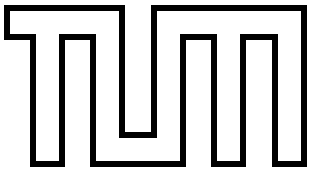


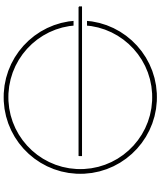
TUM DEPARTMENT OF INFORMATICS

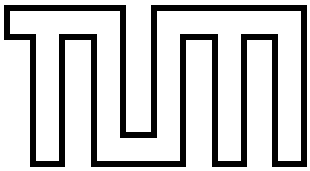
TECHNICAL UNIVERSITY OF MUNICH

Bachelor's Thesis in Informatics

Auto-Tuning via Machine Learning in AutoPas

Deniz Candas





TUM DEPARTMENT OF INFORMATICS

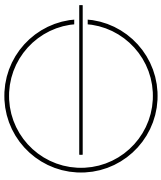
TECHNICAL UNIVERSITY OF MUNICH

Bachelor's Thesis in Informatics

Auto-Tuning via Machine Learning in AutoPas

Auto-Tuning via Maschinellem Lernen in AutoPas

Author:	Deniz Candas
Supervisor:	Univ.-Prof. Dr. Hans-Joachim Bungartz
Advisor:	Steffen Seckler, Fabio A. Gratl
Date:	16.09.2019



I confirm that this bachelor's thesis is my own work and I have documented all sources and material used.

Munich, 16.09.2019 Deniz Candas

Acknowledgments

This thesis was written under the supervision of Steffen Seckler and Fabio Alexander Gratl. We also thank the LRZ for providing the computational units to gather simulation data for machine learning.

Abstract

AutoPas is a C++ library capable of running molecular dynamics simulations with different optimization schemes, which are configurations with differing data structures (i.e. array of structures or structure of arrays), traversal strategies, container types (e.g. linked cells, verlet lists) and optimization techniques (enabling Newton 3 optimization). The current methodology uses an algorithm that tests out the whole search space, which increases day by day. In this thesis, the process of training a machine learning model based on neural networks to create an auto-tuner capable of suggesting the best simulation configuration available to AutoPas is shown. This strategy reduces search time by testing fewer options, and it chooses the optimal configuration with a likelihood of over 99%.

Contents

Part I.

Introduction and Background

1. Introduction

N-Body simulations are an important subset of particle simulations, which bring key insights to many fields, but are time consuming. There are many ways to run such simulations and gather the same results at the end. These types of changes can be called configurations for the given simulations. Some options for these changes include data layout, such as using structure of arrays instead of an array of structures, enabling an optimization called Newton 3, which halves the required amount of computations for pairwise force calculations, changing the container, which leads to preemptively avoiding calculations that are unnecessary, or changing the way the data is traversed. Often, it is not clear what kind of programmatic change would lead to an increase in performance, which is a problem addressed by auto-tuners, devices or software that adjust values to optimize any type of work, such as PID controllers [Mon+08]. There are also auto-tuners for compilers, even dynamic ones that use machine learning [Fur+11]; however, most of the compilers that use auto-tuning do so statically. The library AutoPas [Gra+19], which can currently only simulate short-ranged n-body problems, trials all types of configurations periodically to achieve optimal speeds. However, most configurations are non-linearly correlated with parameters of a simulation, which means that certain parts of the solution subspace can be inferred to be sub-optimal without running trials that are time consuming to create and go through. The non-linearity of this correlation means machine learning, especially with neural networks [Sar94], can be used to choose a sub-space that includes the optimal solution with high likelihood. Moreover, particle distributions are possible parameters to use so that a model can learn to estimate what type of configuration is more likely to be optimal. This is similar to image classification which is highly supported by frameworks that enable machine learning [Tena].
First, a short introduction about the background of numeric programming for molecular dynamic simulations and machine learning is presented in this thesis, which is followed by the steps taken to test out different machine learning models to solve this problem. Finally, the results are reported and interpreted for the given context with possible improvements.

