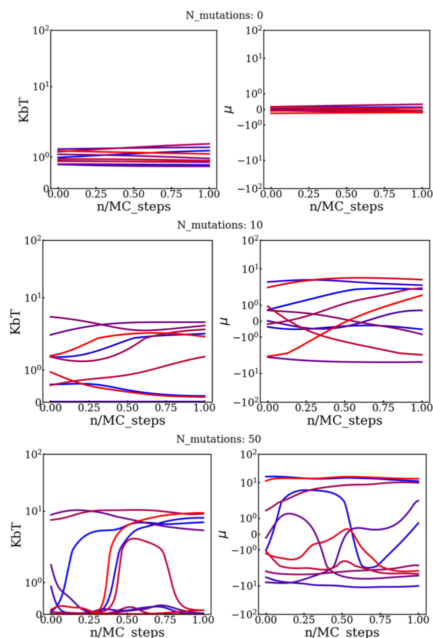
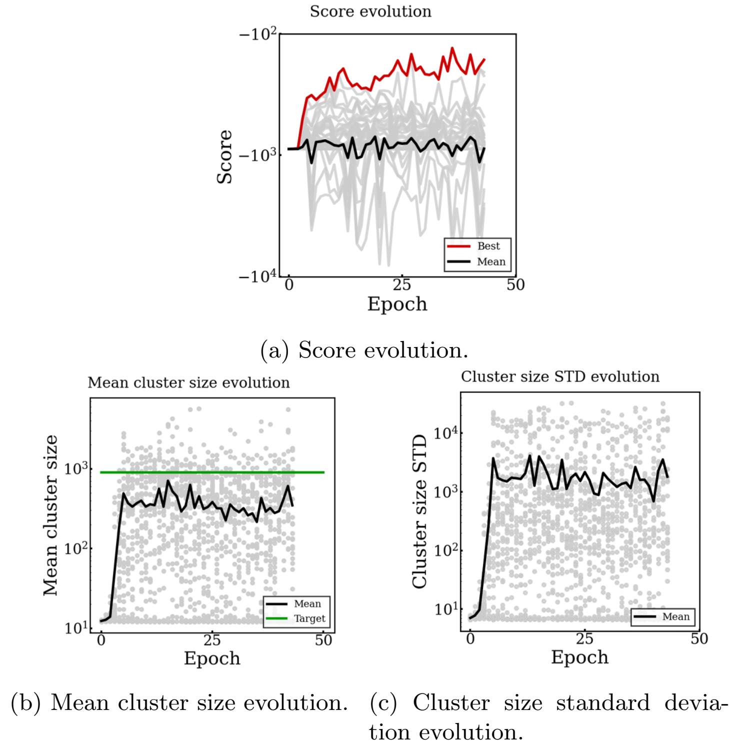


Figure 3.13: Sample of network policies after N accepted mutations. Top: initialm middle: 10, bottom: 50. Left shows the temperature while right shows μ.

With the addition of μ, we can take a closer look at how the policies might evolve. Figure 3.13 shows the output policy of 10 networks after having experienced 0, 10, and, 50 mutations. This Figure gives an idea as to how complicated the policies might become. In only 10 epochs complicated functions already start appearing, while at 50 we can see very complicated functions. This is interesting as it gives an idea as to how long training needs to occur before the breadth of policies can make an appearance.



(a) Score evolution.

(b) Mean cluster size evolution.

(c) Cluster size standard deviation evolution.

Figure 3.14: Mean cluster test with penalty for cluster size standard deviation. Target cluster size of 900.

To try to help training, a penalty for the cluster size standard deviation can be added to the score:

$$Score = -abs(targetSize - meanSize) - 0.1 \times STD, \tag{65}$$

where STD is the standard deviation of the cluster size. This addition tries to mitigate the policies which have a very large variance in scores. Figure 3.14 shows the evolution of the score, mean cluster size and cluster size STD. The green line in the bottom left plot shows the target cluster size. One problem with this type of penalty is the balancing of how impactful to make it. If the penalty is too significant, the training will tend toward policies which generate minimal STD at the expense of the minimizing the mean cluster size difference. With the chosen penalty factor of 0.1 as shown in the score equation, we can see that the STD still reaches high values. However, the mean cluster size manages to approach the target cluster size, which is the main objective.

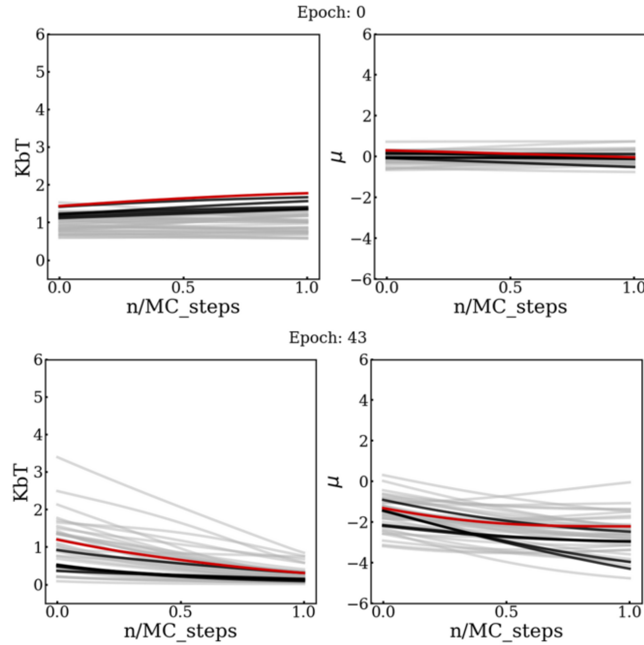


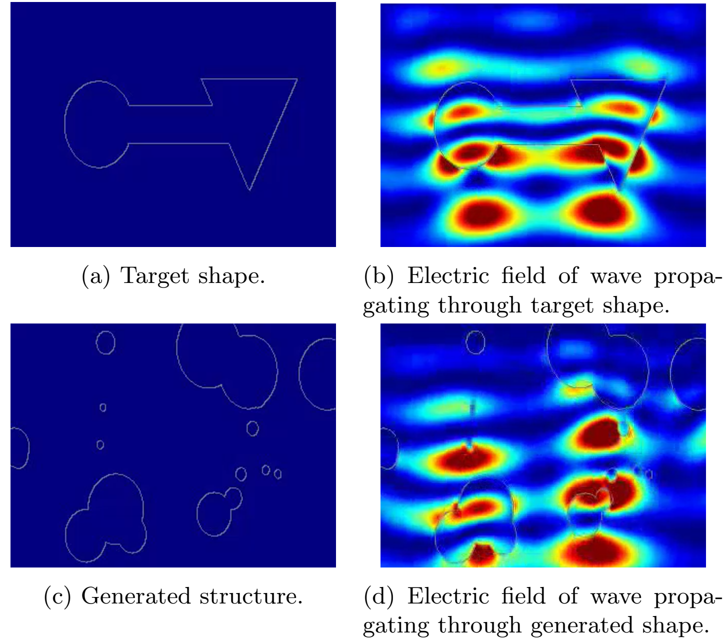
Figure 3.15: Policies for the standard deviation penalty test. Top plots shows the initial policies while the bottom plots show the final ones.

The policies are shown in Figure 3.15. The temperature tend towards a cooling method, freezing the growth after some time. The chemical potential evolve towards values near -2, which corresponds to the equilibrium point for the solvent phases. As μ decreases, evaporation occurs faster which contributes to the system freezing.

The above tests use a structural objective, the grown structure is directly used to score the policy. In this case, calculating statistics from the cluster size. An example of another structural objectives could be the shape of the clusters. For the FDTD pipeline, the objective is not directly tied to the finished structures, but rather the results of the FDTD simulation. This is an example of a performance objective, which are usually more difficult to learn as there is an additional layer between the grown structure and the corresponding score.

3.3.3 FDTD test

Before using light simulations in the full pipeline, the FDTD packages were implemented for a simpler test. This involved a simple inverse design problem of replicating the electric field after light had propagated through a target structure.



(a) Target shape.

(b) Electric field of wave propagating through target shape.

(c) Generated structure.

(d) Electric field of wave propagating through generated shape.

Figure 3.16: Lumerical target shape test. The electric field at the bottom of the generated structure matches that of the target structure.

The results for the first test is shown in Figure 3.16. Here, the algorithm has the objective of matching the electric field magnitude at the bottom boundary across all simulation time. In other words, the outgoing light after travelling through the structure should be the same for the target shape and generated structure. To do this, a basic MC algorithm is used. Every epoch, one of 3 different actions is chosen:

1. Add a circle with random radius
2. Move a circle a random distance in some direction
3. Remove a circle

These moves are accepted if it reduces the loss, defined as:

$$U = \frac{1}{T}\sum_{t=0}^{T}\frac{1}{N}\sum_{i=0}^{N}\left(E^t_{target_i} - E^t_i\right)^2 \tag{66}$$

Where N is the x dimension of the simulation array and T is the total number of simulation time steps. E corresponds to the magnitude of the electric field at the bottom of the simulation. This loss therefore is minimal when the electric field magnitude at the bottom of the simulation is equal to that of the target structure for all time steps. While this could have also been done easier by looking at the frequency domain of the outgoing light, this test was both useful for learning how to use the FDTD package and generating plots of the electric field.

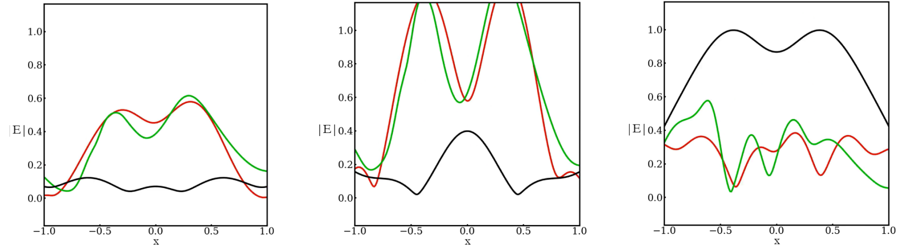


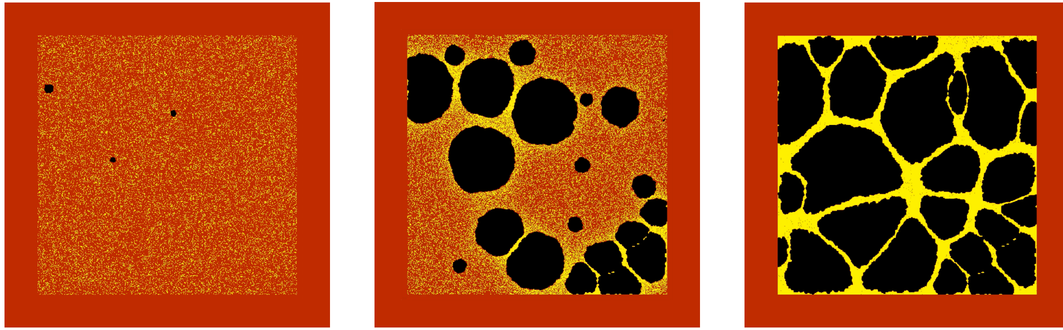
Figure 3.17: Snapshots of the electric field magnitude for the simulation with no structure (black), target structure (red), and generated structure (green).

The performance of the generated structures can be visualized by looking at the electric field magnitude as compared to the target structure and an empty simulation array. As can be seen in Figure 3.17, the electric field of the generated structure closely matches that of the target structure. This is particularly interesting as the generated structure does not match the target structure.

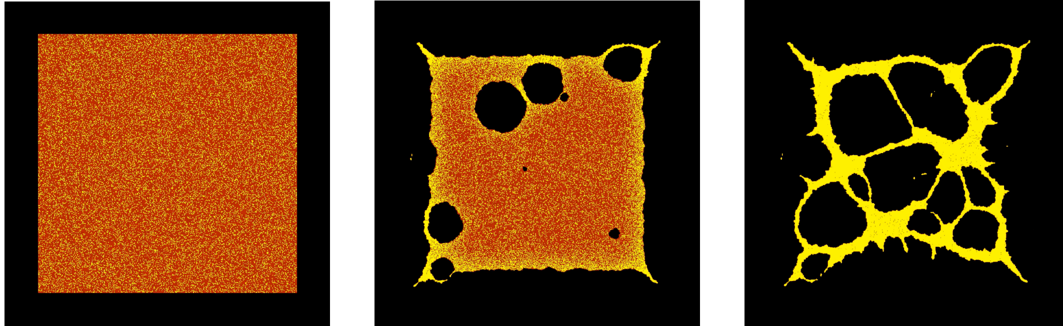
While this test was done using Lumerical [49], the rest of the project used the Python implementation of Meep [50] for the FDTD simulations.

3.3.4 Growth for FDTD simulations

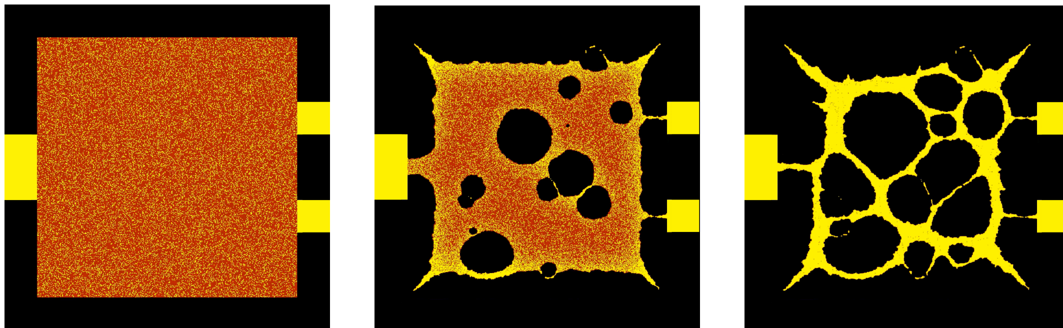
With a working FDTD method to score the growths, the final piece is to ensure the growths will be usable for the purpose of a photonic chip.



(a) Wet boundary; solvent surrounds the simulation area.



(b) Dry boundary; air surrounds the simulation area.



(c) Dry boundary with probes.

Figure 3.18: Non-periodic boundary conditions for growth model. Each has the initial (left), finished (right), and an in-between time (middle) shown. Nanoparticles (yellow) are suspended in the liquid solvent (red) and pushed away from the air (black).

Figure 3.18 shows different boundary conditions implemented for the growth simulation. The growth is shown for three different times during growth, with the most important being the finished growth at the right. The periodic boundary condition used in the previous examples is not suitable for the purposes of designing a photonic chip as it is unrealistic. Instead, non-periodic boundary conditions as shown in the Figure are much more akin to a real experiment. This simulates a fixed amount of nanoparticles that would be present during an experiment. The bottom two boundary conditions also simulate evaporation occurring from the borders in a similar way to how drops of liquid evaporate. The final change is the addition of probes as shown in the bottom plots. These corresponds to where the waveguides would be present, and would be present during the growth to ensure the nanoparticles form a connection.

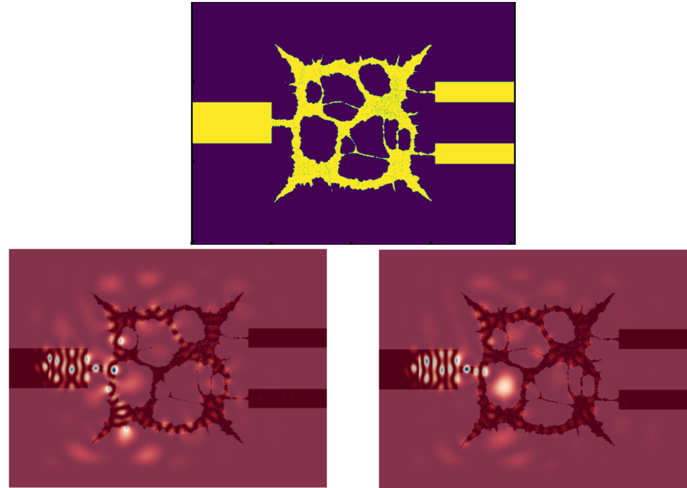


Figure 3.19: Example of grown structure acting as a photonic chip. Bottom plots show light propagating through the structure.

An example of how the probe connection boundary would be used to construct a photonic chip is shown in Figure 3.19. In this example, the boundaries have been padded to form the waveguides leading into and out of the grown chip. The light source is then placed at a far enough distance to simulate the steady state light wave travelling through the waveguide. The resolution of the simulation needs to be carefully chosen to ensure the finer structures present in the growth are simulated correctly. Resolution tests were often done, and some of them are shown in later results.

The configuration shown with a single input and two outputs could be used for objectives such as wavelength splitting. Different configurations are easy to implement, as it is a simple matter to modify the boundary probes used during growth.

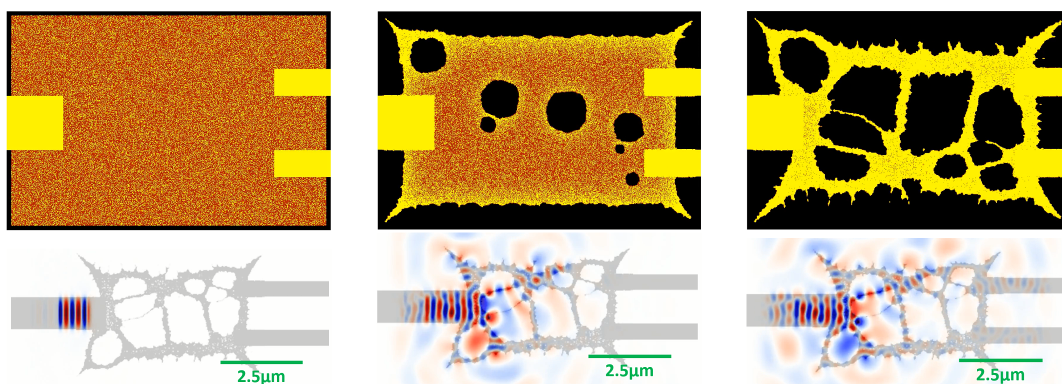


Figure 3.20: Example of growth used as a photonic chip. Growth region is extended past probes to ensure a wider connection.

One issue that arose with the probe only connecting at the boundary of the growth is the very thin connections that formed. A solution to this problem was to extend the growth region past the probes, guaranteeing that a strong connection is made. An example of a growth and the following light simulation

is shown in Figure 3.20. While this increased the time for the growth, the waveguides have much better integration into the growth.

As a realistic simulation is desired, the physical dimensions are also important. The lattice gas model from [45] has each cell of length 1nm. For the shown growth of dimension 1500×1000, we would have lengths 1.5×1.0µm. This size is within the range of real photonic chips. To directly match the simulation parameters in [1], we will manipulate the cell length to ensure the grown chip roughly corresponds to the chip sizes from the paper. One could view this as increasing the nanoparticle size in the original lattice gas model. The length increase is of the order $2x$ and therefore is still within the same regime.

For this test, the electric field is in the z-axis, while the growth spans x and y. The light would therefore correspond to the fundamental TM mode of the waveguide.

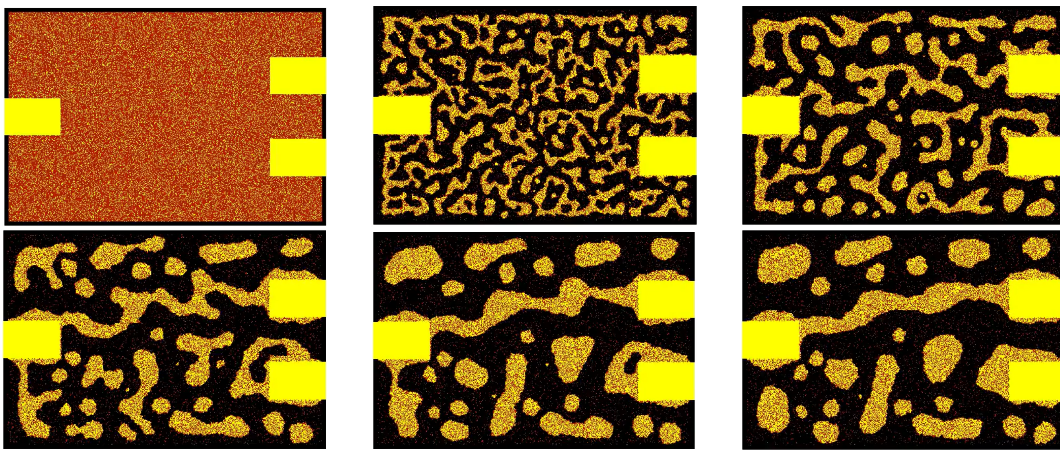
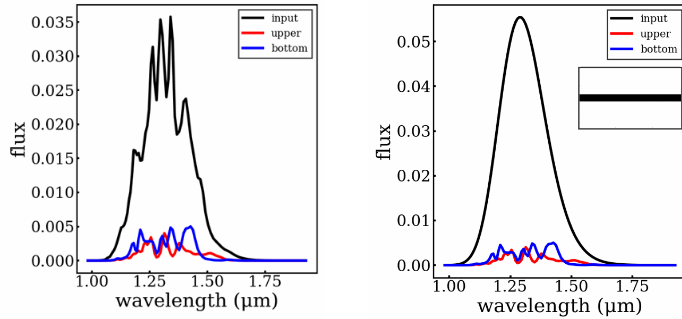


Figure 3.21: Growth with solvent in equilibrium. Liquid solvent is shown in red, nanoparticles in yellow and air in black.

For the purpose of the pipeline, it is helpful to minimize the time each epoch takes. A significant portion of the total time is taken by the growth simulation, which we would want to reduce. Unfortunately, due to the nature of the lattice gas model, there is no easy way of minimizing the growth time. In particular, as shown in Figure 3.21, it is possible for growths to enter an equilibrium state where both the liquid and vapour coexist. Because of this, the simulation cannot be stopped by checking if all the solvent has evaporated. For the pipeline we therefore gave a fixed number of simulation steps that allowed for growth to occur. The consequence of this is that growths which still includes solvent are allowed. For this project, we simply remove the solvent and use the nanoparticle structure left as the growth for the light simulations. A more realistic solution would have been to include an evaporation phase that occur after growth, but this would increase the time each epoch takes.



(a) Input flux for a growth.

(b) Input flux for a waveguide, as shown in the inset.

Figure 3.22: Input flux measurement comparison between a growth (left) and an empty waveguide (right). The blue and red lines correspond to the output flux for the growth's two output probes.

To be able to accurately compute the transmission of the light through the structure, some care has to be taken when measuring the input flux. The input flux here is defined as the flux through the waveguide as it enters the growth, shown in black in Figure 3.22. The output flux, shown in red and blue, is the flux at the output probes of the growth. The left plot shows what the input flux would be measured as if the flux is recorded during the growth light simulation. As can be seen, the shape of the input pulse is not what would be expected of a Gaussian wave. The cause of this is reflection that occurs in the structure, sending some of the light back out through the input. To account for this, a second simulation is run with only the input waveguide. As there is no reflection in this case, the measured pulse corresponds to the correct input wave. As all growths have the same input, this additional simulation only has to be run once for each source of light.

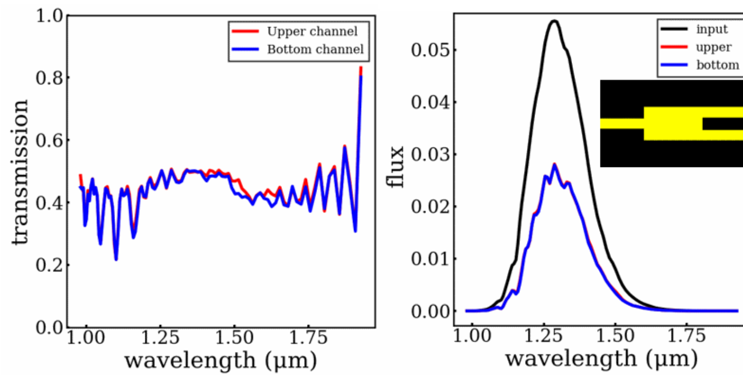


Figure 3.23: Resolution example for a filled box growth. Resolution is 40 $pixels$/µm

As mentioned previously, the resolution of the simulation is incredibly important to maintain a high accuracy. Here the resolution is defined as the number of pixels per µm. This is both true for the limits of FDTD, but also to ensure the small structures in the growths properly interact with the light. To test the effect of resolution, a simple test can be done. As shown in the inset of Figure 3.23, a filled box is used instead of a growth. As this configuration is now symmetric, we would expect the transmission

to be equal in both output probes. By varying the resolution of the simulation, we can see where this holds.

The right plot shows the flux through the different probes, for this resolution the flux seems equal for both outputs. The transmission shown in the left plot shows minor differences.

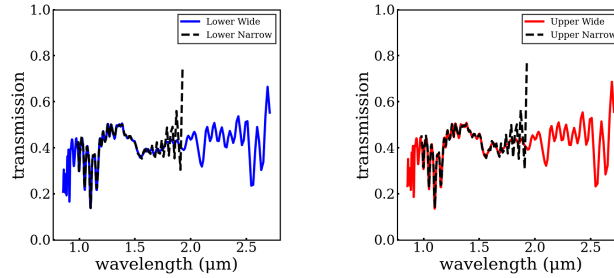
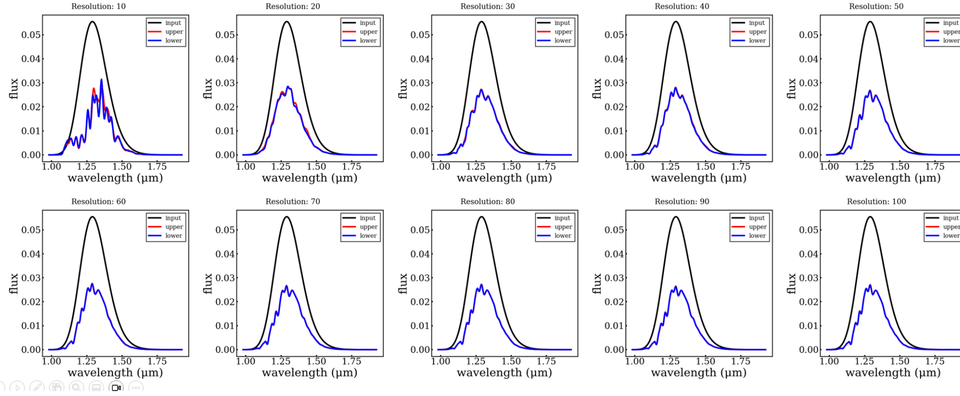
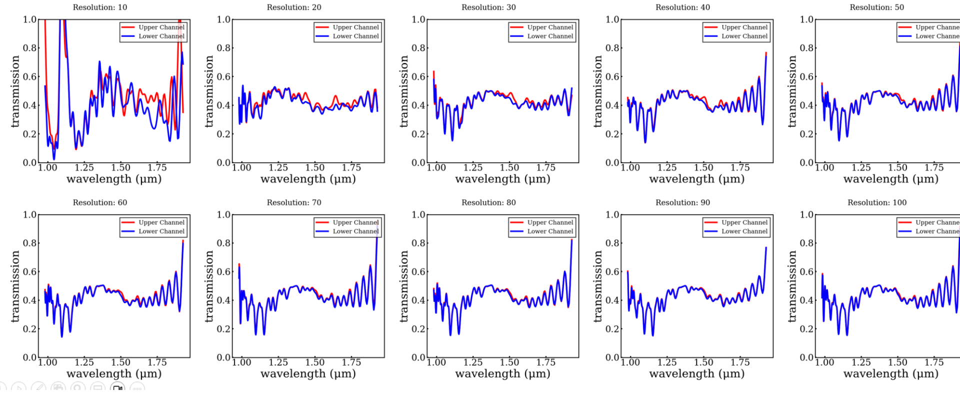


Figure 3.24: Transmission for filled box chip with varying light source width. Each plot corresponds to one of the outputs.

First we can look at how the width of the Gaussian light pulse affects the transmission. As shown in Figure 3.24, the narrow light pulse transmission mostly matches that of the wider light pulse. The only discrepancy occurs at the longest wavelengths of the narrow pulse. From this we can conclude that the accuracy of the transmission near the edge of the light pulse should be treated with caution. For this purpose, the light pulse used for the pipeline is wide enough that the wavelengths of interest do not lie near the edges of the pulse.



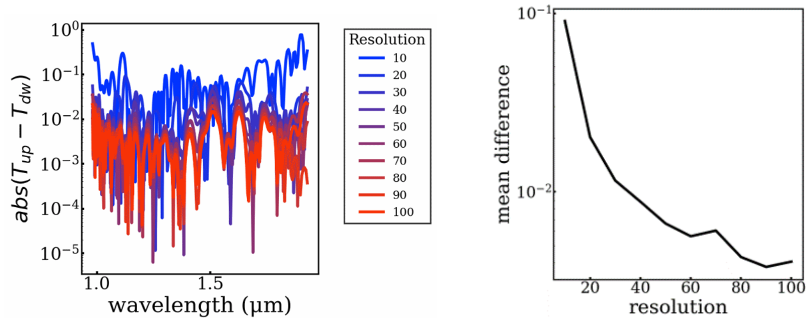
(a) Flux for the filled box chip with various resolutions.



(b) Transmission for the filled box chip with various resolutions.

Figure 3.25: Resolution test for the filled box chip with different resolutions.

Figure 3.25 shows the flux and transmission for different resolutions (pixels per µm). As can be seen, even at a rather low resolution of 20pxl/µm the transmission looks visually similar as compared to the much higher resolution. This gives us an idea of what resolution is necessary to produce a somewhat accurate result. A lower resolution is desirable, as it would reduce the simulation time and allow for a faster pipeline.



(a) Difference between transmission of output probes.

(b) Mean transmission difference between output probes.

Figure 3.26: Difference and mean difference of the transmission for the output probes of the filled box chip.

To characterize the difference in the transmission more precisely, the difference between the two outputs probes is plotted in Figure 3.26. As the resolution increases, the absolute difference shown in the left plot tends to zero. This is made more obvious when plotting the mean difference over all wavelengths as shown on the right.

Figure 3.27: Growth used for resolution test.

While the filled box chip is useful to determine the resolution range for an accurate FDTD simulation, we also need to test a grown chip to make sure the finer structures accurately interact with the light. Figure 3.27 shows the growth used in the following resolution test. The growth features some thin pathways which would be lost on lower resolutions, which makes it an interesting candidate. As stated previously, the growths have dimensions of 1500 × 1000pxl, which is then converted to a 4.2 × 2.8μm simulation box. The necessary resolution would then need to be 357pxl to maintain all the nanoparticles. Unfortunately, this resolution would lead to the FDTD simulations taking too long, rendering the pipeline unusable.

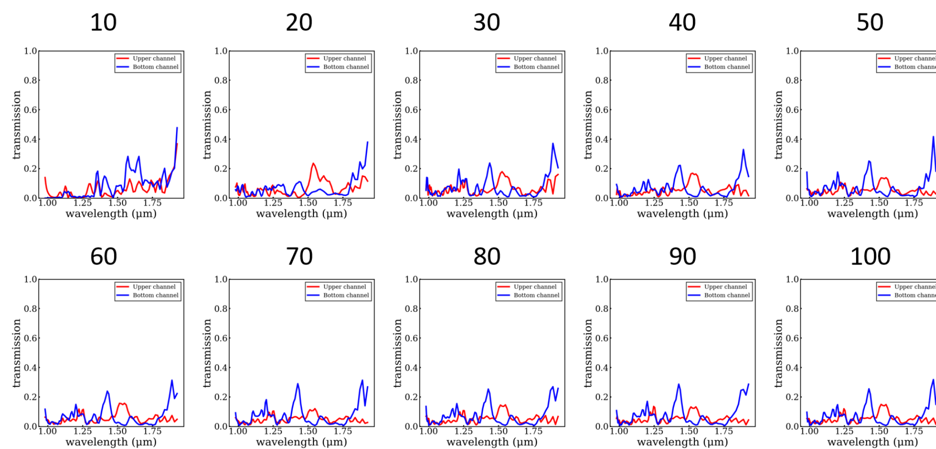


Figure 3.28: Resolution test for grown chip. Transmission is shown for the two output probes for various resolutions.

Figure 3.28 shows the transmission for the chip from Figure 3.27 with different simulation resolutions. In contrast to the filled box chip, there is a noticeable difference between resolution 20 and 30. As will be shown in a subsequent result, a resolution of 25 offers a good trade between accuracy and simulation time.

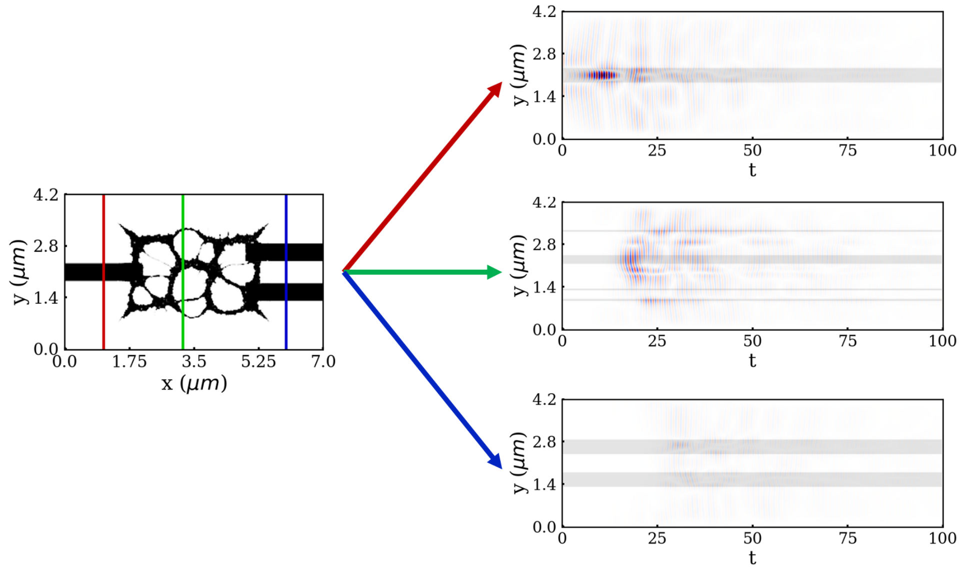


Figure 3.29: Electric field for three different slices during a FDTD simulation.

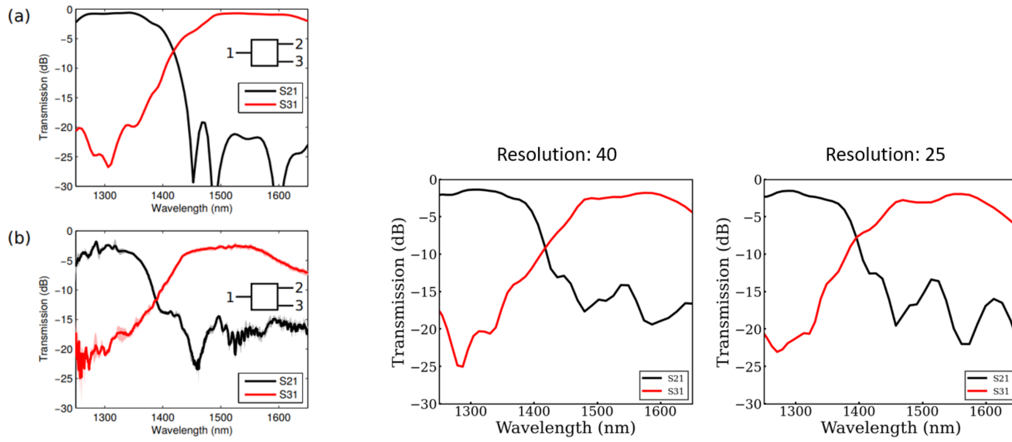
The simulation time is also dependent on the maximum simulation time. We want to give enough time for the light to propagate through the chip and give any light trapped in the chip time to escape. Figure 3.29 shows the electric field time evolution in three different slices, as indicated. From these slices we can see that the light passes through the middle of the chip around $t = 25$, while most of the light has exited the chip around $t = 50$. Running for a total of 100 time units therefore gives a good buffer to ensure the simulation has finished.

3.3.5 3D photonic chips

With the link between growth and FDTD established, the next step is to add another dimension. In all previous examples, the FDTD simulations were done in 2 dimensions. For realism, going up to 3 dimensions is necessary. The first step to accomplish this is by transforming the 2 dimensional array of the grown chip to a 3 dimensional photonic chip. This also necessitates extending the FDTD solver, which is implemented in PyMeep.

Figure 3.30: Example of a 2 dimensional chip transformed to a 3 dimensional photonic chip. The chip is constructed to be made out of silicon Si, resting on a bed of silicate SiO_4.

As a baseline to compare to, the photonic chip from [1] is used. This paper had the objective of designing a wavelength splitter for wavelengths λ = 1300nm and λ = 1550nm. In contrast to the pipeline presented here, this chip design is the result of a gradient based inverse design process. Where the optimization first continuously varies the electric permittivity of the region, after which the structure is converted to a discrete silicon and air material and fine-tuned using gradient descent. To replicate the design in this thesis, the final schematic of the chip presented in the paper [1] was converted to a black and white image. This was then converted into a binary `Numpy` array which could then be used in the same way as the growths from the lattice gas model. As shown in Figure 3.30, the chip is padded in all directions to extend the probes and give a buffer between the chip and the boundaries. The third z axis is then added as shown in the bottom plot. In this example, the 2 dimensional array was expanded into a very large 3 dimensional array that . The purple represents the siliccate SiO_4, while the yellow is silicone Si. This configuration matches the one in [1].

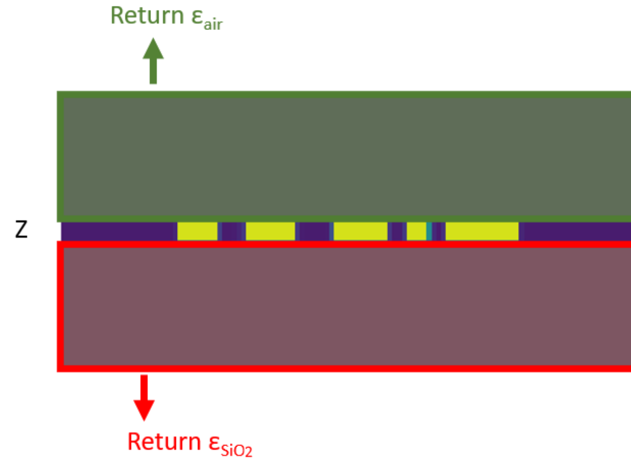


(a) Original paper results. Simulated (top) and experimental (bottom).

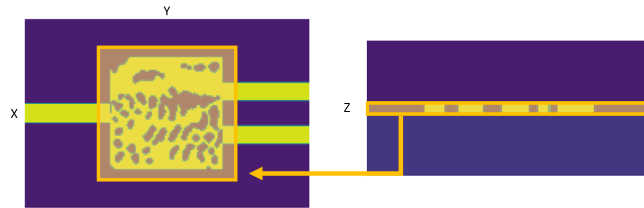
(b) Replicated results with 3 dimensional chip.

Figure 3.31: Replicated transmission spectra for chip from [1].

To replicate the results in [1], we want to calculate the transmission over a wide range of wavelengths. A Gaussian light source with a width that contains both target wavelengths is used. The light is oriented in the fundamental TE mode for the waveguide. Figure 3.31 shows the results of the replicated test using the reconstructed chip from the paper as shown in Figure 3.30. The transmission spectra from the replicated chip as shown on the right matches closely to the results from the paper, shown on the left. From this we can conclude that the simulation parameters are correct, and that a resolution of 25 is sufficient for our desired accuracy.



(a) Epsilon function output based on Z coordinate.



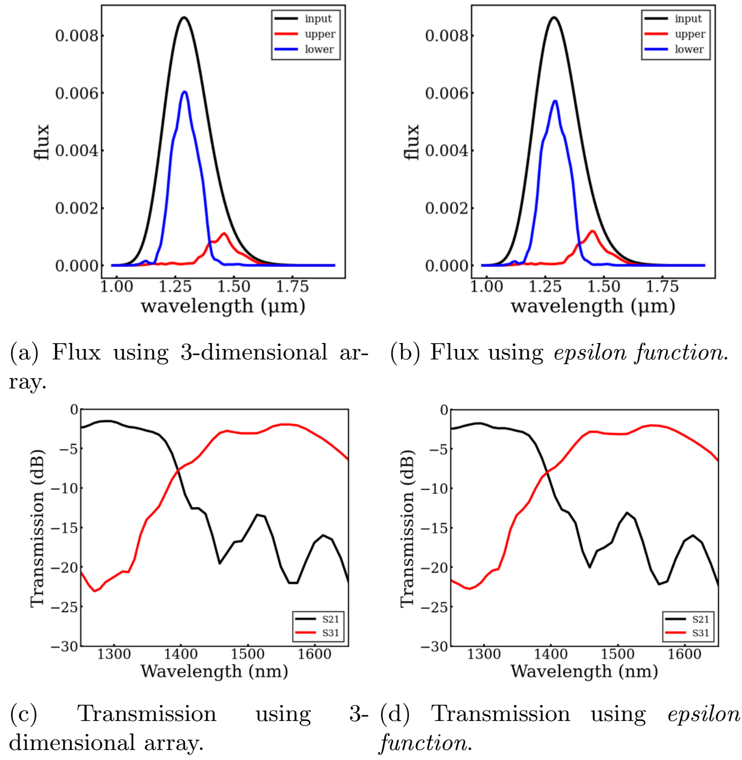
(b) Epsilon function output for chip based on X, Y coordinates.

Figure 3.32: Implementation of epsilon function to define photonic chip structure. Based on Z coordinate, returns either the dielectric constant for air or SiO_4. If the Z corresponds to the chip, returns a value based on the X and Y coordinate.

With the large 3 dimensional array implementation of the photonic chip, an issue concerning computer memory quickly arises. Even if the 3 dimensional conversion is done after manually rescaling the array based on the resolution, it is an inefficient method. Instead, we can define a function that takes spatial coordinates and returns the dielectric constant at that point. This function is referred to as an *epsilon function*.

This is rather easy to construct for the photonic chip used in this project. First, we check the Z coordinate and return either the dielectric constant of air or SiO_4 if it lies in the corresponding region, as shown in the upper plot of Figure 3.32. If we lie in the region of the chip, we check the X and Y coordinate to determine whether to return the dielectric constant of air or Si. For any coordinate lying outside the defined chip array, we simply take the border, effectively padding the chip in all directions to infinity. This is shown in the lower plot.

With this method we no longer need to store a large 3 dimensional array, but instead the original chip. In fact, we no longer need to pad the array to run the FDTD simulation as the epsilon function does it for us.



(a) Flux using 3-dimensional array.

(b) Flux using *epsilon function*.

(c) Transmission using 3-dimensional array.

(d) Transmission using *epsilon function*.

Figure 3.33: Comparison between 3-dimensional array and epsilon function implementation for the transmission on a test chip.

Figure 3.33 shows a comparison for the chip. As can be seen, both methods yield close to equal transmission spectra. The memory intensive implementation using a 3 dimensional array can therefore be switched to the more efficient epsilon function implementation.

3.3.6 Scoring using FDTD simulations

The final necessary component for the full growth pipeline is deciding how to score the growths using the transmission spectra. There are many ways of doing this, explained below is one such method which we chose for this project.

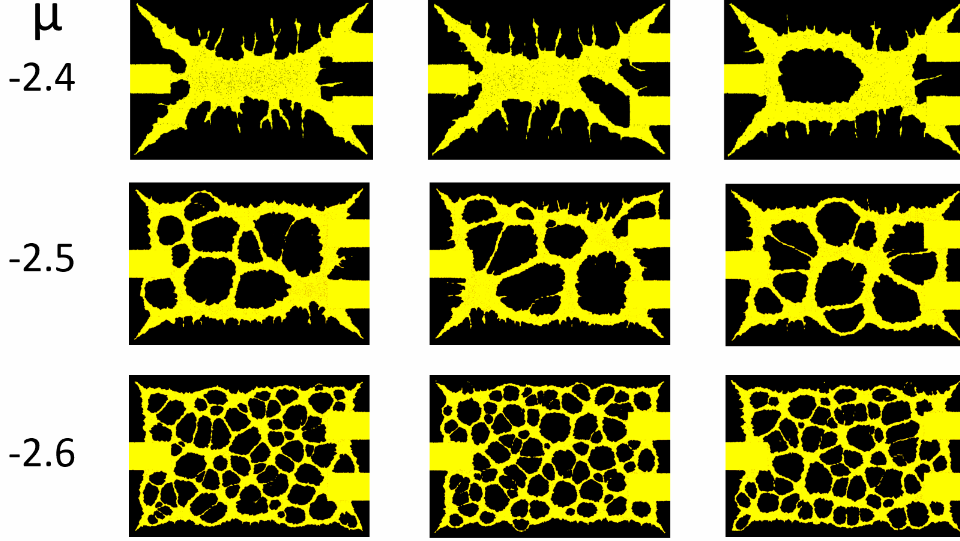


Figure 3.34: Examples of growths with constant growth parameters for different μ.

To demonstrate the scoring method, 200 growths were run for each μ and with constant growth parameters as shown in Figure 3.34. This Figure shows 3 random growths for each μ. The temperature was $k_B T = 0.2$, nanoparticle coverage of $frac = 0.3$, and the total number of simulation steps was 2000. The chemical temperature μ is shown for the corresponding row of growths. As can be seen, μ heavily impacts the characteristics of the final structure that is formed. On average a single growth took 40 minutes to complete. The growths were run in parallel to reduce the total time it took to run all of them.

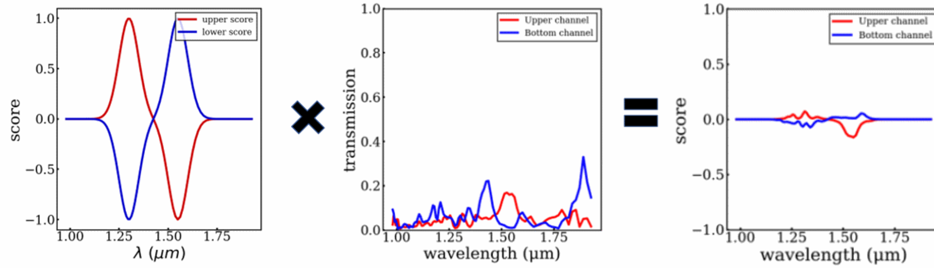
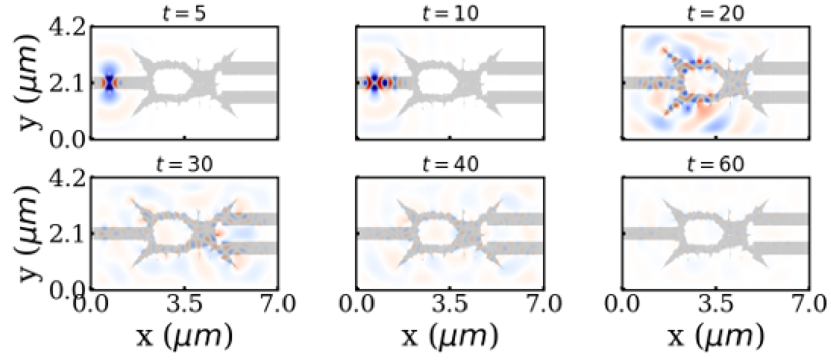
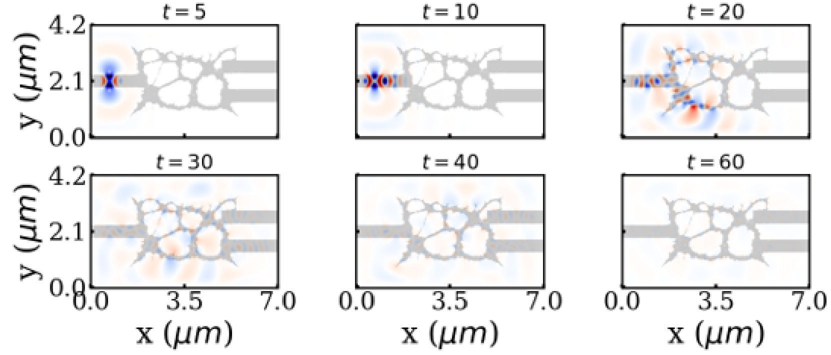


Figure 3.35: Scoring method for growth pipeline. Transmission spectra is multiplied by a scoring spectra. The result is then summed to produce a single value for the score.