2. N-Body Simulations and AutoPas

2.1. N-Body Simulations

A particle simulation is the calculation of numeric values based on discrete approximations of continuous values. A particle simulation may have from only one to millions of particles in an environment, which is why it is also called an N-Body Simulation [TH08], where different forces act upon the particles, such as external forces like gravity, which can be computed individually for each particle, or the Lennard Jones potential, which requires an interaction between all particle pairs. Because some of these forces are only significant when the interaction range is below a threshold, i.e. short-ranged, it is possible to use different strategies to optimize calculations that have values which are close to zero. The aim of a given simulation is; hence, to calculate correct results for given values and constraints as efficiently as possible. The application areas of N-Body Simulations include molecular dynamics.

2.2. Molecular Dynamics

The type of molecular dynamics in this thesis is a method of computer simulation which uses numerical programming to approximate interactions that cannot be modeled analytically because of multiple sources of interaction, such as many bodies that exert a force on each other [AW59]. Molecular dynamics uses N-Body-Simulations, and in this thesis with a focus on the simplest case of Lenard Jones [Ver67] Potential to calculate the forces exerted upon molecules by each other.

$$V_{LJ} = 4\varepsilon[(\frac{\sigma}{2})^{12} - (\frac{\sigma}{2})^{6}]$$

LJ-Potential uses epsilon for the depth of the potential well, sigma for the value where the particles are assumed to be no longer interacting with each other, and r for the distances between the particles. The LJ-Potential is shown in Figure 2.1.

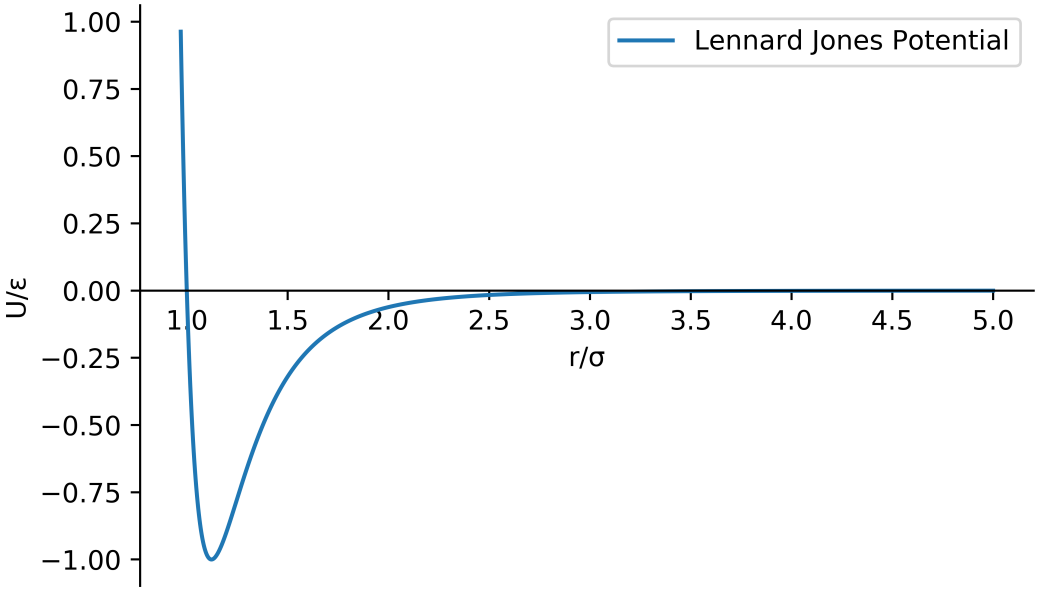


Figure 2.1.: Graph of Lennard Jones Potential

It is clear from the graph that two molecules pull each other until a certain value, after which they strongly push each other away. This way two important properties of atoms and molecules can be approximated with one function: First, that every particle attracts each other due to van der Waals forces [Dzy+92], and second, that particles which bump into each other repel each other strongly, which is due to Pauli Exclusion Principle. By using this equation in each time step, forces acting on all particles can be calculated accordingly. This in turn can be used to recalculate the velocities and the distance traveled within the time step.