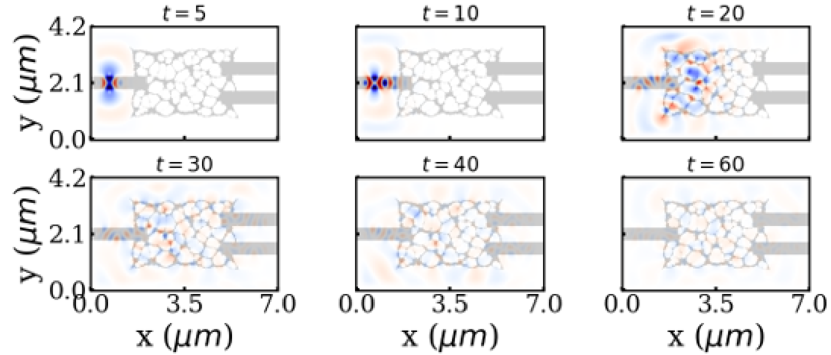
To score the these growths using the transmission from the FDTD simulations, a method as shown in Figure 3.35 is used. The transmission spectra for the two outputs (middle) is multiplied by the corresponding scoring spectra (left) to give a score array. In this example, the upper output shown in red wants to maximize transmission for wavelengths around 1.3µm, and minimize transmission around 1.55 µm. The lower output has the opposite objective. A final score is then calculated by summing the product of these two spectra.



(a) Electric field evolution for growth with $\mu = -2.4$



(b) Electric field evolution for growth with $\mu = -2.5$



(c) Electric field evolution for growth with $\mu = -2.6$

Figure 3.36: Evolution of electric field for three different growths with different chemical potential μ.

Examples of the propagation on light through these grown chip is shown in Figure 3.36. For a selected growth with the stated μ, 6 snapshots are shown taken at different points during the FDTD simulation. This Figure gives an idea as to how light interacts with the grown chip. In the $t = 20$ plot, we can see that a significant portion of the light leaves the chip instead of being diverted to the outputs. For the growth pipeline, we therefore desire this light loss to be minimized.

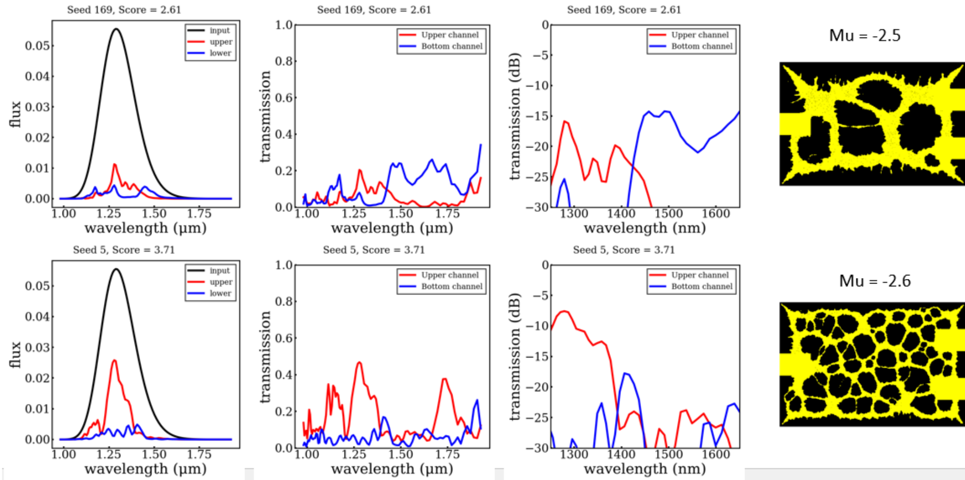


Figure 3.37: The best scoring growths for the constant growth parameter scoring test. The flux, transmission and corresponding chip is shown for $\mu = -2.5, -2.6$.

The best scoring growths for $\mu = -2.5, -2.6$ are shown in Figure 3.37. The flux, transmission and the growth are shown for both chemical potentials. As can be seen, both growths act as a very rough wavelength splitter. One output maximizes wavelengths near 1300nm while the other maximizes wavelengths near 1550nm. Both outputs minimize the other wavelength. As this was a simple example, neither of these structures are particularly good. The transmission is very low for both chips, which would make this impractical in any real life application.

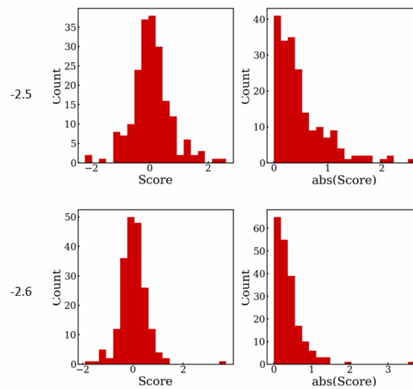
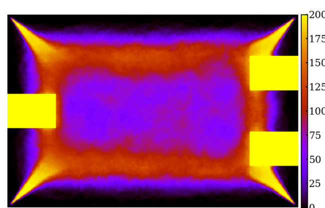


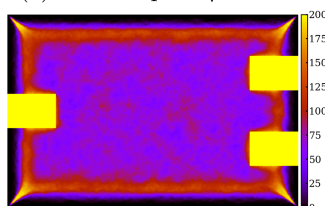
Figure 3.38: Score distributions for all 200 growths with $\mu = -2.5, -2.6$. Both the score and absolute score is plotted.

The scores for all growths are shown in Figure 3.38. For both values of μ, a distribution for the normal score and the absolute of the score are shown. The absolute of the score can be useful if the outputs do not need to have a specific wavelength associated to them. If all we want is a wavelength splitter, it does not matter if the light of 1300nm is maximized at the upper or lower outputs, as long as the 1550nm light is maximized at the other one. A very low score using the normal scoring thus signifies a good wavelength splitter but with the outputs flipped.

From this we can already predict a problem the full pipeline will face. For a single policy, which in this case is the constant growth parameters, the variance on the score is significant. As will be shown later, this proved to be a serious complication for this project.



(a) Heat map for $\mu = -2.5$.



(b) Heat map for $\mu = -2.6$.

Figure 3.39: Growth heat map for score test with constant parameters. Colour corresponds to how many growths have nanoparticles in that location.

Another way to visualize the wide range in produced growths is by superimposing all the growths as shown in Figure 3.39. This produces a heat map which shows where nanoparticles appear in the growths. As can be seen, for $\mu = -2.5$, the 4 corners almost always have the spike-shaped structure. The next most prominent feature is the red square which links these corners, appearing in most of the growths. The purple in the middle signifies that here is a large variety of structures present here, few of which are shared in any two growths. A similar conclusion can be drawn for $\mu = -2.6$, although the region of difference is much larger.

3.3.7 Full pipeline test

With all the components for the growth pipeline now implemented and tested, the next step is to link all the components together and run the full pipeline. To do this, the different components were coded to be their own separate modules, which can be easily individually changed if so desired.

The first test using the full pipeline had the objective of splitting light of wavelength 1300nm and 1550nm as the example given for the scoring. For this initial test, the population based genetic algorithm was used. A population of 40 networks was initialized, and were trained using Algorithm 2. As each network has to direct a growth, the growth and scoring portion of the pipeline were run in parallel. Each network is given a unique random seed to ensure all growths are distinct. These were run on one of the Vector institute supercomputer cluster. Once all growths are finished and scored, the main thread

performs the sorting, culling and mutating of the networks. This process is then repeated for the next epoch.

As this test did not include some optimizations, the 17 epochs took a total of 24 hours to run. The time it takes an epoch to run was further reduced for later tests.

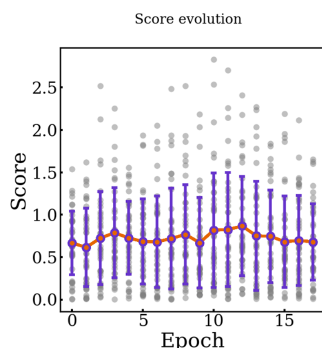


Figure 3.40: Full pipeline score evolution. Mean is shown in orange, standard deviation in purple. The grey dots are individual scores.

The score evolution for this test is shown in Figure 3.40. As can be seen, very learning occurs in these epochs. There are a few reasons as to why that might be. While the relatively few number of epochs might be a potential cause, we would still expect to see some improvement over these epochs. Instead, we believe that the wide score distribution for a given policy is the main culprit. As was discussed in the preceding scoring section, the variance in the score for a given policy is very high. It is clear that a single score often misrepresents how good a policy actually is. Therefore, sorting policies based on a single score is a futile task.

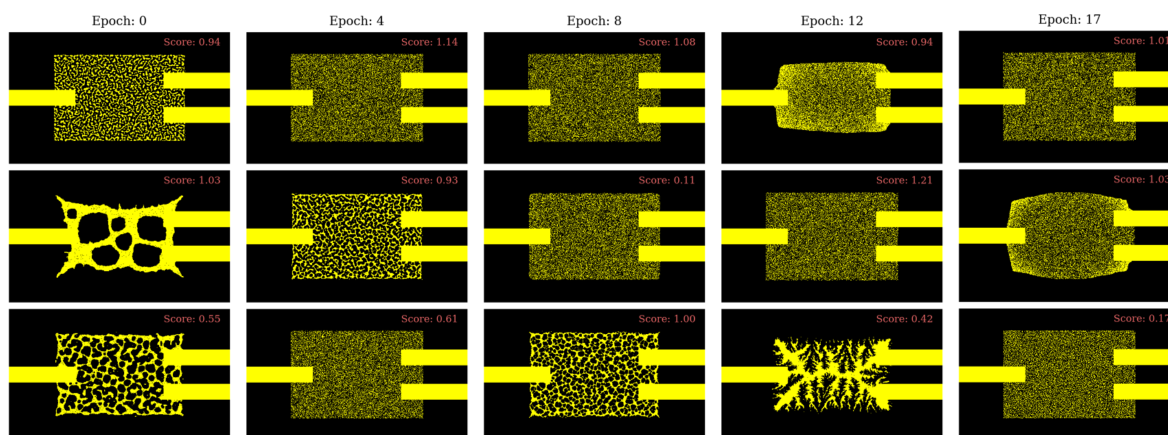
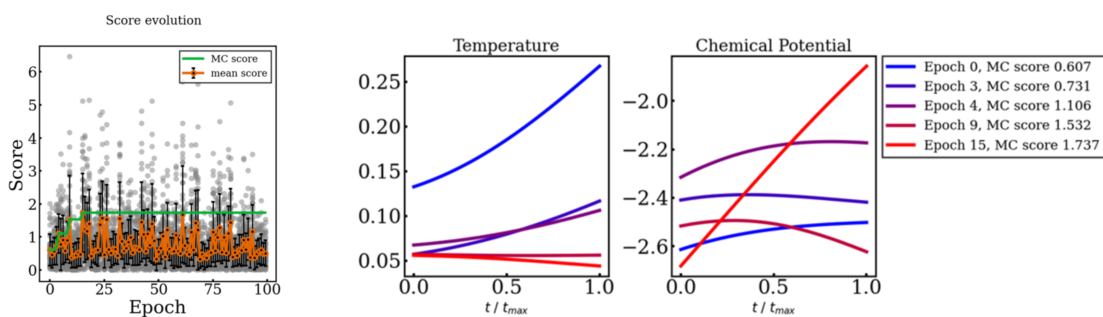


Figure 3.41: Random selection of growths produced by network policies for different epochs.

Figure 3.41 shows a selection of network growths for a number of epochs. The displayed growths were chosen randomly within the population. A wide range of structures can be seen, ranging from noisy distributions of nanoparticles to clearly defined channels. The policies which do not generate any defined

structures are unwanted as these would not be realistic, but the current implementation of the pipeline does not punish these in any way. We believe that the structures which form clear paths for the light would have the ability to produce better scores, and thus the noisy policies would be culled. Allowing these policies to stay is also beneficial for exploration, as we do not want to constrict the potential policies to such a degree that no interesting structures form.

For the second test, the policy training was switched to the MC algorithm shown in Algorithm 3. In this test, $N = 40$ growths are run and scored in parallel following the single network's policy. If the mean score is less than the previous mean score, the network mutations are accepted. Otherwise, the mutations are rejected. While this algorithm takes the same amount of time to run a single epoch, using the mean score is a much more accurate way of comparing two policies. One possibility would have been to run many growths for each network in the population based genetic algorithm, but this would have drastically increased the time it takes to train. A single network will hopefully show slow but constant improvement as the policy is fine-tuned.



(a) Score evolution. The orange represents the mean score for that epoch, while the green represents the current best score.

(b) Policies which improved the mean score.

Figure 3.42: Results of the MC pipeline test with 40 growths. The score evolution as well as the policies which improved the mean score is shown.

The results for this test is shown in Figure 3.42. As can be seen, minimal learning occurs. However, it is an improvement over the previous method. For the first 15 epochs the mean score steadily improves. Unfortunately, the mean score does not seem to be improving, preventing learning from occurring.

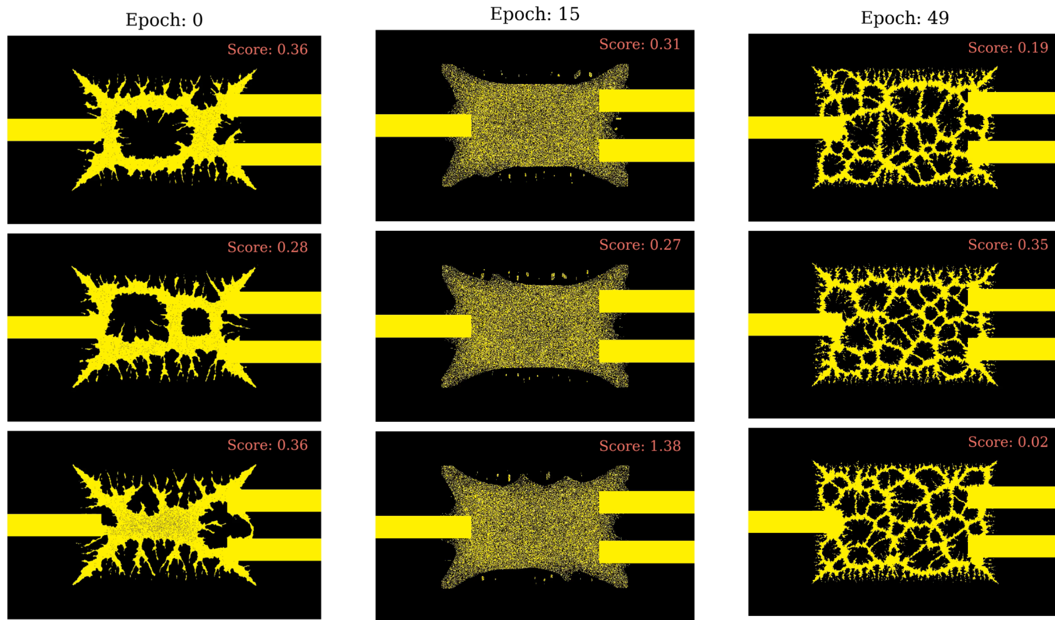
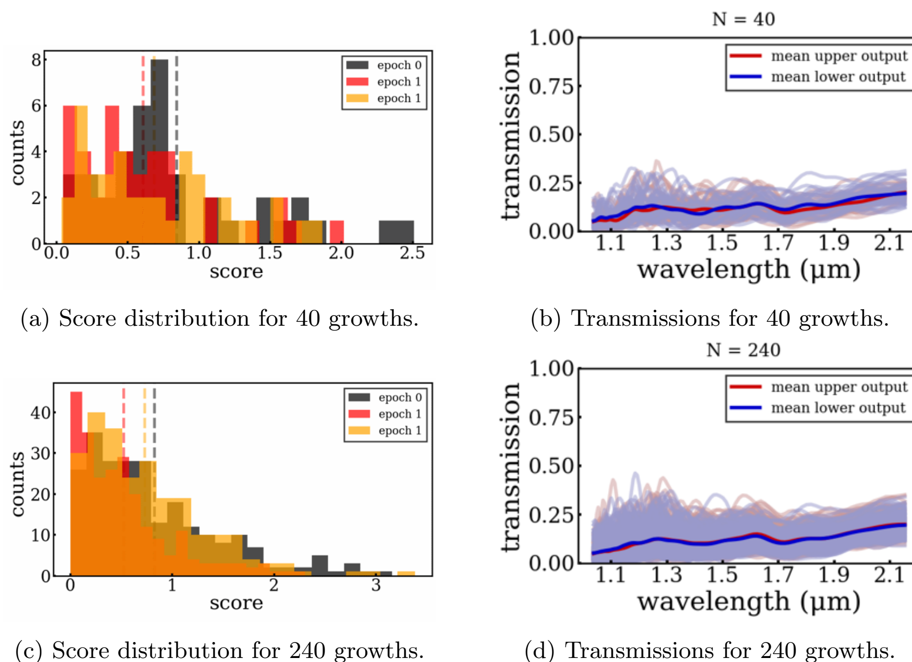


Figure 3.43: Random selection of growths from different epochs during training. The growths' score is shown in the top corner.

Figure 3.43 shows a selection of growths with the corresponding score for different epochs. As compared to the previous test, we can see how all the growths in a particular epoch possess similar structures. This difference stems from growing multiple growths using a single network as opposed to each network having an individual growth. However, significant differences can be seen in the growths. For example, the bottom growth for epoch 0 does not have any holes present in the other two. This explains the large standard deviation in the score, as growths from the same policy has important structural differences.



(a) Score distribution for 40 growths.

(b) Transmissions for 40 growths.

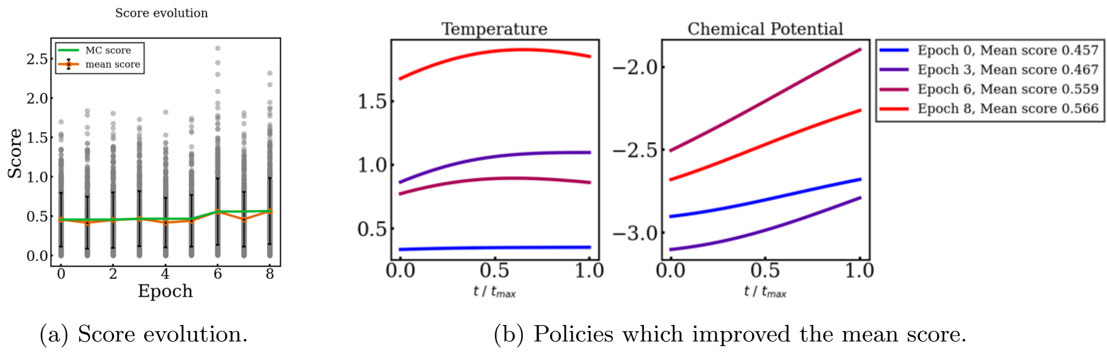
(c) Score distribution for 240 growths.

(d) Transmissions for 240 growths.

Figure 3.44: Score distributions for 40 and 240 growths. The initial score (grey) and the score from mutating once (red, yellow) is shown.

To examine the score distribution, Figure 3.44 shows the distribution for the first initial network, as well as two distributions for that network mutated once. By plotting these, we can see how the distribution is changing during learning. We would want these distributions to be clearly distinct, as the more similar they are the harder it is to distinguish which policy is better. In this example, we can see that 40 growths produces ill-defined distributions, while 240 growths has better defined ones. However, the distributions are still not easily distinguishable even for this amount of growths.

The next test is to increase the number of growths to 520 to see whether more growths will improve learning. Unfortunately, only a few epochs can be run for this test, as each epoch takes much longer due to not all growths being run in parallel.

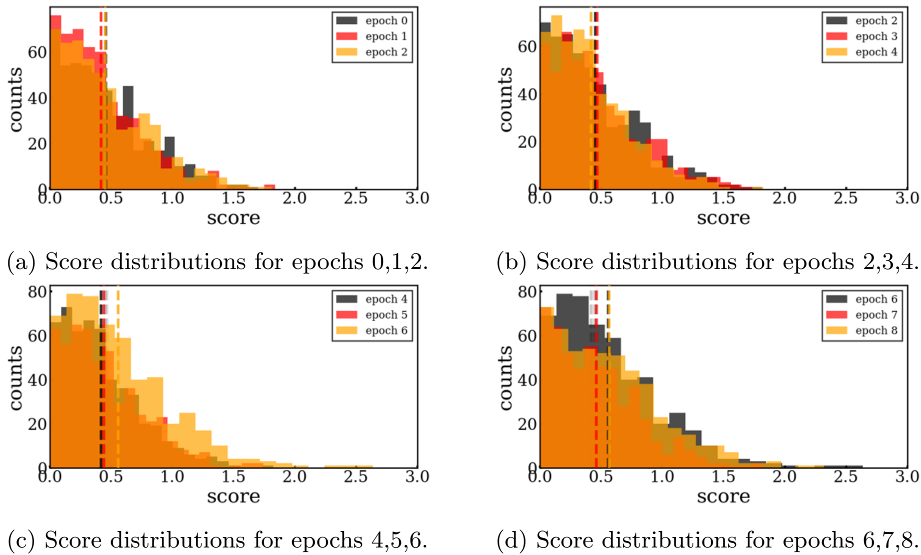


(a) Score evolution.

(b) Policies which improved the mean score.

Figure 3.45: Results of the MC pipeline test with 520 growths. The score evolution as well as the policies which improved the mean score is shown. The MC score is the current best mean score.

The early epoch results is shown in Figure 3.45. From this, we can see the width of the score for every epoch. While this amount of growths might lead to a steady learning, the computation cost is too high for this initial phase of experimentation. With that said, the little learning that occurs is still a good sign that the pipeline has potential.



(a) Score distributions for epochs 0,1,2.

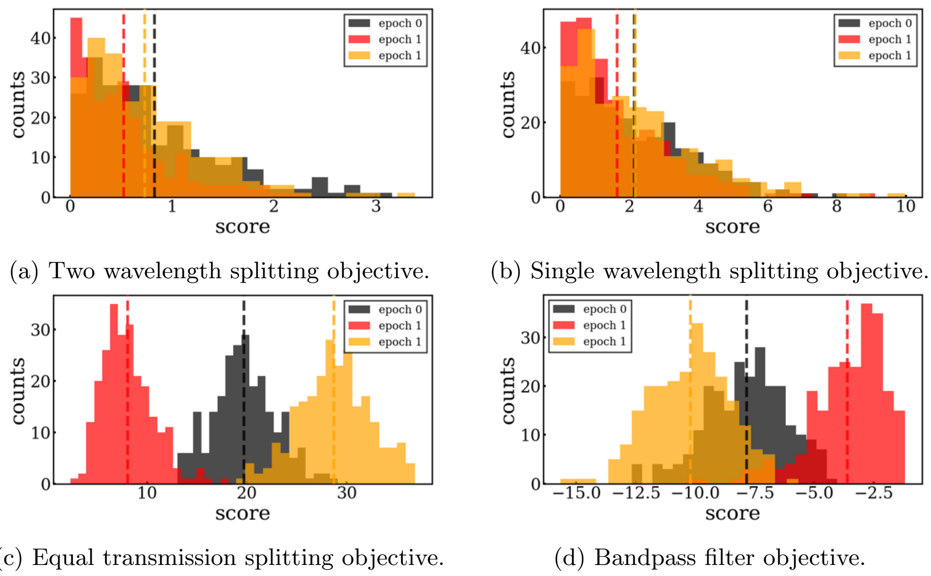
(b) Score distributions for epochs 2,3,4.