2.3. The Modeling of a Simulation

There are multiple methods of representing data and calculating forces, which was briefly mentioned in the previous section. Here, the modeling methods related to AutoPas will be explained, which are various parameters that effect performance.
When calculating forces, movement and everything else that happens during a simulation, most calculations are uniformly done on all particles, such as temperature adjustments or calculation of new coordinates. These calculations are independent for each particle, i.e. it doesn't matter what or where the other particles are. These types of calculations can be done efficiently in parallel, for example by giving each processing unit a fix number of particles to calculate. However, short-ranged force calculations, which are also computationally the most expensive parts of a simulation, are interactions between particles that have an effect only if

the distance between two particles are under a given cutoff value. As this is the case, it is possible to speed up a calculation immensely if one knows which particles interact with each other. A naive method which ignores this concept is called Direct Sum, which checks for each particle, whether the other particle is in range, and if so, calculates the interaction force. A better method called Linked Cells sorts particles into cells, where a cell is a spatial section of a grid structure of the simulation domain. If the cell size is chosen to be greater than the cutoff radius, then only particles inside the same cell and neighboring cells are known to interact with each other, which means that no distance comparison must be made between particles of cells that are further away. More complicated methods such as Verlet Lists hold a list of possible interaction partners for each particle so that the number of calculations are reduced. However, this method takes up a lot of memory and proper tuning is required to make sure that a verlet list never skips a particle that might be in the interaction range. These are some options for containers that hold information about short range particles.

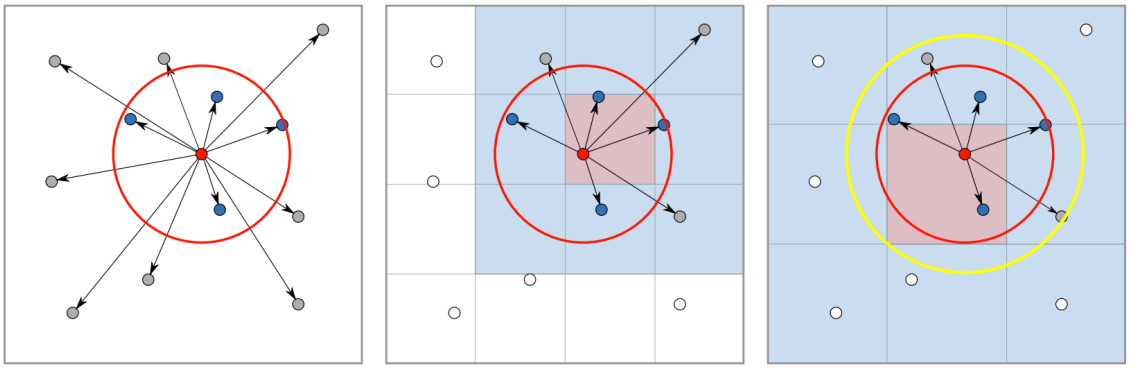


Figure 2.2.: Different containers for particles Source: [Gra+19]

Traversals are, on the other hand, methods that define an order of operations to utilize containers. This is critical for parallel computations, which are managed by the OpenMP API [Ope18], and can cause massive speedups due to less time spent waiting for other computation units caused by unbalanced loads or requests to change structures that are already in use by other computation units.
Data representation is mainly done in two ways, either as an Array of Structures (AoS), or as a Structure of Arrays (SoA) [Adm19]. An Array of Structures can be imagined as a list of particles. Each structure, or each particle, has relevant information for the simulation, such as an identifier, a location vector, a velocity vector, mass, etc. This information is later used by each iteration of a simulation. On the other hand, when using Structure of Arrays, instead of each structure having for example its own mass, the mass of every particle is found in one array. This is done for every attribute of a particle, until a structure is built, which is composed of arrays. SoA are beneficial for vectorization, which leads to operations that work on entire arrays. These are currently the only two possible methods for the data layout.

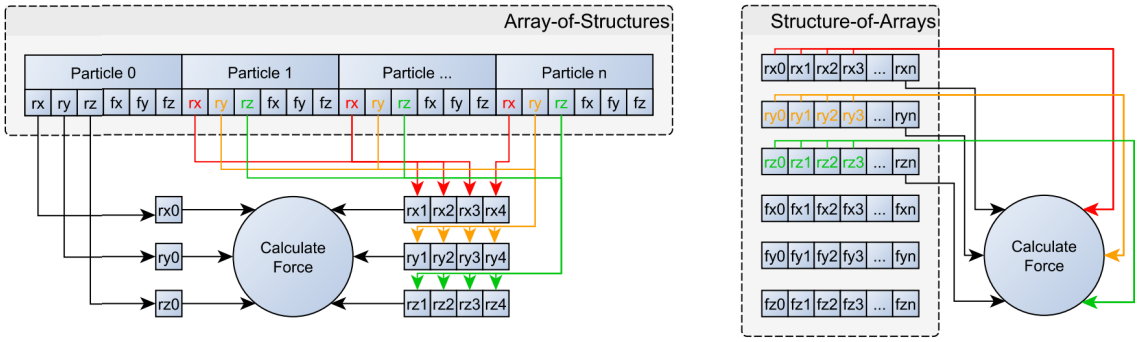


Figure 2.3.: Differences between loading a particle in AoS and SoA Source: [Gra+19]