(c) Score distributions for epochs 4,5,6.

(d) Score distributions for epochs 6,7,8.

Figure 3.46: Score distributions for 3 sequential epochs for the 520 growth MC pipeline test.

The score distributions for these 8 epochs is shown in Figure 3.46. Each plot shows three consecutive epochs to help comparison. Unfortunately, the distributions do not have significant difference between epochs. While there is a possibility that this number of growths will be able to continue learning, a more efficient test is to look at different scores. Finding an objective which has distinct distributions on a per-epoch basis would aid learning greatly, as fewer growths would be necessary to more accurately compare two policies.



(a) Two wavelength splitting objective.

(b) Single wavelength splitting objective.

(c) Equal transmission splitting objective.

(d) Bandpass filter objective.

Figure 3.47: Examples of potential transmission based objectives.

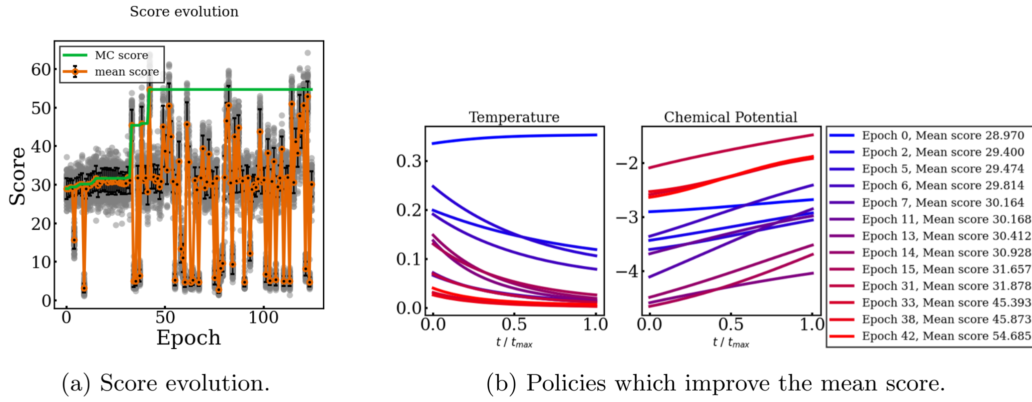
Figure 3.47 shows the score distribution for four objectives that can be constructed using the chip transmission spectra. The top left plot shows the score distribution for the 1300nm and 1550nm wavelength splitting as previously used. Right of this is the single wavelength splitting. This objective seeks to split 1300nm light out one output, and the rest of the light out the other. This objective is similar to the previous one, and the distributions corroborate this. A large difference is seen in the following two distributions however. The bottom left plot shows the distributions for equally separating the light between the two outputs. This score is defined as:

$$score = sum(T_{up} + T_{dw}) - abs\big(sum(T_{up} - T_{dw})\big) \tag{67}$$

Where T_{up} and T_{dw} is the transmission of the upper and lower outputs respectively. The sum runs over a range of wavelengths that have input flux above a lower limit. This ensures the transmissions are accurate. These are vectors of the transmission for the different wavelengths of the light. The score is thus maximized when the total transmission $T_{up} + T_{dw}$ is maximal, and the difference in the transmission between the two outputs $T_{up} - T_{dw}$ is minimal. This occurs when all the input light leaves through the two outputs, and both outputs transmit exactly half of all wavelengths. The square chip

shown previously in Figure 3.23 is an example of a structure which scores high for this objective. The final objective shown in the bottom right is a bandpass filter. This has the objective of only allowing certain wavelengths through the output, penalizing all others. In this case, we want to allow wavelengths near 1300nm and block all others.

The important observation to make for this Figure is the ability to tell distributions from different epochs apart. For the wavelength splitters, distinguishing the distributions is nearly impossible. The other two objectives however have clearly distinct distributions. If less growths were used, the calculated mean would still be useful to compare the different policies.

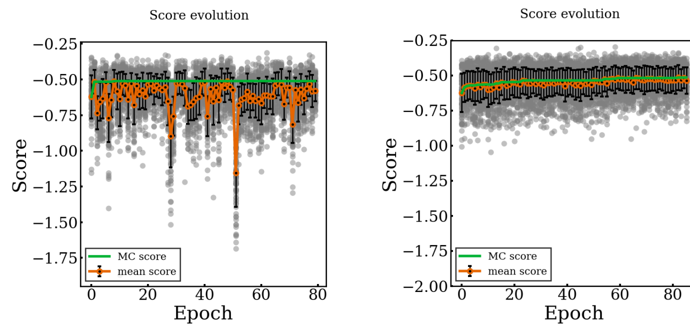


(a) Score evolution.

(b) Policies which improve the mean score.

Figure 3.48: Pipeline results for even split objective.

The results for the pipeline trained on the even split objective from Equation 67 is shown in Figure 3.48. The score evolution with this different objective differs from the previous objective. While learning is initially slow, a few massive spikes drastically increase the score before it plateaus. Once this happens, the mean score becomes extremely unstable. One possibility is that the mutation size was too large for the current learning state, leading to mutations not following a given learning trajectory, and instead randomly exploring. Lowering the learning rate would however lead to much longer training time. This is an issue as this result needed a few days to finish running.

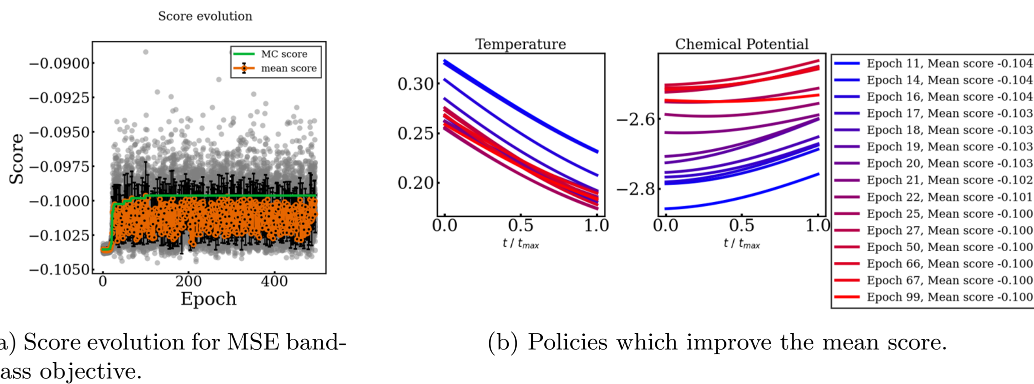


(a) Score evolution for bandpass objective with $\sigma = 0.1$.

(b) Score evolution for bandpass objective with $\sigma = 0.01$.

Figure 3.49: Bandpass filter test with varying mutation standard deviation σ.

The impact of the mutation standard deviation σ can be seen in Figure 3.49. For this test, the bandpass filter objective is used. The score is maximized if only light with wavelength near 1300nm is transmitted through the now single output. As can be seen, the larger σ leads to the plateauing seen before, with a very noisy mean score evolution. With the smaller σ, learning is constant albeit slower. The mean score also remains close to the current best MC score, meaning that the algorithm takes small incremental steps. However, in the number of epochs run, the smaller σ only attains a score similar to what the larger σ within a few epochs. The question would be whether the smaller σ keeps learning after reaching the plateau. The learning does seem to slow down for the smaller σ, so it would need to run for a long time. One possibility would have been to implement the adaptive form of MC, which would scale the σ as consecutive rejected moves occurs.



(a) Score evolution for MSE bandpass objective.

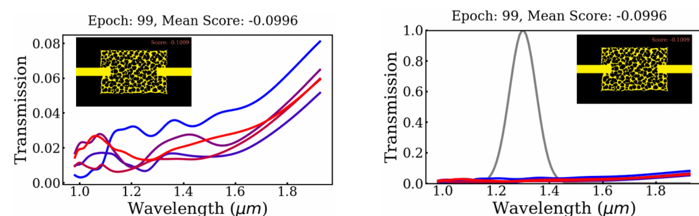
(b) Policies which improve the mean score.

Figure 3.50: Pipeline for piepeline with MSE bandpass scoring.

With the scoring method explained in Section 3.3.6, the final score is made up of two components, a positive part when the correct wavelength exits through the corresponding output, and a negative part when the other wavelength does. For the bandpass objective, this was altered to penalize any wavelength outside of the correct one. A problem that can arise when the score traverses from negative to positive is the algorithm getting stuck at zero. The reasoning for this is that it is a lot easier to have a zero score - all you need is zero transmission, then it is to have a positive score with some transmission. From the initial parameters, the algorithm is rewarded for following a trajectory that blocks all light from leaving through the outputs. Once it has reached a policy that consistently blocks all light, the step it would need to take to permit the desired light from leaving might be too complicated for the basic implementation done here. To try and alleviate this problem, the score was altered to be the MSE. By taking the mean squared error of the different of the transmission spectra and target transmission spectra, the score will not be rewarded for zeroing the transmission. The negative of this score is taken by convention, so a maximize the score is still the objective.

The results for the bandpass filter with MSE scoring is shown in Figure 3.50. Unfortunately, this change was not enough to give the pipeline the ability to continue learning past the plateau. The policy

evolution followed a trajectory which decreased the temperature and increased the chemical potential, before plateauing at some score.



(a) Transmission for best epoch. (b) Transmission with target spectra.

Figure 3.51: Transmission plots for MSE bandpass score test. The plots show the transmission for 5 randomly chosen growths from the best epoch. The right plot additionally shows the target transmission spectra in grey.

From Figure 3.51 we can better see how poorly the pipeline has performed in this objective. As can be seen from the right plot, the transmission for the best policy produces transmission spectra drastically different than the target spectra. Again, we see the transmission being near zero for all wavelengths, hinting that the previous scoring method might not have been the problem.

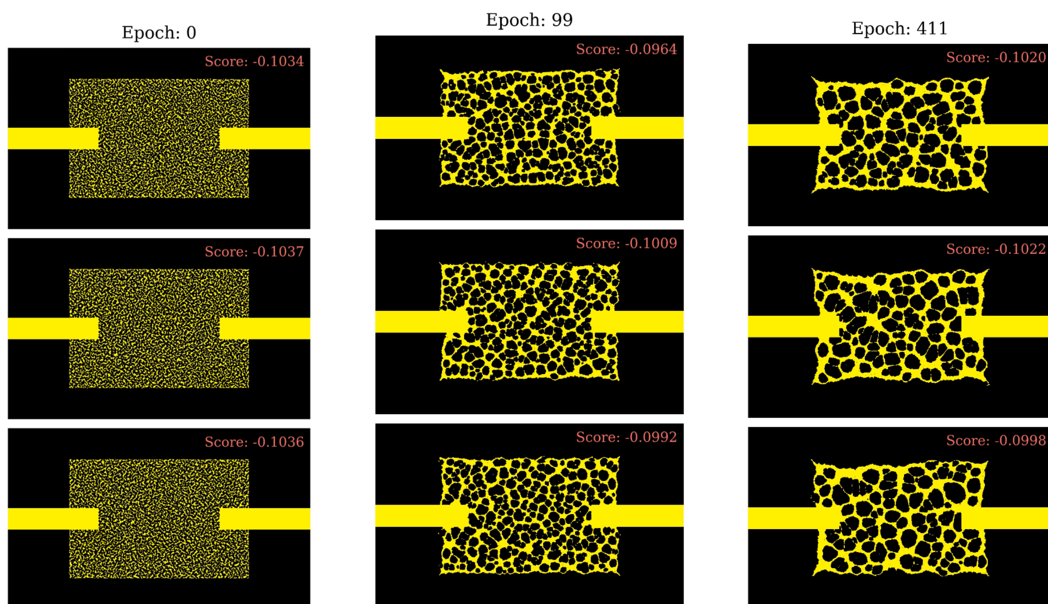


Figure 3.52: Growth for three different epochs from the MSE bandpass test. The first epoch, best scoring, and final epochs are shown.

A selection of growths is shown in Figure 3.52. From the starting epoch which had a policy producing noise, the pipeline then learns to produce growths with better defined structures.

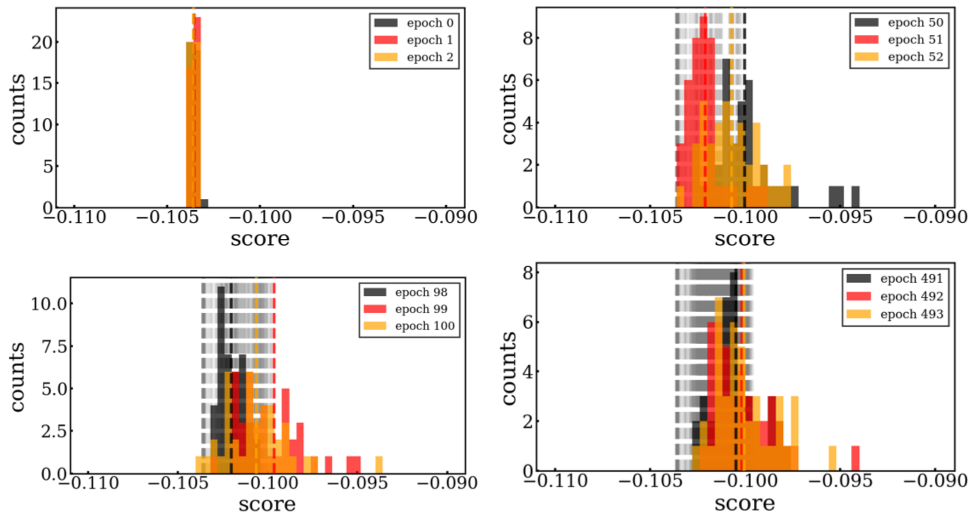


Figure 3.53: Score distributions for the MSE bandpass test. Three consecutive epochs are shown in each plot, from different points during training.

The score distributions for this test is shown in Figure 3.53. Here we can observe how to distributions are decently distinct from one another. However, the distribution for near the top scoring epoch (bottom left) are rather similar. The low number of growths per epoch may be the culprit for this.

3.3.8 Implementing spatial control methods.

With these previous tests the network has minimal control over specific features of the final grown structure. This is a consequence of only global growth parameters being controllable in the pipeline. This is meaningful because, for example, the network has no way of controlling where nucleation initially happens. This is incredibly important in determining the layout of the air bubbles in the final growth.

To investigate whether a spatial form of control helps growing good structures, a simple form of attraction is implemented. This allows the network to control the location and strength of a point or line of attraction. With this change, the network has much more control on the spatial layout of nanoparticles.

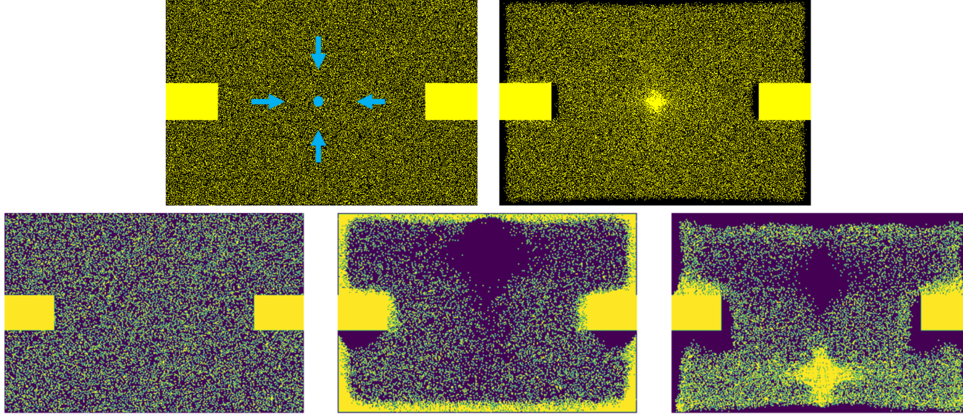


Figure 3.54: Point attraction implementation. Top shows an example of the process, while bottom plots shows a growth first experiencing repulsion and then attraction of two different points.

Figure 3.54 shows the implementation of the point attraction. The network outputs a x and y coordinate corresponding to source of the attraction. This point is used to calculate a distance between the nanoparticle and the source.

$$p_{acc} = min\big(1, e^{-(\Delta E + att_E)/k_B T}\big), \tag{68}$$

where att_E is the attraction energy difference between the nanoparticle before and after movement. This attraction energy scales inversely with the squared distance.

$$att_E = a_E \frac{1}{\epsilon + D^2}, \tag{69}$$

where D is the distance from the nanoparticle to the source, a_E is the attraction strength and ϵ is a small value added to avoid dividing by zero, for computation purposes. The strength value controls how much attraction or repulsion the nanoparticles experience. This attraction strength is also outputted by the network. This attraction energy is added to the Hamiltonian as an additional term, which affects whether proposed nanoparticle movements are accepted or rejected. Nanoparticles moving towards the source of attraction or away from the source of repulsion are more likely to be accepted, based on the distance from the source.

Repulsion occurs when the strength value is negative. The top plots show arrows which point in the direction of the force, as well as the growth after some epochs have based while experiencing this force. The nanoparticles clump together in the center, giving the network control over the spatial layout of the nanoparticles. The bottom plots show both repulsion from a point near the top, followed by attraction at another point.

By changing the location and strength of the attraction force, the network is able to form structures which were impossible when only have control over the temperature and chemical potential.

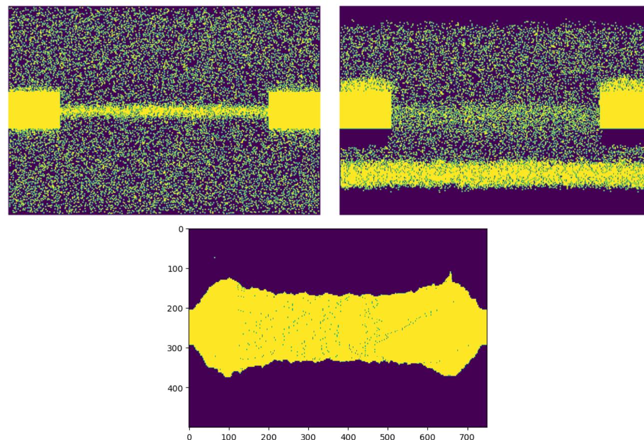


Figure 3.55: Example of line attraction. Top plots show nanoparticles being attracted to the center line followed by a lower one. The bottom plot shows the growth which happens if the attraction stays in the center for the whole duration.

In addition to the point attraction, Figure 3.55 shows the implementation for a line version. This attraction force strength falls off as the inverse distance:

$$att_E = a_E \frac{1}{\epsilon + (D)}. \tag{70}$$

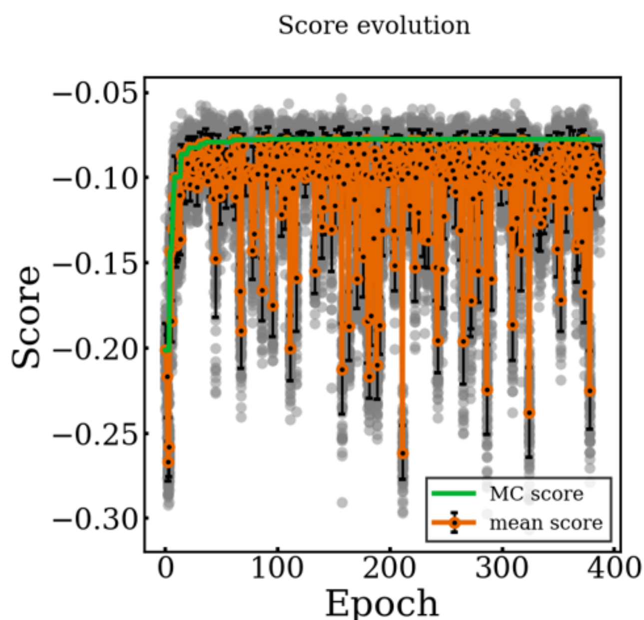


Figure 3.56: Score evolution for pipeline with linear attraction.

The results for the pipeline with linear attraction implemented is shown in Figure 3.56. The score rapidly improves in the first few epochs, and eventually plateaus around −0.08. As can be seen, the

mean score is then very unstable, signifying that the mutation size might be too large, preventing the network from following a specific trajectory.

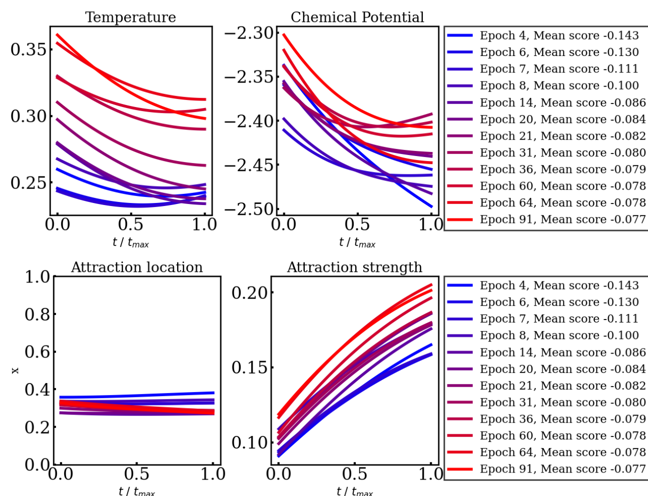


Figure 3.57: Policy evolution for pipeline with linear attraction.

The policy evolution is shown in Figure 3.57. In addition to the temperature and chemical potential (top), the attraction line x-coordinate and strength is shown in the bottom plots. Here the line x coordinate corresponds to the vertical axis in the growth plots, with origin at the top left corner. We can see that the attraction line coordinate did not change much , while the strength increased as learning occurred. Similarly, the temperature and chemical potential both increased.

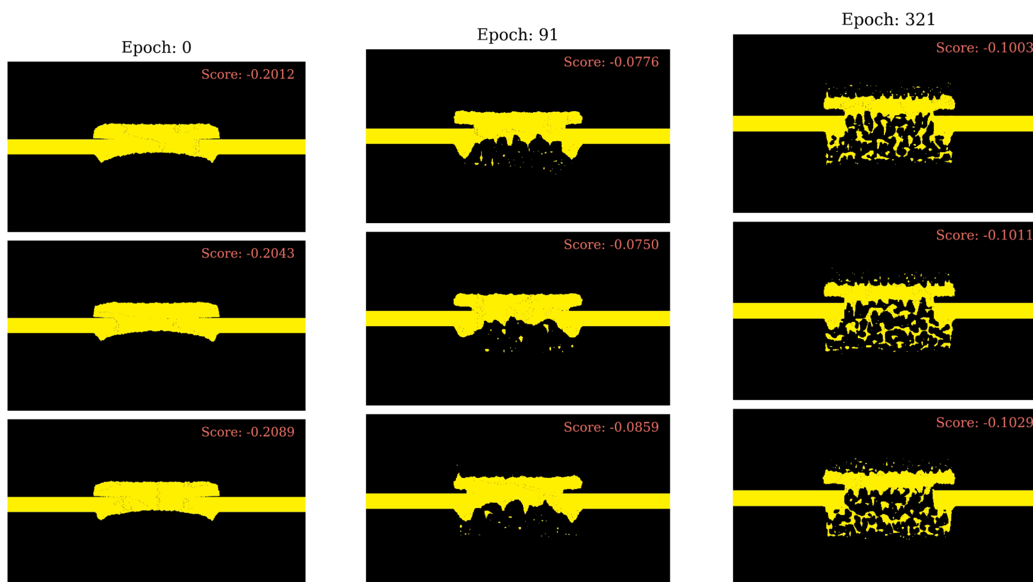


Figure 3.58: Random selection of growths from the first epoch, best scoring epoch and final epoch. The growth's score is shown in the top right corner of each plot.

Growths from this test is shown in Figure 3.58. We can see three growths from the initial epoch, top scoring epoch, and final epoch. As can be seen, the linear attraction allows the growths to be much more

similar in the same epoch, which helps reduce the variation in the score.