A process which reduces the total number of short-ranged force computations by half, but increases complexity, is called Newton 3 Optimization. This method is based on Newton's third law of motion which states that "for every action there is an equal and opposite reaction" [New]. By calculating a pair's force for one particle, it is possible to multiply this force with minus one to calculate the force which will act upon the second particle for a given interaction. The complexity arises from the fact that when using OpenMP to parallelize computation, it is possible that synchronization problems might occur, which can be circumvented with techniques such as coloring. Refer to A under the appendix for additional information.

2.4. AutoPas

Autopas is a C++ library capable of calculating short-ranged pairwise particle interactions and it can also dynamically choose the optimal simulation configuration during run-time with the use of tuning phases. [Gra+19] This can be achieved using different strategies, such as the full-search pattern already implemented in the library, which tests out all possible configurations at given intervals. In this thesis, the main objective is to implement another strategy based on machine learning to find optimal configurations without having to test all possible strategies.

3. Background about Machine Learning

3.1. What does machine learning solve and how

Machine learning is a popular method which is heavily researched and used in industries to solve problems for which there are no obvious or algorithmic solutions, such as financial forecasting [LHT11] or recognizing facial features [Bar+05]. By analyzing data using statistical models, it is possible to predict the value (usually called the label) of an unseen sample. There are mainly two methods of learning, supervised and unsupervised learning. Unsupervised learning uses techniques such as clustering to group a set of samples, thereby deciding which features of the data make certain groups of samples similar. On the other hand, supervised learning receives a dataset of labeled samples, called a training dataset, where it learns to associate features and labels, and a dataset without labels, which it classifies on its own. In most cases, this second dataset has its labels removed, and the accuracy of the model is measured with this second dataset, called the testing dataset. Machine learning is especially effective in cases where a lot of data is available or easy to generate, such as games and simulations [Sil+16]. In the case of this thesis, a methodology which can discover knowledge about simulations without running new ones is paramount; hence, machine learning -especially deep learning, which uses layers of functions to represent and analyze data- has been tested to figure out whether previous simulation data can be used effectively to estimate the quickest configuration of a set of parameters that have not been previously seen.

3.2. Neural Networks and Alternatives

Neural networks are based on human brains, which are composed of interconnected neurons. By simulating the way humans think, it is possible to train computers (or what is simulated by the computers) to think and learn just like humans. This is done by building networks of connected cells that are triggered by other cells or input data until a result is reached at the end. Neural networks are similar to and somewhat inspired by decision trees, which point to a result by continuously choosing between paths that lead to a leaf node containing a possible result. However, neural networks can have multiple links between multiple cells, and they can be fully connected between layers of nodes, which is not possible for the decision tree with limited number of paths that continuously split or join. Another benefit of neural networks, compared to alternatives, is being able to learn from data non-linearly. A popular method for machine learning is called linear regression, which puts together a polynomial formula to calculate values for given parameters, such as calculating the value of a car by its age and model. These types of models are appropriate for situations where linear associations do exist; however, confounding, missing or faulty variables make such models inadequate compared to ones that can model a more complex reality. Non-linear regression

can overcome some of these difficulties, and there are arguments that neural networks can be classified as parametric non-linear regression models, but neural networks allow the user to avoid defining anything about the problem being solved, except as to what parameters and labels are. Hence, a neural network can learn to become a solution from a bigger class of possibilities [httb].

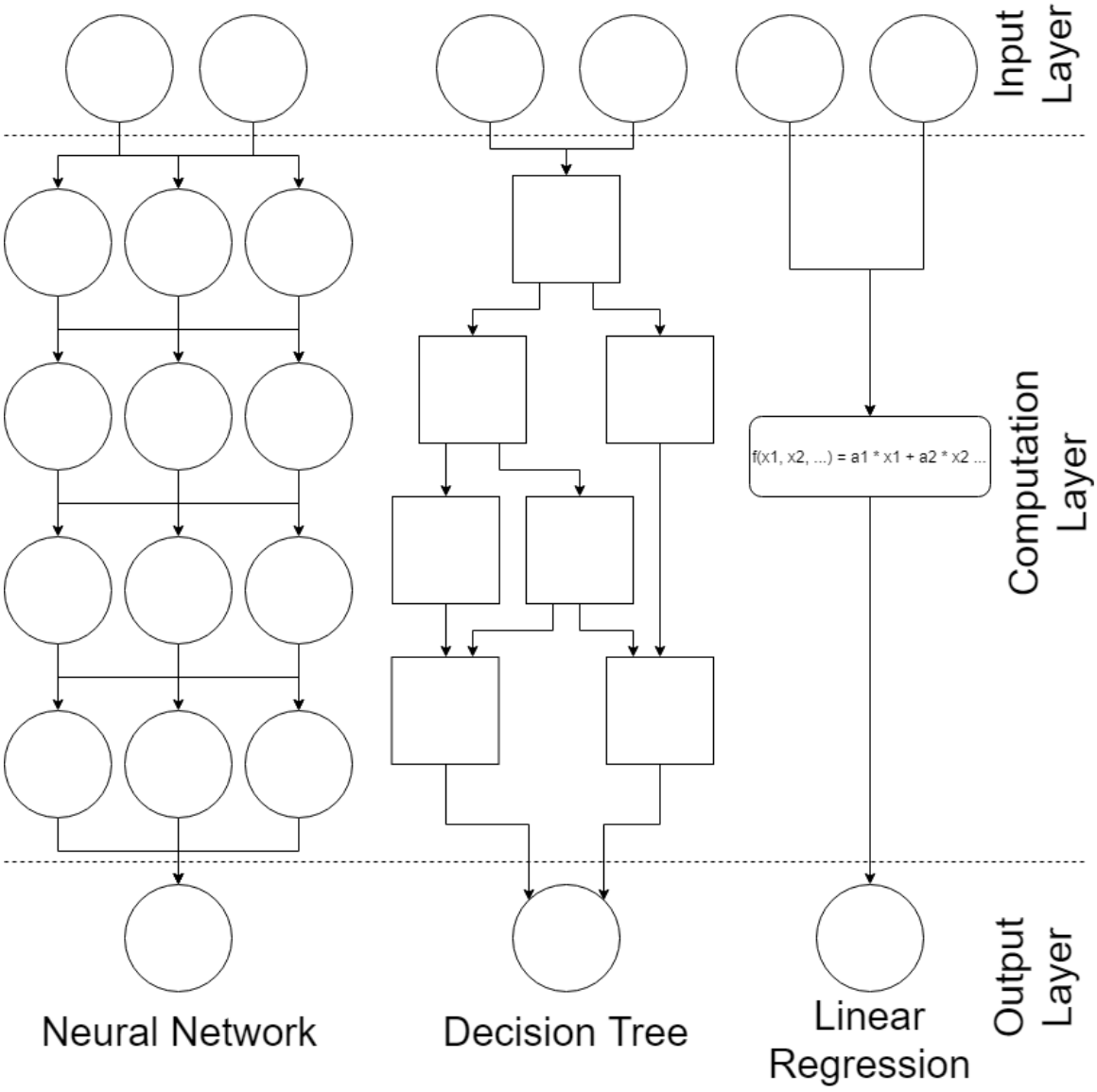


Figure 3.1.: How a neural network, decision tree and a linear regression model would solve the same problem

3.3. Classification or Regression

The two main types of problems supervised learning solves are called classification and regression. Classification is about putting samples of data into correct categories, and the concern of regression is to estimate a numeric value using the given features. As previously explained, the goal of this thesis is to accurately predict which configuration will be the quickest. Both of these methods are appropriate to solve this problem: the configurations can be classified using features, or the duration of an experiment can be regressed and the quickest experiment can be chosen from the calculated values.

3.4. What and why TensorFlow and Keras

TensorFlow is an "end-to-end open source machine learning platform", which provides the necessary tools to build, test and deploy machine learning models [Tenb]. Low level systems are built using C++ libraries and can be called using higher level APIs, such as Keras [Ker], which enables a researcher or data scientist to utilize effective methodologies without having to deal with unnecessary complexity or code that runs slowly. The author of this thesis utilized versions 1.13.1 and 1.14.0 of TensorFlow with the built-in Keras API to create neural networks and to commence deep learning using simulation data.

Part II.