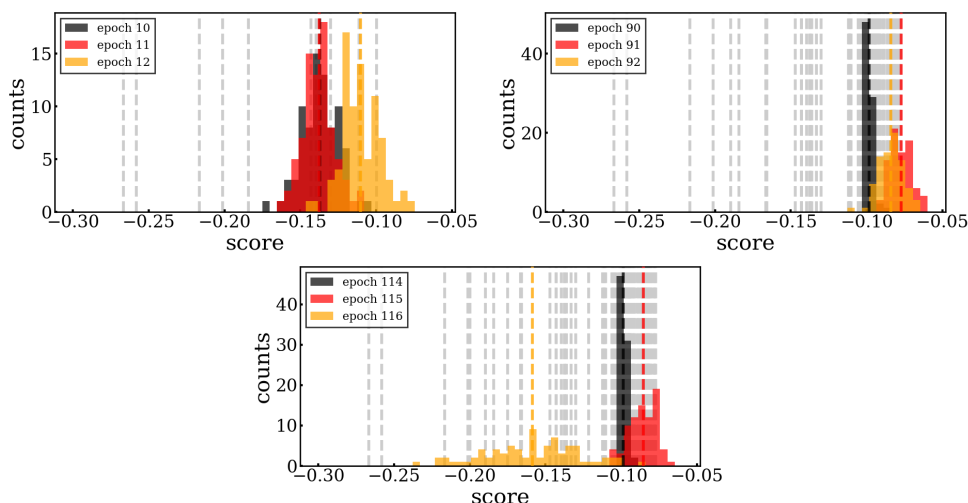


Figure 3.59: Score distributions for pipeline with linear attraction.

The score distributions for different epochs can be seen in Figure 3.59. These distributions are very distinct from epoch to epoch. This is a good sign for the pipeline, as the number of growths is sufficient to compare the mean score for different epochs with relatively high accuracy.

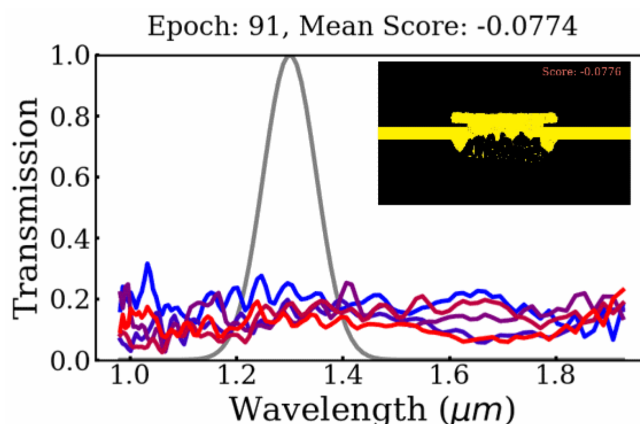


Figure 3.60: Transmission of best performing policy for a number of growths. An example of a growth is shown in the inset image.

The transmission for the best performing policy is shown in Figure 3.60. This plot shows 5 randomly chosen growths from this epoch, with an example of the growth shown in the inset image. As can be seen, there is still a lot of improvement that could be made, but this result is better than the previous tests. As opposed to the near zero transmission, the average growth allows about 20% of light through. Unfortunately, it does not seem to successfully filter out the undesirable wavelengths.

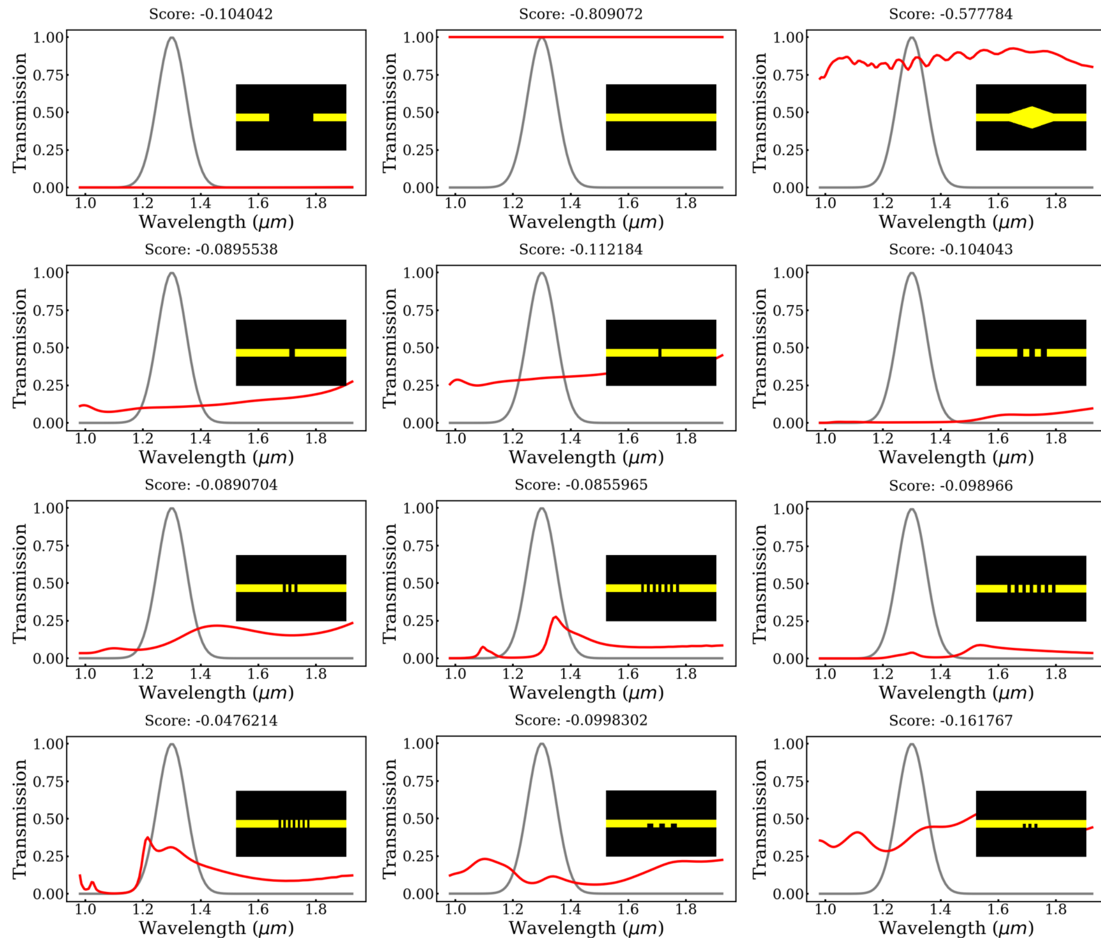


Figure 3.61: Manually designed structures with the corresponding transmission and score.

Figure 3.61 might give us some insight into some of the structural elements shown in the growths. This Figure shows a few manually defined structures with the associated transmission spectra in red. The spectra in black is the target spectra as usual. As can be seen in the bottom left structure, reaching a better score than what the growth reached is possible. The caveat here being that these structures do not necessitate growing, which drastically simplifies the process and allows very specific dimensions to be used with high symmetry. In the growths of epoch 91 in Figure 3.58, a rough symmetric pattern similar to the ones in the manual structures can be seen.

3.4 Discussion

While no specific policy which produced exceptional growths were shown, the results provide an exploratory view on the challenges of growing structures which necessitate precise structural elements with minimal control. While proposing structures which fulfil the target objective better is possible, it would be much more challenging to propose better growth policies. The limited control over the fine structural details of the growth adds a significant hurdle towards creating high scoring growths, but mimics how these would be grown in real life using this growth method. Once a policy which generates

growths that meet some desired score threshold is found, the growths could be mass-produced in much higher quantities than the individual creation of chips which only have the final design.

The next step for this project would be to experiment with different optimization algorithms, such as introducing adaptivity, as well as ways of speeding up training. In particular, determining the minimum resolution needed is an easy way of reducing the time needed per epoch. Changing the target objective to one which is less sensitive on the spatial distribution of the nanoparticles and more depending on global growth properties would also help, as this would reduce the variance in the score for a single policy. An example of an alteration on the target objective is shown in A.

3.5 Conclusion

In this final project we introduced a pipeline to produce policies which generate growths to fulfil some objective. The difficulties in learning policies were explored, which will be useful for improving the pipeline to produce better policies. The pipeline was also implemented to allow the growth simulation, network, and physical solver to be easily exchanged for different ones. Looking at other realistic growth algorithms or non-light target objectives is also an interesting direction to take the project.

Thesis Conclusion

This thesis presented three projects and my contributions to each. The first compared a popular optimization algorithm in the form of Gradient Descent (GD) to a lesser used non-gradient base algorithm Monte Carlo (MC). An analytical correspondence between these two algorithms was proved, with numerical simulations used to show the correspondence. These results were published in [18].

Next, the non-gradient optimization algorithm MC was altered to introduce adaptiveness, in the form of aMC. This allowed it to compete with the popular adaptive GD algorithm, Adam. The performance of aMC was compared to Adam and vanilla GD on a regression task, with aMC performing similar to Adam. The algorithms were further tested in different learning situations, such as very deep networks, demonstrating cases where aMC outperforms Adam on its own. These results were published in [32].

Finally, a pipeline for the design of interesting structures using a semi-realistic growth simulation was introduced. Examples of this pipeline was shown, including a hard photonic chip target objective. Further results were shown giving examples on how the different components of the pipeline can be altered to adapt it to the problem at hand, such as implementing a controllable attractive force in the growth simulation. The pipeline was then discussed in the context of how it would be used to replicate

the results in real life. These results have not been published yet, as more work is planned to show the pipeline's ability to look at different systems and physical solvers.

References

[1] A. Y. Piggott, J. Lu, K. G. Lagoudakis, J. Petykiewicz, T. M. Babinec, and J. Vučković, "Inverse design and demonstration of a compact and broadband on-chip wavelength demultiplexer," *Nature Photonics*, vol. 9, pp. 374–377, Jun 2015.

[2] T. B. Brown, B. Mann, N. Ryder, M. Subbiah, J. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell, S. Agarwal, A. Herbert-Voss, G. Krueger, T. Henighan, R. Child, A. Ramesh, D. M. Ziegler, J. Wu, C. Winter, C. Hesse, M. Chen, E. Sigler, M. Litwin, S. Gray, B. Chess, J. Clark, C. Berner, S. McCandlish, A. Radford, I. Sutskever, and D. Amodei, "Language models are few-shot learners," 2020.

[3] OpenAI, :, J. Achiam, S. Adler, S. Agarwal, L. Ahmad, I. Akkaya, F. L. Aleman, D. Almeida, J. Altenschmidt, S. Altman, S. Anadkat, R. Avila, I. Babuschkin, S. Balaji, V. Balcom, P. Baltescu, H. Bao, M. Bavarian, J. Belgum, I. Bello, J. Berdine, G. Bernadett-Shapiro, C. Berner, L. Bogdonoff, O. Boiko, M. Boyd, A.-L. Brakman, G. Brockman, T. Brooks, M. Brundage, K. Button, T. Cai, R. Campbell, A. Cann, B. Carey, C. Carlson, R. Carmichael, B. Chan, C. Chang, F. Chantzis, D. Chen, S. Chen, R. Chen, J. Chen, M. Chen, B. Chess, C. Cho, C. Chu, H. W. Chung, D. Cummings, J. Currier, Y. Dai, C. Decareaux, T. Degry, N. Deutsch, D. Deville, A. Dhar, D. Dohan, S. Dowling, S. Dunning, A. Ecoffet, A. Eleti, T. Eloundou, D. Farhi, L. Fedus, N. Felix, S. P. Fishman, J. Forte, I. Fulford, L. Gao, E. Georges, C. Gibson, V. Goel, T. Gogineni, G. Goh, R. Gontijo-Lopes, J. Gordon, M. Grafstein, S. Gray, R. Greene, J. Gross, S. S. Gu, Y. Guo, C. Hallacy, J. Han, J. Harris, Y. He, M. Heaton, J. Heidecke, C. Hesse, A. Hickey, W. Hickey, P. Hoeschele, B. Houghton, K. Hsu, S. Hu, X. Hu, J. Huizinga, S. Jain, S. Jain, J. Jang, A. Jiang, R. Jiang, H. Jin, D. Jin, S. Jomoto, B. Jonn, H. Jun, T. Kaftan, Łukasz Kaiser, A. Kamali, I. Kanitscheider, N. S. Keskar, T. Khan, L. Kilpatrick, J. W. Kim, C. Kim, Y. Kim, H. Kirchner, J. Kiros, M. Knight, D. Kokotajlo, Łukasz Kondraciuk, A. Kondrich, A. Konstantinidis, K. Kosic, G. Krueger, V. Kuo, M. Lampe, I. Lan, T. Lee, J. Leike, J. Leung, D. Levy, C. M. Li, R. Lim, M. Lin, S. Lin, M. Litwin, T. Lopez, R. Lowe, P. Lue, A. Makanju, K. Malfacini, S. Manning, T. Markov, Y. Markovski, B. Martin, K. Mayer, A. Mayne, B. McGrew, S. M. McKinney, C. McLeavey, P. McMillan, J. McNeil, D. Medina, A. Mehta, J. Menick, L. Metz, A. Mishchenko, P. Mishkin, V. Monaco, E. Morikawa, D. Mossing, T. Mu, M. Murati, O. Murk, D. Mély, A. Nair, R. Nakano, R. Nayak, A. Neelakantan, R. Ngo, H. Noh, L. Ouyang, C. O'Keefe, J. Pachocki, A. Paino, J. Palermo, A. Pantuliano, G. Parascandolo, J. Parish, E. Parparita, A. Passos, M. Pavlov, A. Peng, A. Perelman, F. de Avila

Belbute Peres, M. Petrov, H. P. de Oliveira Pinto, Michael, Pokorny, M. Pokrass, V. Pong, T. Powell, A. Power, B. Power, E. Proehl, R. Puri, A. Radford, J. Rae, A. Ramesh, C. Raymond, F. Real, K. Rimbach, C. Ross, B. Rotsted, H. Roussez, N. Ryder, M. Saltarelli, T. Sanders, S. Santurkar, G. Sastry, H. Schmidt, D. Schnurr, J. Schulman, D. Selsam, K. Sheppard, T. Sherbakov, J. Shieh, S. Shoker, P. Shyam, S. Sidor, E. Sigler, M. Simens, J. Sitkin, K. Slama, I. Sohl, B. Sokolowsky, Y. Song, N. Staudacher, F. P. Such, N. Summers, I. Sutskever, J. Tang, N. Tezak, M. Thompson, P. Tillet, A. Tootoonchian, E. Tseng, P. Tuggle, N. Turley, J. Tworek, J. F. C. Uribe, A. Vallone, A. Vijayvergiya, C. Voss, C. Wainwright, J. J. Wang, A. Wang, B. Wang, J. Ward, J. Wei, C. Weinmann, A. Welihinda, P. Welinder, J. Weng, L. Weng, M. Wiethoff, D. Willner, C. Winter, S. Wolrich, H. Wong, L. Workman, S. Wu, J. Wu, M. Wu, K. Xiao, T. Xu, S. Yoo, K. Yu, Q. Yuan, W. Zaremba, R. Zellers, C. Zhang, M. Zhang, S. Zhao, T. Zheng, J. Zhuang, W. Zhuk, and B. Zoph, "Gpt-4 technical report," 2023.

[4] R. Rombach, A. Blattmann, D. Lorenz, P. Esser, and B. Ommer, "High-resolution image synthesis with latent diffusion models," 2022.

[5] A. Ramesh, M. Pavlov, G. Goh, S. Gray, C. Voss, A. Radford, M. Chen, and I. Sutskever, "Zero-shot text-to-image generation," 2021.

[6] W. Sun, Y. Zheng, K. Yang, Q. Zhang, A. A. Shah, Z. Wu, Y. Sun, L. Feng, D. Chen, Z. Xiao, S. Lu, Y. Li, and K. Sun, "Machine learning-assisted molecular design and efficiency prediction for high-performance organic photovoltaic materials," *Science advances*, vol. 5, no. 11, pp. eaay4275–eaay4275, 2019.

[7] H. Sahu, F. Yang, X. Ye, J. Ma, W. Fang, and H. Ma, "Designing promising molecules for organic solar cells via machine learning assisted virtual screening," *Journal of materials chemistry. A, Materials for energy and sustainability*, vol. 7, no. 29, pp. 17480–17488, 2019.

[8] Y. Liu, T. Zhao, W. Ju, and S. Shi, "Materials discovery and design using machine learning," *Journal of Materiomics*, vol. 3, no. 3, pp. 159–177, 2017.

[9] W. Xia, M. Sakurai, B. Balasubramanian, T. Liao, R. Wang, C. Zhang, H. Sun, K.-M. Ho, J. R. Chelikowsky, D. J. Sellmyer, and C.-Z. Wang, "Accelerating the discovery of novel magnetic materials using machine learning-guided adaptive feedback," *Proceedings of the National Academy of Sciences - PNAS*, vol. 119, no. 47, pp. e2204485119–e2204485119, 2022.

[10] H. Jin, H. Zhang, J. Li, T. Wang, L. Wan, H. Guo, and Y. Wei, "Discovery of novel two-dimensional photovoltaic materials accelerated by machine learning," *The journal of physical chemistry letters*, vol. 11, no. 8, pp. 3075–3081, 2020.

[11] K. Ryczko, S. J. Wetzel, R. G. Melko, and I. Tamblyn, "Toward orbital-free density functional theory with small data sets and deep learning," *Journal of Chemical Theory and Computation*, vol. 18, p. 1122–1128, Jan. 2022.

[12] L. Sun, H. Gao, S. Pan, and J.-X. Wang, "Surrogate modeling for fluid flows based on physics-constrained deep learning without simulation data," *Computer methods in applied mechanics and engineering*, vol. 361, pp. 112732–, 2020.

[13] I. Goodfellow, Y. Bengio, and A. Courville, *Deep Learning*. MIT Press, 2016. `http://www.deeplearningbook.org`.

[14] D. E. Rumelhart, G. E. Hinton, and R. J. Williams, "Learning representations by back-propagating errors," *Nature (London)*, vol. 323, no. 6088, pp. 533–536, 1986.

[15] N. Thomas, P. Sreekeerthi, and P. Swaminathan, "Combined experimental and simulation study of self-assembly of colloidal gold nanoparticles on silanized glass," *Physical chemistry chemical physics : PCCP*, vol. 24, no. 4, pp. 2525–2535, 2022.

[16] B. Ranguelov and C. Nanev, "2d monte carlo simulation of cocrystal formation using patchy particles," *Crystals (Basel)*, vol. 12, no. 10, pp. 1457–, 2022.

[17] S. Whitelam and P. L. Geissler, "Avoiding unphysical kinetic traps in monte carlo simulations of strongly attractive particles," *The Journal of chemical physics*, vol. 127, no. 15, pp. 154101–154101–19, 2007.

[18] S. Whitelam, V. Selin, S.-W. Park, and I. Tamblyn, "Correspondence between neuroevolution and gradient descent," *Nature Communications*, vol. 12, p. 6317, 2021.

[19] 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*, pp. 8024–8035, Curran Associates, Inc., 2019.

[20] R. Pascanu, T. Mikolov, and Y. Bengio, "On the difficulty of training recurrent neural networks," 2013.

[21] X. Glorot and Y. Bengio, "Understanding the difficulty of training deep feedforward neural networks," in *Proceedings of the Thirteenth International Conference on Artificial Intelligence and Statistics* (Y. W. Teh and M. Titterington, eds.), vol. 9 of *Proceedings of Machine Learning Research*, (Chia Laguna Resort, Sardinia, Italy), pp. 249–256, PMLR, 2010.

[22] B. Hanin, “Which neural net architectures give rise to exploding and vanishing gradients?,” 2018.

[23] K. He, X. Zhang, S. Ren, and J. Sun, “Deep residual learning for image recognition,” 2015.

[24] A. E. Orhan and X. Pitkow, “Skip connections eliminate singularities,” 2018.

[25] H. Li, Z. Xu, G. Taylor, C. Studer, and T. Goldstein, “Visualizing the loss landscape of neural nets,” 2018.

[26] I. J. Goodfellow, O. Vinyals, and A. M. Saxe, “Qualitatively characterizing neural network optimization problems,” 2015.

[27] D. J. Im, M. Tao, and K. Branson, “An empirical analysis of the optimization of deep network loss surfaces,” 2017.

[28] L. Liu, H. Jiang, P. He, W. Chen, X. Liu, J. Gao, and J. Han, “On the variance of the adaptive learning rate and beyond,” *CoRR*, vol. abs/1908.03265, 2019.

[29] D. P. Kingma and J. Ba, “Adam: A method for stochastic optimization,” 2017.

[30] R. M. Schmidt, F. Schneider, and P. Hennig, “Descending through a crowded valley - benchmarking deep learning optimizers,” in *Proceedings of the 38th International Conference on Machine Learning* (M. Meila and T. Zhang, eds.), vol. 139 of *Proceedings of Machine Learning Research*, pp. 9367–9376, PMLR, 18–24 Jul 2021.

[31] S. Sun, Z. Cao, H. Zhu, and J. Zhao, “A survey of optimization methods from a machine learning perspective,” *IEEE Transactions on Cybernetics*, vol. 50, no. 8, pp. 3668–3681, 2020.

[32] S. Whitelam, V. Selin, I. Benlolo, C. Casert, and I. Tamblyn, “Training neural networks using metropolis monte carlo and an adaptive variant,” 2022.

[33] J. L. Ba, J. R. Kiros, and G. E. Hinton, “Layer normalization,” 2016.

[34] Y. Lecun, L. Bottou, Y. Bengio, and P. Haffner, “Gradient-based learning applied to document recognition,” *Proceedings of the IEEE*, vol. 86, no. 11, pp. 2278–2324, 1998.

[35] J. Wang, Y. Wang, and Y. Chen, “Inverse design of materials by machine learning,” *Materials*, vol. 15, no. 5, pp. 1811–, 2022.

[36] H. Choubisa, P. Todorović, J. M. Pina, D. H. Parmar, Z. Li, O. Voznyy, I. Tamblyn, and E. H. Sargent, “Interpretable discovery of semiconductors with machine learning,” *npj computational materials*, vol. 9, no. 1, pp. 117–11, 2023.

[37] B. Zheng, J. Yang, B. Liang, and J.-c. Cheng, "Inverse design of acoustic metamaterials based on machine learning using a gauss–bayesian model," *Journal of applied physics*, vol. 128, no. 13, pp. 134902–, 2020.

[38] S. Sarkar, A. Ji, Z. Jermain, R. Lipton, M. Brongersma, K. Dayal, and H. Y. Noh, "Physics-informed machine learning for inverse design of optical metamaterials," *Advanced photonics research*, vol. 4, no. 12, 2023.

[39] X. Shi, T. Qiu, J. Wang, X. Zhao, and S. Qu, "Metasurface inverse design using machine learning approaches," *Journal of physics. D, Applied physics*, vol. 53, no. 27, pp. 275105–, 2020.

[40] S. Li and A. S. Barnard, "Inverse design of nanoparticles using multi-target machine learning," *Advanced theory and simulations*, vol. 5, no. 2, pp. 2100414–n/a, 2022.

[41] B. Sanchez-Lengeling and A. Aspuru-Guzik, "Inverse molecular design using machine learning: Generative models for matter engineering," *Science (American Association for the Advancement of Science)*, vol. 361, no. 6400, pp. 360–365, 2018.

[42] Y. Long, J. Ren, Y. Li, and H. Chen, "Inverse design of photonic topological state via machine learning," *Applied physics letters*, vol. 114, no. 18, pp. 181105–, 2019.

[43] Z. Liu, D. Zhu, L. Raju, and W. Cai, "Tackling photonic inverse design with machine learning," *Advanced science*, vol. 8, no. 5, pp. 2002923–n/a, 2021.

[44] H. Ahmed, Z. Xiaoping, H. Bello, and N. Iqbal, "Inverse design of multiparameter antenna using hybrid machine learning-driven training dataset," *Microwave and optical technology letters*, vol. 66, no. 1, 2024.

[45] E. Rabani, D. R. Reichman, P. L. Geissler, and L. E. Brus, "Drying-mediated self-assembly of nanoparticles," *Nature*, vol. 426, pp. 271–274, Nov 2003.

[46] A. Stannard, C. P. Martin, E. Pauliac-Vaujour, P. Moriarty, and U. Thiele, "Dual-scale pattern formation in nanoparticle assemblies," *The Journal of Physical Chemistry C*, vol. 112, pp. 15195–15203, Oct 2008.

[47] S. K. Lam, A. Pitrou, and S. Seibert, "Numba: a llvm-based python jit compiler," in *Proceedings of the Second Workshop on the LLVM Compiler Infrastructure in HPC*, LLVM '15, (New York, NY, USA), Association for Computing Machinery, 2015.

[48] J. B. Schneider, *Understanding the Finite-Difference Time-Domain Method.* 2010. `www.eecs.wsu.edu/~schneidj/ufdtd`.

[49] L. S. Inc., "Fdtd solution, a commercial professional software."

[50] A. F. Oskooi, D. Roundy, M. Ibanescu, P. Bermel, J. Joannopoulos, and S. G. Johnson, "Meep: A flexible free-software package for electromagnetic simulations by the fdtd method," *Computer Physics Communications*, vol. 181, no. 3, pp. 687–702, 2010.

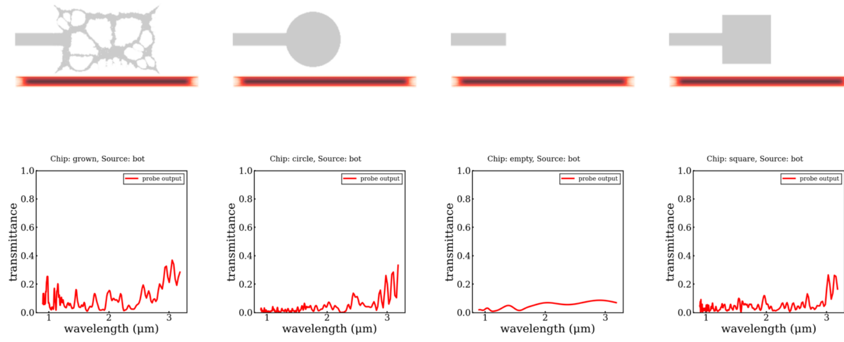
[51] J.-P. Berenger, "A perfectly matched layer for the absorption of electromagnetic waves," *Journal of Computational Physics*, vol. 114, no. 2, pp. 185–200, 1994.

[52] D.-A. Clevert, T. Unterthiner, and S. Hochreiter, "Fast and accurate deep network learning by exponential linear units (elus)," 2016.

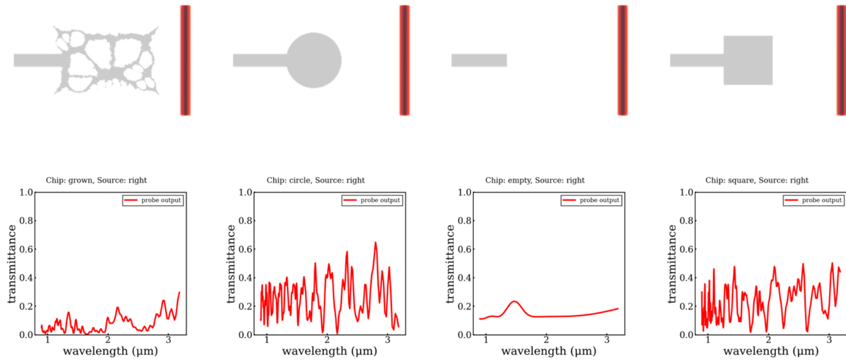
Appendices

A Antenna Growth Objective

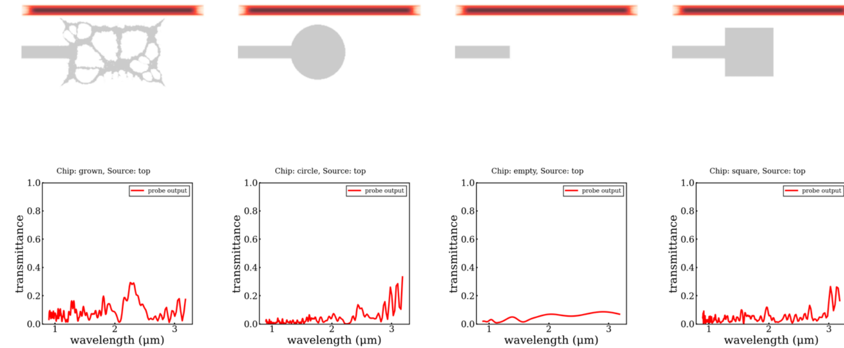
An alternate performance objective briefly tested is a type of antenna that seeks to maximize the captured incoming light.



(a) Antenna objective with top source light.



(b) Antenna objective with right source light.



(c) Antenna objective with top source light.

Figure A.1: Antenna objective with different source light directions. The initial light is shown, which was then propagated towards the chip. The transmission spectra is shown under the corresponding simulation.

Figure A.1 showns an example of this antenna objective. A plane wave is sent towards the chip, and then the transmission that exits through the prob is measured. The objective is to maximize the capture light. In this case, we want the light to be transmitted from any direction, but the objective could be to only capture light from a single direction.

B Code

Here I have listed the most important code I developed for these projects.

B.1 MC-GD correspondence.

Code for a simple PyTorch FCNN.

```
#Import
import torch.nn as nn

#Define neural network
#Takes the depth, width, input size and output size as arguments
class Net(nn.Module):
    def __init__(self,depth,width,input_size,output_size):
        super(Net, self).__init__()

        self.layers = nn.ModuleList([nn.Linear(input_size,width)]+[nn.Linear(width,width) for i in rang

        self.output = nn.Linear(width,output_size) #Output layer: width -> output_size

        self.act = nn.Tanh()


    def forward(self, x):
        #Forward pass through all hidden layers with relu activation
        for i, layer in enumerate(self.layers):
          x = self.act(layer(x))

        x = self.output(x) #Output layer
        return x
```

```
#####
# Main training function for clipped gradient descent
#####
# model: neural network to be trained
# data: dictionary contain data for training in the form: data['x'], data['y'], has shape (n,1) for n d
# n_epochs: number of epochs of training
# learning_rate: learning rate
# limit: loss at which to stop training; None means train for all epochs
# skip: number of epochs between data output
#####
# returns a dictionary of all tracked values
# contains 'epoch', 'loss', 'params', 'time'
```

```
#####
def train_clipped_GD(model, data, n_epochs=100, learning_rate=0.001,limit=None,skip=100):
  #-----------------
  #Setup Training Data
  #-----------------
  x_tensor = torch.from_numpy(data['x']).float()
  y_tensor = torch.from_numpy(data['y']).float()

  #MSE loss function
  loss_func = nn.MSELoss()

  #Set up lists for plotting
  losses = []
  parameters = []
  epochs = []

  t0 = time.time()
  #-----------------
  #Calculate initial loss
  #-----------------
  prediction = model(x_tensor)
  old_loss = loss_func(prediction,y_tensor).item()
  print('Initial_loss: %g'%(old_loss))



  #-----------------
  #Save initial parameters
  #-----------------
  current_parameters = np.array([])
  for name,p in model.named_parameters():
    current_parameters = np.concatenate((current_parameters,p.detach().clone().numpy().reshape(-1)))
  parameters.append(current_parameters)
  #Save initial loss
  losses.append(old_loss)
  epochs.append(0)
  #-----------------
  #Train for n_epochs or until loss < limit
  #-----------------
  for epoch in range(1,n_epochs+1):
    #Calculate loss
    prediction = model(x_tensor)
    loss = loss_func(prediction,y_tensor)
    loss_item = loss.item()
```

```
    #Gradient descent
    loss.backward()

    with torch.no_grad():
      #Calculate |grad U|
      W_grad_squared_sum = 0
      for name, W in model.named_parameters():
        W_grad_squared_sum += torch.sum(torch.square(W.grad))
      W_grad_squared_sum = torch.sqrt(W_grad_squared_sum)

      #Perform clipped gradient descent including |grad U|
      for name, W in model.named_parameters():
        W -= 1/(W_grad_squared_sum) * learning_rate * W.grad
        W.grad.zero_()

    if epoch%skip == 0:
      current_parameters = np.array([])
      for name,p in model.named_parameters():
        current_parameters = np.concatenate((current_parameters,p.detach().clone().numpy().reshape(-1))
      parameters.append(current_parameters)
      losses.append(loss_item)
      epochs.append(epoch)
    #-----------------
    #Stop training if limit is reached
    #-----------------
    if limit is not None and loss_item < limit:
      break

    #Print progress
    if epoch%(n_epochs//10)==0:
      #clear_output(wait=True)
      print("Progress=%g%%,␣Loss=%g"%(epoch/n_epochs*100,loss_item))

  t1 = time.time()
  training_time = t1-t0
  print('Final␣loss:%g,␣Training␣time:%gs'%(loss_item,training_time))
  vals = {'loss':np.array(losses),'params':np.array(parameters),
              'time':np.array([training_time]),'epoch':np.array(epochs)}
  return vals
```

Training function to use MC optimizer.

```
"""# MC training function"""
#####
# Main training function for gradient descent using a zero-temperature Metropolis Monte Carlo (MC)
#              algorithm, a simple form of neuroevolution
# Trains a given network on given data using either vanilla MC
#####
# model: neural network to be trained
# data: dictionary contain data for training in the form: data['x'], data['y'],
#           has shape (n,1) for n data points
# n_epochs: number of epochs of training
# elr: effective learning rate, the parameter "lambda" described in the paper
# limit: loss at which to stop training; None means train for all epochs
# skip: number of epochs between data output
#####
# returns a dictionary of all tracked values
# contains 'epoch', 'loss', 'params', 'time'
#####
def train_MC(model, data, n_epochs=100,elr=0.1,limit=None,skip=100):
  #Calculate basic step size sigma as a function of learning rate, per Eq. (54) of the paper
  sigma = elr*np.sqrt(2*np.pi)
  print("Sigma:%g"%(sigma))
  #-----------------
  #Setup Training Data
  #-----------------
  x_tensor = torch.from_numpy(data['x']).float()
  y_tensor = torch.from_numpy(data['y']).float()

  #MSE loss function
  loss_func = nn.MSELoss()

  #Set up lists for plotting
  losses = []
  parameters = []
  epochs = []

  t0 = time.time()
  torch.set_grad_enabled(False)

  #-----------------
  #Calculate initial loss
  #-----------------
  prediction = model(x_tensor)
  loss = loss_func(prediction,y_tensor)
  old_loss = loss.item()
```

```
45  print('Initial␣loss:␣%g'%(old_loss))
46
47  #-----------------
48  #Save initial parameters
49  #-----------------
50  #Save zeroth epoch
51  epochs.append(0)
52  #Save Loss
53  losses.append(old_loss)
54  #Save Parameters
55  current_parameters = np.array([])
56  for name,p in model.named_parameters():
57    current_parameters = np.concatenate((current_parameters,p.detach().clone().numpy().reshape(-1)))
58  parameters.append(current_parameters)
59
60
61  #-----------------
62  #Train for n_epochs or until loss < limit
63  #-----------------
64  for epoch in range(1,n_epochs+1):
65    #Old parameters
66    old_par = copy.deepcopy(model.state_dict())
67
68
69    #Update model parameters with Gaussian
70    for name,p in model.named_parameters():
71      p += torch.empty(p.size()).normal_(mean=0,std=sigma)
72
73    #Calculate loss with updated parameters
74    prediction = model(x_tensor)
75    loss = loss_func(prediction,y_tensor)
76    new_loss = loss.item()
77
78    #-----------------
79    #Accept or reject updated weights
80    #-----------------
81    #Accept if new loss is smaller
82    if new_loss <= old_loss:
83      old_loss = new_loss
84
85    else:
86      #Keep old parameters
87      model.load_state_dict(old_par)
88      new_loss = old_loss
```

```

    #Save every skip epochs
    if epoch%skip == 0:
      current_parameters = np.array([])
      for name,p in model.named_parameters():
        current_parameters = np.concatenate((current_parameters,p.detach().clone().numpy().reshape(-1))
      parameters.append(current_parameters)
      losses.append(new_loss)
      epochs.append(epoch)

    #-----------------
    #Stop training if limit is reached
    #-----------------
    if limit is not None and new_loss < limit:
      break

    #Print progress
    if epoch%(n_epochs//10)==0:
      print("Progress=%g%%,␣Loss=%g"%(epoch/n_epochs*100,new_loss))

  t1 = time.time()
  training_time = t1-t0
  print('Final␣loss:%g,␣Training␣time:%gs'%(new_loss,training_time))
  vals = {'loss':np.array(losses),'params':np.array(parameters),
              'time':np.array([training_time]),'epoch':np.array(epochs)}
  return vals
```

More code can be found in the paper, which show examples of how to use the training functions to train a NN.

B.2 aMC.

NN code for aMC optimizer. The λ quantity for the signal norm is calculated in the forward pass.

```
import torch.nn as nn
import torch

#Define neural network
#Takes the depth, width, input size and output size and activation function as arguments
class Net(nn.Module):
    def __init__(self,depth=1,width=4,input_size=1,output_size=1):
        super(Net, self).__init__()

         #N hidden layers, N = depth-1: width -> width
```

```
        self.layers = nn.ModuleList([nn.Linear(input_size,width)]+
                                                      [nn.Linear(width,width) for i in range(depth-2)]
                                                      [nn.Linear(width,10)])

        #Output layer: width -> output_size
        self.output = nn.Linear(10,output_size,bias=False)

        self.act = nn.Tanh()


    def forward(self, x):
        #List of lambdas
        lambdas = []
        #Forward pass through all hidden layers with tanh activation
        for i, layer in enumerate(self.layers):
          #Calculate lambda for current layer
          #x has shape (N_data,n_nodes)
          lambdas.append(torch.pow(torch.sum(torch.square(x))/x.shape[0],-1/2).item())
          #calculate layer
          x = self.act(layer(x))

        #final layer lambda
        lambdas.append(torch.pow(torch.sum(torch.square(x))/x.shape[0],-1/2).item())
        x = self.output(x) #Output layer; has no activation
        #return output and lambdas
        return x, lambdas

if __name__ == '__main__':
  import sys
  import torch
  depth = int(sys.argv[1])
  width = int(sys.argv[2])

  print('Generating␣initial␣parameters␣for␣model␣with␣depth:␣%g,␣width:␣%g'%(depth,width))
  model = Net(depth,width)
  savefile = 'initial_params_%g_%g'%(depth,width)
  torch.save(model.state_dict(), savefile)
  print('Done,␣saved␣as:␣'+savefile)
```

aMC optimizer code.

```
import torch
from torch.optim.optimizer import Optimizer
import numpy as np
```

```
4
5  ######
6  # Adaptive MC optimizer
7  # Takes model parameters as well as the following 3 hyperparameters:
8  # sigma = Initial value of the standard deviation of the move-proposal distribution
9  # epsilon = Rate at which parameters of the move-proposal distribution are modified
10 # N_scale = Number of consecutive rejected moves before a parameter's move-proposal
11 #                   distribution is reset to initial values
12 # track_acc = Bool, whether to track the accuracy as a function of loss
13 # single_batch = if using a single batch per epoch, speed up by storing the previous loss
14 ######
15 class aMC(Optimizer):
16   ######
17   #Initialize optimizer
18   ######
19   def __init__(self, model, sigma, epsilon, N_scale,track_acc=False,single_batch=False):
20
21     params = model.parameters()
22
23     #Intialize optimizer
24     defaults = dict(sigma=sigma, epsilon=epsilon, N_scale=float(N_scale))
25     super().__init__(params, defaults)
26
27     state = self.state[0]
28
29     #Initalize mean list, sigma, and, consecutive rejection counter
30     state['is_bias'] = ['bias' in named_params[0] for named_params in model.named_parameters()]
31     state['mean_i'] = []
32     state['sigma'] = sigma
33     state['N_CR'] = 0
34     #Initialize prev loss
35     state['prev_loss'] = None
36     state['prev_lambdas'] = None
37     state['track_acc'] = track_acc
38     state['single_batch'] = single_batch
39
40     if track_acc:
41       state['bins'] = np.logspace(0,-20,100)
42       state['acc_counts'] = np.zeros(np.shape(state['bins']),dtype=int)
43       state['rej_counts'] = np.zeros(np.shape(state['bins']),dtype=int)
44
45     params = self.param_groups[0]['params']
46     state['limit_sigma'] = True
47     for i, param in enumerate(params):
```

```
48          if param.dtype == torch.float64:
49            state['limit_sigma'] = False
50          #Initialize mean array on same device as parameters
51          #get_device() returns -1 if param is on cpu; returns index >= 0 for corresponding GPU.
52          device = param.get_device()
53          if device >= 0:
54            state['mean_i'].append(torch.zeros(param.shape,dtype=param.dtype,device=device))
55          else:
56            state['mean_i'].append(torch.zeros(param.shape,dtype=param.dtype,device='cpu'))
57
58      ######
59      # Optimizer step function, needs a closure to calculate the loss in the form:
60      # @torch.no_grad()
61      # def closure():
62      #   pred, lambdas = model(x_batch_tensor)
63      #   loss = loss_func(pred,y_batch_tensor)
64      #   return loss.item(), lambdas
65      #
66      #model needs to return lambdas, see model_FanIn.py for an example of how to calculate this factor
67      ######
68      @torch.no_grad()
69      def step(self, closure):
70        #Load state and params
71        group = self.param_groups[0]
72        params = group['params']
73        state = self.state[0]
74
75        #Calculate initial loss
76        if not state['single_batch'] or state['prev_loss'] == None:
77          state['prev_loss'], state['prev_lambdas'] = closure()
78        #Get the previous loss
79        prev_loss = state['prev_loss']
80        prev_lambdas = state['prev_lambdas']
81        ##############
82        #Step 1.0 and 2.0
83        ##############
84        #Initial parameters
85        initial_params = []
86        #Perturb parameters
87        perturbations = []
88        j = 0
89        for i, param in enumerate(params):
90          #Save copy of parameters before perturbation
91          initial_params.append(param.clone().detach())
```

```
92
93              #Step 2.0 sigma of perturbations for weights are scaled based on lambdas
94              if state['is_bias'][i]:
95                  perturbation = torch.normal(state['mean_i'][i],state['sigma'])
96              else:
97                  perturbation = torch.normal(state['mean_i'][i],prev_lambdas[j]*state['sigma'])
98                  j += 1
99              param += perturbation
100             if group['epsilon'] != 0:
101                 perturbations.append(perturbation)
102
103         #Calculate new loss
104         new_loss, new_lambdas = closure()
105         #############
106         #Step 3.0
107         ##############
108         #Accept move if new loss is less than previous loss
109         if new_loss <= prev_loss:
110             if state['track_acc']:
111                 bin_i = min(np.where(prev_loss > state['bins'])[0])
112                 state['acc_counts'][bin_i] += 1
113             #Shift mean based on perturbation
114             if group['epsilon'] != 0:
115                 for i, param in enumerate(params):
116                     state['mean_i'][i] += group['epsilon'] * (perturbations[i] - state['mean_i'][i])
117             #Reset consecutive rejection counter
118             state['N_CR'] = 0
119             state['prev_loss'] = new_loss
120             state['prev_lambdas'] = new_lambdas
121             return new_loss
122
123         #############
124         #Step 4.0
125         ##############
126         else:
127             if state['track_acc']:
128                 bin_i = min(np.where(prev_loss > state['bins'])[0])
129                 state['rej_counts'][bin_i] += 1
130             #Increment consecutive rejection counter
131             state['N_CR'] += 1
132             for i, param in enumerate(params):
133                 #Reset parameter to initial parameters
134                 param[:] = initial_params[i]
135                 #Reset mean and scale sigma
```

```
        if state['N_CR'] >= group['N_scale']:
          state['mean_i'][i].zero_()

      #Reset rejection counter if a move was rejected; scale sigma
      if state['N_CR'] >= group['N_scale']:
        if state['limit_sigma']:
          state['sigma'] = max(state['sigma']*0.95,1e-5)
        else:
          state['sigma'] *= 0.95
        state['N_CR'] = 0
    #Return previous loss
    return prev_loss
```

Main training function.

```
import sys, copy, time
import numpy as np
import torch
import torch.nn as nn

#Main training function
#Needs:
#model: pytorch neural network
#data: dictionnary with x - y data as keys
#n_epochs: maximum number of epochs to train for
#limit: minimum loss, stops training when reached
#skip: number of epochs to skip before saving statistics (loss, parameters, prediction, mag_grad ...)
#opti: optimizer to use accepts "SGD", "Adam", "AMC" ---> SGD is vanilla GD,
#                      Adam is adam optimizer, AMC is adaptive MC
#opti_param: For SGD and Adam, needs the form: {'lr':0.001},
#                       for AMC needs: {'N_memory':10,'N_scale':1,'N_reset':500,
#                                              'N_steps':1,'sigma':0.001,'f_chosen':0.5}
#trac_X: Whether to track the given statistic. can track Loss, Parameters,
#                                              Prediction, Magnitude of Gradient
#### Note: Time and epochs are always tracked
#time_limit: Time limit in seconds
#Returns a dictionnary of all the statistics: tracked_val
#### Can be saved using: np.savez_compressed(save_file,**tracked_val)
def train_model(model, data, datatype, n_epochs=1000,loss_limit=None,opti='SGD',
                            opti_param = {'lr':0.001},track_loss=True,track_param=False,track_pred=Fals
                            track_mag_grad=False,time_points=100,time_limit=60,device='cpu'):

  #-----------------
  #Setup Training Data
```

```
    #-----------------
    x_array = data['x']
    y_array = data['y']
    length = np.shape(x_array)[0]
    if datatype == 'double':
      x_tensor = torch.from_numpy(data['x']).double()
      y_tensor = torch.from_numpy(data['y']).double()
    elif datatype == 'float':
      x_tensor = torch.from_numpy(data['x']).float()
      y_tensor = torch.from_numpy(data['y']).float()
    else:
      print('Invalid datatype')

    x_tensor = x_tensor.to(device)
    y_tensor = y_tensor.to(device)

    #MSE loss function and optimizer
    loss_func = nn.MSELoss()
    if opti == 'SGD':
      torch.set_grad_enabled(True)
      optimizer = torch.optim.SGD(model.parameters(), **opti_param)
    elif opti == 'Adam':
      torch.set_grad_enabled(True)
      optimizer = torch.optim.Adam(model.parameters(), **opti_param)
    elif opti == 'AMC':
      #Import AdaptiveMC2 optimizer
      from aMC_FanIn import aMC
      torch.set_grad_enabled(False)
      optimizer = aMC(model, **opti_param)
    else:
      print("Invalid optimizer")
      return

    #Set up lists for plotting
    tracked_val = {}
    tracked_val['epoch'] = []
    tracked_val['time'] = []
    tracked_val['loss'] = []
    tracked_val['pred'] = []

    if track_param:
      tracked_val['param'] = []
    if track_mag_grad:
      tracked_val['mag_grad'] = []
```

```
74
75
76    #-----------------
77    #Calculate initial parameters loss
78    #-----------------
79    prediction, _ = model(x_tensor)
80    old_loss = loss_func(prediction,y_tensor).item()
81    print('Initial_loss:␣%g'%(old_loss))
82
83
84    #-----------------
85    #Save initial parameters
86    #-----------------
87    #Save zeroth epoch
88    tracked_val['epoch'].append(0)
89    tracked_val['time'].append(0)
90    #Save Loss
91    if track_loss:
92      tracked_val['loss'].append(old_loss)
93    #Save Parameters
94    if track_param:
95      current_parameters = np.array([])
96      for name,p in model.named_parameters():
97        current_parameters = np.concatenate((current_parameters,p.detach().clone().numpy().reshape(-1)))
98      tracked_val['param'].append(current_parameters)
99    #Save prediction
100   if track_pred:
101     tracked_val['pred'].append(prediction.detach().cpu().numpy().reshape(-1))
102   if track_mag_grad:
103     torch.set_grad_enabled(True)
104     model.zero_grad()
105     prediction, _ = model(x_tensor)
106     loss = loss_func(prediction,y_tensor)
107     loss.backward()
108     #Calculate special factor
109     W_grad_squared_sum = 0
110     for name, W in model.named_parameters():
111       W_grad_squared_sum += torch.sum(torch.square(W.grad))
112     W_grad_squared_sum = torch.sqrt(W_grad_squared_sum)
113     tracked_val['mag_grad'].append(W_grad_squared_sum.detach().numpy())
114     if opti == 'AMC':
115         torch.set_grad_enabled(False)
116   #-----------------
117   #Train for a given amount of time, maximum of n_epochs epochs,
```

```
118     #stops if a loss limit is reached (if given)
119     #-----------------
120     t0 = time.time()
121     time_diff = time_limit/time_points
122     save_time = time_diff
123 
124     #
125     #time_points_a = np.logspace(0,np.log10(time_limit),time_points) -1
126     #time_i = 0
127 
128     is_finished = False
129     epoch = 0
130     while not is_finished:
131       epoch += 1
132       #Calculate loss with updated parameters
133       if opti != 'AMC':
134         prediction = model(x_tensor)
135         loss = loss_func(prediction,y_tensor)
136         loss_item = loss.item()
137 
138         #Gradient descent
139         optimizer.zero_grad()
140         loss.backward()
141         optimizer.step()
142       elif opti == 'AMC':
143         @torch.no_grad()
144         def closure():
145           pred, lambdas = model(x_tensor)
146           loss = loss_func(pred,y_tensor)
147           return loss.item(), lambdas
148         loss_item = optimizer.step(closure)
149         #prediction, lambdas = model(x_tensor)
150 
151       #Save every time_diff sec
152       t1 = time.time()
153       current_time = t1-t0
154       #if current_time >= time_points_a[time_i]:
155       #  time_i += 1
156         #save_time += time_diff
157       if current_time >= save_time:
158         save_time += time_diff
159       #if epoch%skip == 0:
160         #Save epoch
161         tracked_val['epoch'].append(epoch)
```

```
162        tracked_val['time'].append(current_time)
163        #Save Loss
164        if track_loss:
165          tracked_val['loss'].append(loss_item)
166        #Save Parameters
167        if track_param:
168          current_parameters = np.array([])
169          for name,p in model.named_parameters():
170            current_parameters = np.concatenate((current_parameters,
171                                      p.detach().clone().numpy().reshape(-1)))
172          tracked_val['param'].append(current_parameters)
173        #Save prediction
174        if track_pred:
175          #tracked_pred['pred'].append(prediction.detach().numpy()[unshuffled_indices].reshape(-1))
176          prediction, _ = model(x_tensor)
177          tracked_val['pred'].append(prediction.detach().cpu().numpy().reshape(-1))
178        if track_mag_grad:
179          torch.set_grad_enabled(True)
180          model.zero_grad()
181          prediction, _ = model(x_tensor)
182          loss = loss_func(prediction,y_tensor)
183          loss.backward()
184          #Calculate special factor
185          W_grad_squared_sum = 0
186          for name, W in model.named_parameters():
187            W_grad_squared_sum += torch.sum(torch.square(W.grad))
188          W_grad_squared_sum = torch.sqrt(W_grad_squared_sum)
189          tracked_val['mag_grad'].append(W_grad_squared_sum.detach().numpy())
190          if opti == 'AMC':
191            torch.set_grad_enabled(False)
192        #Print progress
193        if opti == 'AMC':
194          params = list(param for param in model.parameters())
195          print("Progress=%.2f%%, Loss=%s, Time=%.2f min, epoch=%g,
196                      weight_0:%s, sigma=%g"%(current_time/time_limit*100,str(loss_item),
197                      current_time/60,epoch,str(params[0][0].item()),optimizer.state[0]['sigma']))
198        else:
199          print("Progress=%.2f%%, Loss=%g, Time=%.2f min,
200                     epoch=%g"%(current_time/time_limit*100,loss_item,
201                     current_time/60,epoch))
202      #-----------------
203      #Stop training if limit is reached
204      #-----------------
205      if time_limit is not None and current_time > time_limit:
```

```
      print("Reached time limit: %gs"%(current_time))
      is_finished = True

    if loss_limit is not None and loss_item < loss_limit:
      print("Reached loss limit")
      is_finished = True

    if n_epochs is not None and epoch >= n_epochs:
      print("Reached epoch limit")
      is_finished = True

  #Save final result
  final_prediction, _ = model(x_tensor)
  final_loss = loss_func(final_prediction,y_tensor).item()
  print('final_loss: %g'%(final_loss))
  t1 = time.time()
  tracked_val['epoch'].append(epoch+1)
  tracked_val['time'].append(t1-t0)
  tracked_val['loss'].append(final_loss)
  tracked_val['pred'].append(final_prediction.detach().cpu().numpy().reshape(-1))

  training_time = t1-t0
  print('Final loss:%g, Training time:%.2f min'%(final_loss,training_time/60))


  #Turn tracked values into arrays
  tracked_val['epoch'] = np.array(tracked_val['epoch'])
  tracked_val['time'] = np.array(tracked_val['time'])
  tracked_val['loss'] = np.array(tracked_val['loss'])
  tracked_val['pred'] = np.array(tracked_val['pred'])
  if opti == 'AMC' and opti_param['track_acc']:
    tracked_val['acc'] = np.array(optimizer.state[0]['acc_counts'])
    tracked_val['rej'] = np.array(optimizer.state[0]['rej_counts'])
    tracked_val['bins'] = np.array(optimizer.state[0]['bins'])
  if track_param:
    tracked_val['param'] = np.array(tracked_val['param'])
  if track_mag_grad:
    tracked_val['mag_grad'] = np.array(tracked_val['mag_grad'])
  return tracked_val
```

Other supporting functions for plotting and manipulating the results are not shown.

B.3 Growth Pipeline.

First, the code for the Lumerical inverse design test using circles is shown.

```
import lumapi
import numpy as np
import time
import sys

#Base class for actions
class Base_Action:
  def doAction(self):
    return

  def accept(self):
    return

  def reject(self):
    return


##Place a circle
class Place_Circle(Base_Action):
  def __init__(self):
    self.current_circle = None

  def doAction(self):
    global object_index
    sim.groupscope('::model::Prediction␣Shape')
    x = rng.random()*xspan - xspan/2.0
    y = rng.random()*yspan - yspan/2.0
    radius = (rng.random()*0.95+0.05)*max_radius
    name = "circle_%g"%(object_index)
    object_index += 1
    sim.addcircle()
    sim.set('name',name)
    sim.set('x',x)
    sim.set('y',y)
    sim.set('radius',radius)
    sim.set('material',"SiO2␣(Glass)␣-␣Palik")
    print('placing␣circle␣with␣radius␣%g␣at␣x␣=␣%g,␣y␣=␣%g'%(radius,x,y))
    self.current_circle = name

  def reject(self):
    if self.current_circle != None:
```

```
42        sim.groupscope('::model::Prediction␣Shape')
43        sim.select(self.current_circle)
44        print('Rejected,␣deleted␣'+ self.current_circle)
45        sim.delete()
46        self.current_circle = None
47    def accept(self):
48      if self.current_circle != None:
49        print('Accepted,␣kept␣'+ self.current_circle)
50        self.current_circle = None
51
52  ##Remove a shape
53  class Remove_Shape(Base_Action):
54    def __init__(self):
55      self.current_shape = None
56
57    def doAction(self):
58      sim.groupscope('::model::Prediction␣Shape')
59      sim.selectall()
60      n_shapes = sim.getnumber()
61      if n_shapes > 0:
62        index = rng.integers(1, n_shapes+1)
63        shape = sim.get('name',index)
64        sim.select(shape)
65        sim.set('enabled',0)
66        self.current_shape = shape
67        print('Removing␣Shape␣'+shape)
68
69    def accept(self):
70      if self.current_shape != None:
71        sim.groupscope('::model::Prediction␣Shape')
72        sim.select(self.current_shape)
73        sim.delete()
74        print('Accepted,␣deleted␣'+self.current_shape)
75        self.current_shape = None
76
77    def reject(self):
78      if self.current_shape != None:
79        sim.groupscope('::model::Prediction␣Shape')
80        sim.select(self.current_shape)
81        sim.set('enabled',1)
82        print('Rejected,␣added␣back␣'+self.current_shape)
83        self.current_shape = None
84
85
```

```
86  rng = np.random.default_rng(126)
87  n_epochs = int(sys.argv[1])
88  show_gui = int(sys.argv[2]) == 0
89
90  def calculate_loss():
91    sim.switchtolayout()
92    sim.run()
93    sim.groupscope('::model')
94    field_monitor = sim.getresult('field_monitor','E')
95    field_pred = field_monitor['E']
96    field_pred = np.squeeze(field_pred)
97    field_pred_mag = np.linalg.norm(field_pred,axis=-1)
98    loss = np.square(field_pred_mag - field_target_mag)
99    loss_mean = np.mean(loss, axis = 0)
100   loss_mean = np.mean(loss_mean)
101   sim.switchtolayout()
102   return loss_mean
103
104
105
106 ##INITIALIZE SIMULATION
107 #Load solution
108 sim = lumapi.FDTD("Dumbell_Plane.fsp",hide=show_gui)
109 sim.save("Dumbell_Plane_Target.fsp")
110
111 ##RUN SOLUTION
112
113 #Switch to layout mode
114 sim.switchtolayout()
115
116 #Root
117 sim.groupscope("::model")
118 sim.select("FDTD")
119 xspan = sim.get("x␣span")
120 yspan = sim.get("y␣span")
121 max_radius = min(xspan,yspan)/4.0
122 object_index = 0
123
124 #Select Solution
125 sim.select("Solution␣Shape")
126 sim.set('enabled',1)
127 sim.run()
128
129 field_monitor = sim.getresult('field_monitor','E')
```

```
130 field_target = field_monitor['E']
131 field_target = np.squeeze(field_target)
132 field_target_mag = np.linalg.norm(field_target,axis=-1)
133 np.savez("target_field",field_target_mag)
134 sim.save("Dumbell_Plane_Initial.fsp")
135 ##PREDICTION MODE
136 #INITIALIZE ACTIONS
137 
138 actions = []
139 placeCircle = Place_Circle()
140 removeShape = Remove_Shape()
141 
142 actions = [placeCircle, removeShape]
143 
144 #Select solution and disable
145 sim.switchtolayout()
146 sim.select("Solution␣Shape")
147 sim.set('enabled',0)
148 
149 #Select prediction shape
150 sim.groupscope("Prediction␣Shape")
151 sim.deleteall()
152 
153 sim.run()
154 field_monitor = sim.getresult('field_monitor','E')
155 field_empty = field_monitor['E']
156 field_empty = np.squeeze(field_empty)
157 field_empty_mag = np.linalg.norm(field_empty,axis=-1)
158 np.savez("empty_field",field_empty_mag)
159 
160 
161 sim.switchtolayout()
162 
163 sim.groupscope('::model')
164 sim.select("movie_monitor")
165 sim.set('enabled',0)
166 
167 sim.save("Dumbell_Plane_Prediction.fsp")
168 
169 initial_loss = calculate_loss()
170 old_loss = initial_loss
171 
172 
173 #MAIN LEARNING LOOP
```

```
t0 = time.time()
for epoch in range(n_epochs):
  action = actions[rng.integers(0,len(actions))]
  print('\nSelected action ' + str(action.__class__.__name__))
  action.doAction()
  new_loss = calculate_loss()
  print(new_loss, old_loss)
  #Reject
  if new_loss > old_loss:
    action.reject()
  else:
    old_loss = new_loss
    action.accept()
t1 = time.time()
print('\nTotal elapsed time: %.2fs'%(t1-t0))
print('Time per action: %.2fs'%((t1-t0)/n_epochs))


##FINAL RUN
sim.switchtolayout()

sim.save("Dumbell_Plane_Prediction_Finished.fsp")
sim.groupscope('::model')
sim.select("movie_monitor")
sim.set('enabled',1)

sim.run()
field_monitor = sim.getresult('field_monitor','E')

field_pred = field_monitor['E']
field_pred = np.squeeze(field_pred)
field_pred_mag = np.linalg.norm(field_pred,axis=-1)
np.savez("pred_field",field_pred_mag)

#fdtd.close()
input('Waiting for input\n')
sim.close()
```

First the code for the growth pipeline with linear attraction. The version without attraction only requires removing small portions of the code. This first code is the main growth class.

```
#Imports
import numpy as np
import numba as nb
```

```
import math
from numba.experimental import jitclass
from numba import jit
from numba.types import UniTuple
from typing import List
from numba.typed import List as NumbaList

'''
Main Growth class

Args:
dim: dimension of lattice (dim x dim)
n_nano: number of nanoparticles to place
KbT: temperature * Kb
mu: chemical potential
e_nn: nano-nano attraction
e_nl: nano-solvent attraction
e_ll: solvent-solvent attraction -- All other attractions and mu are given in relation to this,
#leave as 1
seed: random seed for rng generator
nano_mob: number of nanoparticle cycles to perform per solvent cycle
nano_size: size of nanoparticles (nano_size x nano_size)

Use:
growth = Growth(args)
growth.initialize_nano
for range(n_epochs):
  growth.step_simulation()
'''
@jitclass
class Growth_NonPeriodic:
    #Type annotation for Numba
    x_dim: int
    x_length: float
    y_dim: int
    n_nano: int
    n_nano_placed: int
    nano_size: int
    fluid: nb.int64[:,:]
    nano: nb.int64[:,:]
    nano_list: List[UniTuple(nb.int64,2)]
    attraction_x: float
    attraction_e: float
    total_energy: float
```

```
48    e_ll: float
49    e_nl: float
50    e_nn: float
51    mu: float
52    nano_mob: int
53    solv_iter: int
54    nano_iter: int
55    KbT: float
56    seed: int
57    norm: float
58    tot_E_norm: float
59
60    def __init__(self, x_dim, y_dim, n_nano, KbT, mu, e_nn, e_nl, e_ll, nano_mob, nano_size, seed):
61        #Keep parameters
62        self.x_dim = x_dim
63        self.y_dim = y_dim
64        self.x_length = 2.8
65        self.n_nano = n_nano
66        self.n_nano_placed = 0
67        self.seed = seed
68        self.nano_size = nano_size
69
70        #Initialize RNG seed
71        np.random.seed(self.seed)
72
73        #Initialize lattices, padded with an extra entry in each direction
74        self.fluid = np.ones((self.x_dim,self.y_dim),dtype=np.int64)
75        self.nano = np.zeros((self.x_dim,self.y_dim),dtype=np.int64)
76        #Initialize list of nanoparticles
77        self.nano_list = NumbaList([(1,2) for x in range(0)])
78        self.attraction_x = 0.5
79        self.attraction_e = 0.04
80        #Initialize prev energy
81        self.total_energy = 0
82
83        #Initialize constants
84        #Liquid - liquid attraction
85        self.e_ll = e_ll
86        #Nano - nano attraction
87        self.e_nn = e_nn*self.e_ll
88        #Nano - liquid attraction
89        self.e_nl = e_nl*self.e_ll
90        #Chemical potential
91        self.mu = mu*self.e_ll
```

```
92
93          #Nano mobility and per step solvent iterations
94          self.nano_mob = nano_mob
95          self.solv_iter = x_dim*y_dim
96          self.nano_iter = 0
97
98          #Boltzmann*Temperature
99          self.KbT = self.e_ll*KbT
100
101         self.norm = 1/math.sqrt(2)
102         self.tot_E_norm = 1/(1+1/math.sqrt(2))
103
104     '''
105     Old function to calculate energy
106     Calculates total energy on entire lattice
107     '''
108     #Function that multiplies all elements with neighbours and sums
109     def sum_neighbour_energy(self,A,B):
110         return np.sum(A*(np.roll(B,1,0) + np.roll(B,-1,0) + np.roll(B,1,1) + np.roll(B,-1,1)))
111
112     #Slow way of calcuating total energy
113     def calculate_total_energy(self,nano,fluid):
114         total_energy = 0
115
116         #Liquid - liquid contribution
117         total_energy -= self.e_ll * self.sum_neighbour_energy(fluid,fluid) / 2
118
119         #Nano - nano contribution
120         total_energy -= self.e_nn * self.sum_neighbour_energy(nano,nano) / 2
121
122         #Nano - liquid contribution
123         total_energy -= self.e_nl * self.sum_neighbour_energy(nano,fluid)
124
125         #Liquid phase contribution
126         total_energy -= self.mu * np.sum(fluid)
127
128         return total_energy
129
130
131
132
133     '''
134     Function that performs a single solvent step
135     Chooses a random lattice site and attempts to change phase
```

```
136     '''
137     def step_fluid(self):
138         #Choose random lattice cell
139         x_i = np.random.randint(1,self.x_dim-1)
140         y_i = np.random.randint(1,self.y_dim-1)
141 
142         #Only proceed if no nano in cell
143         delta_e = 0
144         if self.nano[x_i,y_i] == 0:
145 
146             #Calculate change in energy
147             delta_e = -(1-2*self.fluid[x_i,y_i])*((self.tot_E_norm)*(self.e_ll*
148                                                           (self.fluid[(x_i-1),y_i]+
149                                                            self.fluid[(x_i+1),y_i]+
150                                                            self.fluid[x_i,(y_i-1)]+
151                                                            self.fluid[x_i,(y_i+1)]+
152                                                            (self.norm)*
153                                                            (self.fluid[(x_i-1),(y_i-1)]+
154                                                            self.fluid[(x_i+1),(y_i-1)]+
155                                                            self.fluid[(x_i-1),(y_i+1)]+
156                                                            self.fluid[(x_i+1),(y_i+1)]))
157                                                 +self.e_nl*(self.nano[(x_i-1),y_i]+
158                                                            self.nano[(x_i+1),y_i]+
159                                                            self.nano[x_i,(y_i-1)]+
160                                                            self.nano[x_i,(y_i+1)]+
161                                                            (self.norm)*
162                                                            (self.nano[(x_i-1),(y_i-1)]+
163                                                            self.nano[(x_i+1),(y_i-1)]+
164                                                            self.nano[(x_i-1),(y_i+1)]+
165                                                            self.nano[(x_i+1),(y_i+1)]))
166                                                 +self.mu)
167 
168 
169 
170             #Calculate probability of accepting move
171             Pacc = min(1,np.exp(-delta_e/self.KbT))
172 
173             if np.random.random() <= Pacc:
174                 #Accept move - update total energy
175                 self.total_energy += delta_e
176 
177                 #Change solvent phase
178                 self.fluid[x_i,y_i] = 1-self.fluid[x_i,y_i]
179 
```

```
180            return delta_e
181
182        '''
183        Function that calculates the change in bond energy for a nanoparticle move
184        '''
185        def neigh_de(self,cur_cell,cell,dis):
186            #x_i = cell[0]
187            #y_i = cell[1]
188            x_i_neigh = (cell[0]+dis[0])
189            y_i_neigh = (cell[1]+dis[1])
190
191            #Calculate bond energy change
192            #(1-2*fluid), (1-2*nano)
193            fluid_neigh = self.fluid[x_i_neigh,y_i_neigh]
194            nano_neigh = self.nano[x_i_neigh,y_i_neigh]
195            de = (-cur_cell)*(self.e_ll*(fluid_neigh)+\
196                                  self.e_nl*(nano_neigh))+\
197                 (cur_cell)*(self.e_nn*(nano_neigh)+\
198                                 self.e_nl*(fluid_neigh))
199            return -de
200
201        '''
202        Function that calculates total energy using neighbour bonds
203        '''
204        def calculate_total_energy_neigh(self):
205            total_e = 0
206            diss = [(1,0),(-1,0),(0,1),(0,-1)]
207            nneigh = [(1,1),(-1,1),(1,-1),(-1,-1)]
208            for i in range(1,self.x_dim-1):
209                for j in range(1,self.y_dim-1):
210                    for k in range(4):
211                        dis = diss[k]
212                        neigh = nneigh[k]
213                        x = (i+dis[0])
214                        y = (j+dis[1])
215                        x_n = (i+neigh[0])
216                        y_n = (j+neigh[1])
217                        total_e -= self.fluid[i,j]*((1/(1+1/math.sqrt(2)))*
218                                      (self.e_ll*(self.fluid[x,y]+(1/math.sqrt(2))*self.fluid[x_n,y_n])/2 +
219                                       self.e_nl*(self.nano[x,y]+(1/math.sqrt(2))*self.nano[x_n,y_n])) +
220                                       self.mu/4)+
221                                       self.nano[i,j]*((1/(1+1/math.sqrt(2)))*
222                                       self.e_nn*(self.nano[x,y]+(1/math.sqrt(2))*self.nano[x_n,y_n])/2)
223            return total_e
```

```
    '''
    Function that performs a single nanoparticle step
    Chooses a random nanoparticle and attempts to move it in a random direction
    '''
    def step_nano(self):
        #Select nano particle to move
        i_nano = np.random.randint(0,len(self.nano_list))

        x_nano = self.nano_list[i_nano][0]
        y_nano = self.nano_list[i_nano][1]

        #Select displacement direction
        dir_nano = np.random.randint(0,4)
        hitBoundary = False
        match dir_nano:
            case 0: #+y
                dis = (0,1)
                abs_dis = dis
                offset = (0,self.nano_size)
                wake_offset = (0,0)
                hitBoundary = y_nano >= (self.y_dim-self.nano_size-1)
            case 1: #-y
                dis = (0,-1)
                abs_dis = (0,1)
                offset = (0,-1)
                wake_offset = (0,self.nano_size-1)
                hitBoundary = y_nano <= 1
            case 2: #+x
                dis = (1,0)
                abs_dis = dis
                offset = (self.nano_size,0)
                wake_offset = (0,0)
                hitBoundary = x_nano >= (self.x_dim-self.nano_size-1)
            case 3: #-x
                dis = (-1,0)
                abs_dis = (1,0)
                offset = (-1,0)
                wake_offset = (self.nano_size-1,0)
                hitBoundary = x_nano <= 1
        fluid_sum = 0
        nano_sum = 0
        for i in range(self.nano_size):
```

```
268                 fluid_sum += self.fluid[(x_nano+offset[0]+i*abs_dis[1]),(y_nano+offset[1]+i*abs_dis[0])]
269                 nano_sum += self.nano[(x_nano+offset[0]+i*abs_dis[1]),(y_nano+offset[1]+i*abs_dis[0])]
270 
271             delta_e = 0
272             nnn_e = 0
273             #Move only if no nanoparticles blocking and all cells are occupied by fluid
274             if not hitBoundary and fluid_sum == self.nano_size and nano_sum == 0:
275 
276                 #Get cell and wake cell indices
277                 x = (x_nano + offset[0])
278                 y = (y_nano + offset[1])
279                 x_wake = (x_nano + wake_offset[0])
280                 y_wake = (y_nano + wake_offset[1])
281 
282                 for i in range(self.nano_size):
283                     #Get indices of cells
284                     x_i = (x+i*abs_dis[1])
285                     y_i = (y+i*abs_dis[0])
286                     x_i_wake = (x_wake+i*abs_dis[1])
287                     y_i_wake = (y_wake+i*abs_dis[0])
288 
289                     #Add needed bond contributions - nearest neighbours
290                     delta_e += self.neigh_de(1,(x_i,y_i),dis)
291                     delta_e += self.neigh_de(-1,(x_i_wake,y_i_wake),(-dis[0],-dis[1]))
292 
293                     #Second nearest neighbours
294                     nnn_e += self.neigh_de(1,(x_i,y_i),(dis[0]+dis[1],dis[1]+dis[0]))
295                     nnn_e += self.neigh_de(1,(x_i,y_i),(dis[0]-dis[1],dis[1]-dis[0]))
296 
297                     nnn_e += self.neigh_de(-1,(x_i_wake,y_i_wake),(-dis[0]+dis[1],-dis[1]+dis[0]))
298                     nnn_e += self.neigh_de(-1,(x_i_wake,y_i_wake),(-dis[0]-dis[1],-dis[1]-dis[0]))
299 
300                     if i == 0:
301                         delta_e += self.neigh_de(1,(x_i,y_i),(-abs_dis[1],-abs_dis[0]))
302                         delta_e += self.neigh_de(-1,(x_i_wake,y_i_wake),(-abs_dis[1],-abs_dis[0]))
303 
304                         nnn_e += self.neigh_de(1,(x_i,y_i),(-dis[0]-abs_dis[1],-dis[1]-abs_dis[0]))
305                         nnn_e += self.neigh_de(-1,(x_i_wake,y_i_wake),
306                                               (dis[0]-abs_dis[1],dis[1]-abs_dis[0]))
307                     elif i == (self.nano_size-1):
308                         delta_e += self.neigh_de(1,(x_i,y_i),(abs_dis[1],abs_dis[0]))
309                         delta_e += self.neigh_de(-1,(x_i_wake,y_i_wake),(abs_dis[1],abs_dis[0]))
310 
311                         nnn_e += self.neigh_de(1,(x_i,y_i),(-dis[0]+abs_dis[1],-dis[1]+abs_dis[0]))
```

```
312                         nnn_e += self.neigh_de(-1,(x_i_wake,y_i_wake),
313                                                      (dis[0]+abs_dis[1],dis[1]+abs_dis[0]))
314                 delta_e += (self.norm)*nnn_e
315                 delta_e *= (self.tot_E_norm)
316                 #Calculate probability of accepting move
317                 att_e = self.attraction_energy(x_nano,x_nano+dis[0])
318 
319 
320                 Pacc = min(1,np.exp(-(delta_e+att_e)/self.KbT))
321                 if np.random.random() <= Pacc:
322                     #Accept
323                     self.total_energy += delta_e
324                     #Move nanoparticle
325                     for i in range(self.nano_size):
326                         x_i = (x_nano + offset[0] + i*abs_dis[1])
327                         y_i = (y_nano + offset[1] + i*abs_dis[0])
328                         x_i_wake = (x_nano + wake_offset[0] + i*abs_dis[1])
329                         y_i_wake = (y_nano + wake_offset[1] + i*abs_dis[0])
330                         self.fluid[x_i,y_i] = (1-self.fluid[x_i,y_i])
331                         self.fluid[x_i_wake,y_i_wake] = (1-self.fluid[x_i_wake,y_i_wake])
332                         self.nano[x_i,y_i] = (1-self.nano[x_i,y_i])
333                         self.nano[x_i_wake,y_i_wake] = (1-self.nano[x_i_wake,y_i_wake])
334 
335                     self.nano_list[i_nano] = ((x_nano+dis[0]),(y_nano+dis[1]))
336             return delta_e
337     #Function for attraction energy towards a line at x = att_x (in lattice coordinate)
338     #Attraction energy falls off as 1/(0.2+dist)
339     def attraction_energy(self,prev_x,next_x):
340         att_x = int(self.attraction_x*self.x_dim)
341         prev_dist = abs(prev_x-att_x)
342         next_dist = abs(next_x-att_x)
343         return self.attraction_e*(next_dist-prev_dist)/(0.2+(prev_dist/self.x_dim)*self.x_length)
344 
345     '''
346     Function that populates nanoparticle lattice with a number of nanoparticles
347     Only attempts 100 random placements - can result in a lower fraction
348     '''
349     #Randomly populate nanoparticles
350     def initialize_nano(self):
351         for i in range(self.n_nano):
352             tries = 0
353             isdone = False
354             while not isdone:
355                 tries += 1
```

```
356
357                 x_i = np.random.randint(1,self.x_dim-(self.nano_size))
358                 y_i = np.random.randint(1,self.y_dim-(self.nano_size))
359
360                 nano_sum = 0
361
362                 for i in range(self.nano_size):
363                     for j in range(self.nano_size):
364                         nano_sum += self.nano[(x_i+i),(y_i+j)]
365                 #Only place if no intersection
366                 if nano_sum == 0:
367                     for i in range(self.nano_size):
368                         for j in range(self.nano_size):
369                             self.nano[(x_i+i),(y_i+j)] = 1
370                             self.fluid[(x_i+i),(y_i+j)] = 0
371                     self.nano_list.append((x_i,y_i))
372                     self.n_nano_placed += 1
373                     isdone = True
374                 elif tries > 100:
375                     isdone = True
376         self.nano_iter = self.nano_mob*self.n_nano_placed
377         self.total_energy = self.calculate_total_energy_neigh()
378
379     '''
380     Function that performs a single epoch
381     '''
382     def step_simulation(self):
383         for i in range(self.solv_iter):
384             self.step_fluid()
385         for i in range(self.nano_iter):
386             self.step_nano()
```

Next is the network used for the pipeline.

```
1  import torch.nn as nn
2  import torch
3
4  class Net(nn.Module):
5      def __init__(self,depth=1,width=100,input_size=1,output_size=4):
6          super(Net, self).__init__()
7          self.layers = nn.ModuleList([nn.Linear(input_size,width)]+
8                                      [nn.Linear(width,width) for i in range(depth-1)])
9          #N hidden layers, N = depth-1: width -> width
10         self.output = nn.Linear(width,output_size) #Output layer: width -> output_size
```

```
        self.act = nn.Tanh()
        self.elu = nn.ELU()

    def forward(self, x):
        #Forward pass through all hidden layers with relu activation
        for i, layer in enumerate(self.layers):
            x = self.act(layer(x))
        x = self.output(x) #Output layer
        x[:,0] = self.elu(x[:,0]) + 1.0
        x[:,2] = (self.act(x[:,2])+1)/2
        x[:,3] = x[:,3]/5
        return x
```

This snippet of code contains all supporting functions necessary:

```
import numpy as np
import copy
import torch
import meep as mp
from GrowthSimCleaner import Growth_NonPeriodic as Growth

mp.verbosity(0)

#Function that generates a random seed
def Generate_Seed():
    return np.random.randint(0,np.iinfo(np.uint32).max,dtype=np.uint32)

#Function that generates random new seeds for each network
def Generate_New_Seeds(seeds):
    for n in range(len(seeds)):
        seeds[n][0] = Generate_Seed()

#Function that initializes networks
def Initialize_Net(net):
    net.output.bias[0] -= 1.0
    net.output.bias[1] -= 2.5
    net.output.bias[3] += 0.4
#Function that sorts networks based on score - return sorted indices
def Sort_Growths(seeds,scores):
    #Sort from high to low
    sorted_indices = np.argsort(scores)[::-1]
    scores = [scores[i] for i in sorted_indices]
```

```
28      seeds = [seeds[i] for i in sorted_indices]
29
30      #Reorder the indices
31      for i in range(len(seeds)):
32          seeds[i][1] = i
33      return seeds, scores
34
35  #Function that copies kept networks into culled networks
36  #def Clone_Nets(listOfNets,kept):
37  #    for i in range(len(listOfNets)-kept):
38  #        listOfNets[kept+i][0] = copy.deepcopy(listOfNets[i%kept][0])
39
40  #Function that mutates the networks
41  #Does not mutate
42  def Mutate_Net(net,sigma):
43      for param in net.parameters():
44          param += torch.normal(torch.zeros(param.shape),sigma)
45
46  #Function that initializes a growth simulation
47  def Initialize_Growth(seed,baseline=False):
48      #Setup parameters
49      y_dim = 750
50      x_dim = 500
51      frac = 0.3
52      mu = -2.5
53      KbT = 0.2
54      e_nn = 2
55      e_nl = 1.5
56      e_ll = 1
57      nano_mob = 30
58      nano_size = 3
59      n_nano = int(frac*(x_dim*y_dim)/(nano_size*nano_size))
60      lattice_size = 2.8/x_dim
61
62      args = {'x_dim':x_dim,
63              'y_dim':y_dim,
64              'n_nano':n_nano,
65              'KbT':KbT,
66              'mu':mu,
67              'e_nn':e_nn,
68              'e_nl':e_nl,
69              'e_ll':e_ll,
70              'nano_mob':nano_mob,
71              'nano_size':nano_size}
```

```
72
73
    #Initialize growth
    growth = Growth(**args,seed=seed)

    #Empty boundaries
    growth.fluid[:,0] = 0
    growth.fluid[:,-1] = 0
    growth.fluid[0,:] = 0
    growth.fluid[-1,:] = 0

    #Add probes
    #Probe width
    w = 0.5

    #Current dimension, 1000 = 2.8 micro, this gives us how many lattice cells for the probe width
    probe_width = int(w/lattice_size)
    mid_probe = int((x_dim-probe_width)/2)

    if baseline:
        #Width probe
        growth.nano[mid_probe:mid_probe+probe_width,:] = 1
        growth.fluid[mid_probe:mid_probe+probe_width,:] = 0

    else:
        #Left probe
        growth.nano[mid_probe:mid_probe+probe_width,0:(x_dim//4)] = 1
        growth.fluid[mid_probe:mid_probe+probe_width,0:(x_dim//4)] = 0

        #Right probe
        growth.nano[mid_probe:mid_probe+probe_width,-(x_dim//4):] = 1
        growth.fluid[mid_probe:mid_probe+probe_width,-(x_dim//4):] = 0

        #Initialize nanoparticles
        growth.initialize_nano()
    return growth

#Function that generates an e_func based on lattice_size and chip
def generate_chip_e_func(chip,lattice_size):
    n_si = 3.49
    n_sio4 = 1.45
    n_air = 1.0
    epsilon_si = n_si**2
    epsilon_sio4 = n_sio4**2
```

```
116     epsilon_air = n_air**2
117
118     x_shape = chip.shape[0]
119     y_shape = chip.shape[1]
120
121     xspan = (chip.shape[0])*lattice_size
122     yspan = (chip.shape[1])*lattice_size
123
124     def chip_e_func(coord):
125         x,y,z = coord
126
127         if z > 0.11:
128             return epsilon_sio4
129         elif z < -0.11:
130             return epsilon_air
131         else:
132             x_lat = int((x+x_shape/2*lattice_size)/lattice_size)
133             y_lat = int((y+y_shape/2*lattice_size)/lattice_size)
134             if x >= xspan/2:
135                 x_lat = 0
136             elif x <= -xspan/2:
137                 x_lat = -1
138             if y >= yspan/2:
139                 y_lat = -1
140             elif y <= -yspan/2:
141                 y_lat = 0
142             return chip[x_lat,y_lat]
143     return chip_e_func
144
145
146 #Function that runs PyMeep and scores growth in 3D
147 def Score_Growth_3D(chip,resolution = 25, flux_width=1.1, tmax=100, baseline=False, e_func=False):
148     w_lattice = np.sum(chip[:,0])
149     x_dim = chip.shape[0]
150     y_dim = chip.shape[1]
151     lattice_size = 2.8/x_dim
152     w = w_lattice * lattice_size
153     h = 0.22
154     h_lattice = int(h/lattice_size)
155
156     n_si = 3.49
157     n_sio4 = 1.45
158     n_air = 1.0
159     epsilon_si = n_si**2
```

```
160     epsilon_sio4 = n_sio4**2
161     epsilon_air = n_air**2
162 
163     if not e_func:
164         chip = np.pad(chip,((x_dim//4,x_dim//4),(x_dim//2,x_dim//2)),mode='edge')
165 
166         chip = (chip)*(epsilon_si-epsilon_air) + epsilon_air
167         chip.shape += 1,
168         chip = np.pad(chip,((0,0),(0,0),(h_lattice//2,h_lattice//2)),mode='edge')
169 
170         #SiO4 pad
171         chip = np.pad(chip,((0,0),(0,0),(0,int(1/lattice_size))),
172                                         mode='constant',constant_values=epsilon_sio4)
173 
174         #Air pad
175         chip = np.pad(chip,((0,0),(0,0),(int(1/lattice_size),0)),mode='constant',constant_values=1.0)
176 
177         #Setup for PyMeep simulation
178         shape = chip.shape
179         xspan = shape[0]*lattice_size
180         yspan = shape[1]*lattice_size
181         zspan = shape[2]*lattice_size
182 
183     elif e_func:
184         chip = (chip)*(epsilon_si-epsilon_air) + epsilon_air
185         xspan = (chip.shape[0]+(x_dim//4)*2)*lattice_size
186         yspan = (chip.shape[1]+(x_dim//2)*2)*lattice_size
187         zspan = 2+h
188         e_func = generate_chip_e_func(chip,lattice_size)
189     cell = mp.Vector3(xspan,yspan,zspan)
190     pml_layers = [mp.PML(0.5)]
191 
192     wcen = 1.3
193     fcen = 1/wcen
194     df = 0.5
195 
196     sources = [mp.Source(mp.GaussianSource(fcen,fwidth=df),
197                         component=mp.Ex,
198                         center=mp.Vector3(0,-2.8,0),
199                         size=mp.Vector3(w*flux_width,0,h*flux_width))]
200 
201     if not e_func:
202         sim = mp.Simulation(cell_size=cell,
203                             boundary_layers=pml_layers,
```

```
204                             sources=sources,
205                             default_material=chip,
206                             resolution=resolution)
207     elif e_func:
208          sim = mp.Simulation(cell_size=cell,
209                             boundary_layers=pml_layers,
210                             sources=sources,
211                             epsilon_func=e_func,
212                             resolution=resolution)
213     nfreq = 100
214 
215     if baseline:
216         flux_fr_in = mp.FluxRegion(center=mp.Vector3(0,2.5,0),
217                                    size=mp.Vector3(w*flux_width,0,h*flux_width))
218         flux_in = sim.add_flux(fcen,df,nfreq,flux_fr_in)
219         print('Starting␣PyMeep␣baseline␣simulation',flush=True)
220         sim.run(until=tmax)
221         print('Finished␣PyMeep␣baseline␣simulation',flush=True)
222         baseline_flux = mp.get_fluxes(flux_in)
223         flux_freqs = mp.get_flux_freqs(flux_in)
224         eps = sim.get_array(center=mp.Vector3(),size=cell,component=mp.Dielectric)
225         sim.reset_meep()
226         np.savez_compressed('Baseline_eps.npz',eps=eps)
227         np.savez_compressed('Baseline.npz',flux=baseline_flux,freqs=flux_freqs)
228         print('Baseline␣saved',flush=True)
229         return eps
230 
231     else:
232         flux_fr_out = mp.FluxRegion(center=mp.Vector3(0,2.5,0),
233                                       size=mp.Vector3(w*flux_width,0,h*flux_width))
234         flux_out = sim.add_flux(fcen,df,nfreq,flux_fr_out)
235         print('Starting␣PyMeep␣simulation',flush=True)
236         sim.run(until=tmax)
237         print('Finished␣PyMeep␣simulation',flush=True)
238 
239         chip_flux_out = mp.get_fluxes(flux_out)
240 
241         eps = sim.get_array(center=mp.Vector3(),size=cell,component=mp.Dielectric)
242         sim.reset_meep()
243 
244         baseline = np.load('Baseline.npz')
245         input_flux = baseline['flux']
246         input_freqs = baseline['freqs']
247         wvl = 1/input_freqs
```

```
248
249         Ts_out = chip_flux_out/input_flux
250         scorer = bandpass_scorer_target(wvl)
251         score = -np.sum(np.square(scorer-Ts_out))/len(Ts_out)
252
253
254         return score,chip_flux_out,Ts_out
255
256
257 def bandpass_scorer(wavelength):
258     scorer = np.zeros(wavelength.shape)-2
259     scorer += 2*np.exp(-np.square(wavelength-1.3)/(2*np.square(0.05)))
260     return scorer
261
262 def bandpass_scorer_target(wavelength):
263     scorer = np.zeros(wavelength.shape)
264     scorer += np.exp(-np.square(wavelength-1.3)/(2*np.square(0.05)))
265     return scorer
266
267
268 #Score transmissions as to equally split the light out both outputs
269 #Maximize total transmission and minimize difference in outputs (both in magnitude and what wvls)
270 def equal_trans(trans_up,trans_dw):
271     total_trans = trans_up+trans_dw
272     score = np.sum(total_trans) - np.sum(np.abs(trans_up-trans_dw))
273     return score
274
275 #Function that returns an array to score an array of transmission
276 def gaussian_scorer(wavelength):
277     scorer = np.zeros(wavelength.shape)
278     scorer += np.exp(-np.square(wavelength-1.3)/(2*np.square(0.05)))
279     scorer -= np.exp(-np.square(wavelength-1.55)/(2*np.square(0.05)))
280     return scorer
281
282
283 #Save a checkpoint
284 def Save_Checkpoint(net,seeds,epoch,cur_score):
285     seeds_l = []
286     np_rng = np.random.get_state()
287     torch_rng = np.array(torch.get_rng_state())
288     torch.save(net.state_dict(),'Checkpoint/NetStateDict_epoch_%g.pt'%(epoch))
289     for j in range(len(seeds)):
290         seeds_l.append(seeds[j][0])
291     np.savez_compressed('Checkpoint/Checkpoint_epoch_%g.npz'%(epoch),
```

```
                            cur_score=np.array([cur_score]),seeds=np.array(seeds_l),
                            np_rng=np_rng,torch_rng=torch_rng)
    print('Checkpoint␣Created',flush=True)

#Load a checkpoint
def Load_Checkpoint(net,seeds,epoch):
    checkpoint = np.load('Checkpoint/Checkpoint_epoch_%g.npz'%(epoch),allow_pickle=True)
    seeds_check = checkpoint['seeds']
    cur_score = checkpoint['cur_score'][0]
    np.random.set_state(tuple(checkpoint['np_rng']))
    torch.set_rng_state(torch.tensor(checkpoint['torch_rng']))
    for j in range(len(seeds)):
        seeds[j][0] = seeds_check[j]
    net.load_state_dict(torch.load('Checkpoint/NetStateDict_epoch_%g.pt'%(epoch)))
    print('Checkpoint␣Loaded',flush=True)
    return cur_score

#Save best performing networks
def Save_Network(net,epoch):
    torch.save(net.state_dict(),'Results/HighScore_NetStateDict_epoch_%g.pt'%(epoch))
```

Finally, here is the code to run the full pipeline:

```
from GrowthPipeline import Generate_Seed, Generate_New_Seeds, Initialize_Net,
                          Sort_Growths, Mutate_Net, Score_Growth_3D,
                          Initialize_Growth, Save_Checkpoint, Load_Checkpoint,
                          Save_Network
from GrowthNetworks import Net

#Multiprocessing for Pool and cpu_count
import multiprocessing

import torch
import numpy as np
import time
import sys
import copy


####
#Function for Pool - single epoch
#####
def Simulate(seed,index):
    global net, epoch, n_pool
```

```
    #Initialize growth simulator - need random seed to be random for all processes
    if index%n_pool == 0:
        print('Index␣%g␣initializing␣growth'%(index),flush=True)
    growth = Initialize_Growth(seed)
    if index%n_pool == 0:
        print('Index␣%g␣growth␣initialized'%(index),flush=True)
    #Grow, querying net
    if index%n_pool == 0:
        print('Index␣%g␣starting␣growth'%(index),flush=True)
        growth_t0 = time.time()
    for n in range(MC_steps):
        if n%MC_query == 0:
            netOutput = net(torch.tensor([n/MC_steps]).reshape(-1,1).float()).detach().numpy()[0]
            newKbT = netOutput[0]
            newMu = netOutput[1]
            newAtt_x = netOutput[2]
            newAtt_e = netOutput[3]
            growth.KbT = newKbT
            growth.mu = newMu
            growth.attraction_x = newAtt_x
            growth.attraction_e = newAtt_e
            if index%n_pool == 0:
                growth_t1 = time.time()
                print('Index␣%g␣growth␣progress:␣%.1f%%,
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣%g/%g␣in␣%.1f␣min'%(index,((n)/(MC_steps))*100,
                                      n,MC_steps,(growth_t1-growth_t0)/60),flush=True)
        growth.step_simulation()

    if index%n_pool == 0:
        growth_t1 = time.time()
        print('Index␣%g␣growth␣finished␣in␣%1f␣min'%(index,(growth_t1-growth_t0)/60),flush=True)

    #calculate score of net
    score,flux_out,ts_out = Score_Growth_3D(growth.nano,e_func=True)

    if index < 5:
        np.savez_compressed('Results/growth_epoch_%g_index_%g.npz'%(epoch,index),
                                    growth=growth.nano,score=np.array([score]),flux_out=np.array(flux_out),
                                    ts_out=np.array(ts_out))
    return [score]


if __name__ == '__main__':
    #####
```

```
    #Setup parameters
    #####
    #Default on Linux - set it for consistency
    multiprocessing.set_start_method('fork')

    print('Number␣of␣CPU␣cores:␣%g'%(multiprocessing.cpu_count()))

    torch.set_grad_enabled(False)

    #Reproducibility
    np.random.seed(1239)
    torch.manual_seed(1239)

    #Evolutionary learning parameters
    epochs = 5
    MC_steps = 2000
    MC_query = 50
    sigma = 0.01

    #Network
    net = Net()

    #Setup
    n_growths = 80
    n_pool = 40
    seeds = []
    for i in range(n_growths):
        seeds.append([Generate_Seed(),i])

    #Determine current epoch and load checkpoint if not start
    base_epoch = int(int(sys.argv[1])*epochs)
    print('Starting␣epoch:␣%g'%(base_epoch))
    if base_epoch == 0:
        Initialize_Net(net)
        cur_score = -np.inf
        print('Initialized␣Net␣for␣epoch␣0␣with␣initial␣score␣%.2f'%(cur_score),flush=True)
    else:
        cur_score = Load_Checkpoint(net,seeds,base_epoch-1)
        print('Loaded␣checkpoint␣from␣epoch␣%g'%(base_epoch - 1),flush=True)

    #Starting time
    t0 = time.time()

    checkpointAlways = True
```

```
110     #Run neuroevolution growth
111     for epoch in range(base_epoch,base_epoch+epochs):
112         tes = time.time()
113 
114         print('\n------------------------\nEpoch␣%g'%(epoch),flush=True)
115         #Get previous state dict, mutate and generate new seeds
116         net_params = copy.deepcopy(net.state_dict())
117         Mutate_Net(net,sigma)
118         Generate_New_Seeds(seeds)
119         print('Starting␣parallel␣growths',flush=True)
120         #Run growths in parallel
121         with multiprocessing.Pool(n_pool) as pool:
122             simulate_output = pool.starmap(Simulate, seeds)
123 
124         #Calculate mean score
125         scores = [simulate_output[i][0] for i in range(len(simulate_output))]
126         mean_score = np.mean(np.array(scores))
127 
128         #Accept if new score is higher, reject otherwise
129         if mean_score >= cur_score:
130             cur_score = mean_score
131         else:
132             net.load_state_dict(net_params)
133 
134         #Saving data
135         #Sort networks and scores
136         seeds, scores = Sort_Growths(seeds,scores)
137 
138         print('Scores:␣',scores)
139         np.savez_compressed('Results/scores_epoch_%g.npz'%(epoch),
140                              score=np.array(scores),mean_score=np.array([mean_score]),
141                              cur_score=np.array([cur_score]))
142 
143 
144         #Checkpoint every N epochs and at final epoch
145         if checkpointAlways or epoch%5 == 0 or epoch == base_epoch+epochs-1:
146             Save_Checkpoint(net,seeds,epoch,cur_score)
147 
148         tee = time.time()
149         print('\nEpoch␣%g␣Finished␣in␣%.2fmin\ntime␣remaining:␣%.2fhr\nmean␣score:%.2f\ncur␣score:%.2f
150 ␣␣␣␣␣␣␣␣␣␣␣␣\nbest␣score:%.2f\n'%(epoch,(tee-tes)/60,((tee-tes)/3600)*(base_epoch+epochs-epoch-1),
151              mean_score,cur_score,max(scores)),flush=True)
152 
153 
```

```
154     t1 = time.time()
155     print('Finished job in %.1fhr'%((t1-t0)/3600),flush=True)
```