Defining and Testing Models

4. Parameter Identification

A set of simulations must be run with different options to train the ML model. These options, or parameters, are chosen to reflect different use cases so that the model can estimate the optimal simulation configuration for a wide range of simulation possibilities. One of the main challenges is to identify other parameters, in addition to user given parameters which shall be referred to as basic parameters. Basic parameters are sufficient to achieve high accuracy for most situations; however, additional parameters that further increase accuracy of predictions will be discussed in the following sections.

4.1. Basic Parameters

Basic parameters are values given by the user that have a strict and easy to identify feature. These include the number of particles (how many particles there are in the simulation), box length (the length of one side of the simulation's boundary), cut off radius (maximum interaction range between two particles), verlet rebuild frequency and verlet skin radius. Although these values are critical to determine the optimal configuration, they are in certain cases insufficient. An example can be seen between two simulations that use the same basic parameters, however one being equally distributed across the simulation space and the other having all of its particles in one cell. These two simulations have wildly different simulation times and react differently to containers, one being much quicker with e.g. linked cells and the other with direct sum. Here, it can be seen that basic parameters are global values that highly influence which configurations are optimal, yet they are not sufficient on their own to determine whether a given configuration is really the optimal one in all cases. Without values that describe the particle distribution, it is impossible to classify the previously given example.

4.2. Particle Distribution

As there is no easy way to describe spatial features of many bodies with simple parameters, a solution is to note down how crowded each part of the simulation space is. This is done by splitting the simulation space into a N^3 multi-level array, which represents how many particles there are in each subspace. Hence, this process can be seen as creating a three-dimensional picture of the particle distribution. A naive way to process the distribution is simply to offer is as a set of N^3 parameters.

4.3. Using Distribution Statistics

Another way to analyze the distribution is to create meaningful statistics that describe it. For example, one could offer parameters that describe how crowded each cell is compared to the most crowded cell. This way more parameters can be identified to further optimize the model.

5. Model Selection

5.1. Choosing Parameters

Three different sets of parameters have been chosen as input layers. The first set is four basic parameters, namely particle count, box length, cut off and verlet skin radius. Another option is only using the generated picture, which is first reduced to a resolution of 8x8x8. Finally, one can also use the super set of these parameters for higher accuracy.

5.2. Metrics and Loss

There are multiple ways to define loss metrics, which are the functions machine learning uses to optimize neural networks. The models based on classification between multiple classes use sparse categorical cross-entropy, which functions like categorical cross-entropy, except integers that represents the correct class are used instead of a one-hot encoding, which has one high bit and the rest of the bits are low [HHS]. The idea is to assign each class a probability value between zero and one, and to ensure their sum is also one. Sparse categorical cross-entropy can be calculated using the negative log of a prediction for each category separately and summing up the negative log values of samples, where a sample belongs to the given class. Cross entropy loss is the log loss values; hence, a greater entropy means that the estimator is a poor one, and less entropy for a given vector of results means that the estimator is better. Entropy approaches zero as the estimators success increases, and a value of zero means that the estimator is perfect, which isn't possible if there is noise in the data. On the other hand, the loss can approach infinity because there is no upper limit to how bad an estimator can be. A formula for it is as follows [M69]:

$$-\frac{1}{N}\sum_{s\in S}\sum_{c\in C} 1_{s\in c}\log p(s\in c),\ \text{where C = Classes, S = Samples} \tag{5.1}$$

Regression models minimize mean squared error to calculate the most likely simulation time for each given set of parameters, where the simulation configuration (which was a label for sparse categorical cross-entropy) is not used as a classification label. By calculating the expected simulation time for each configuration in a separate branch, a set of regression models return a vector of timings, and the index of the shortest timing is the configuration that is determined to be quickest.

5.3. Defining Layout

The layout for categorization can be a fully-connected network (Figures 5.1-5.4) or a network that has two separate branches (Figures 5.5 and 5.6), one branch for the number of particles of sub-spaces (called distribution or picture parameters) and the other branch for the basic parameters. Additionally, if regression is used to estimate simulation time, this estimation must be done separately for each configuration. Afterwards, these estimations are collected in a results vector, where they are compared, and the shortest estimates are returned as most likely predictions. Because all results are calculated separately in their own model, they don't have to be trained using the same data. A list of layouts that depict these possibilities, Figures 5.1 to 5.6, can be observed in the following pages.

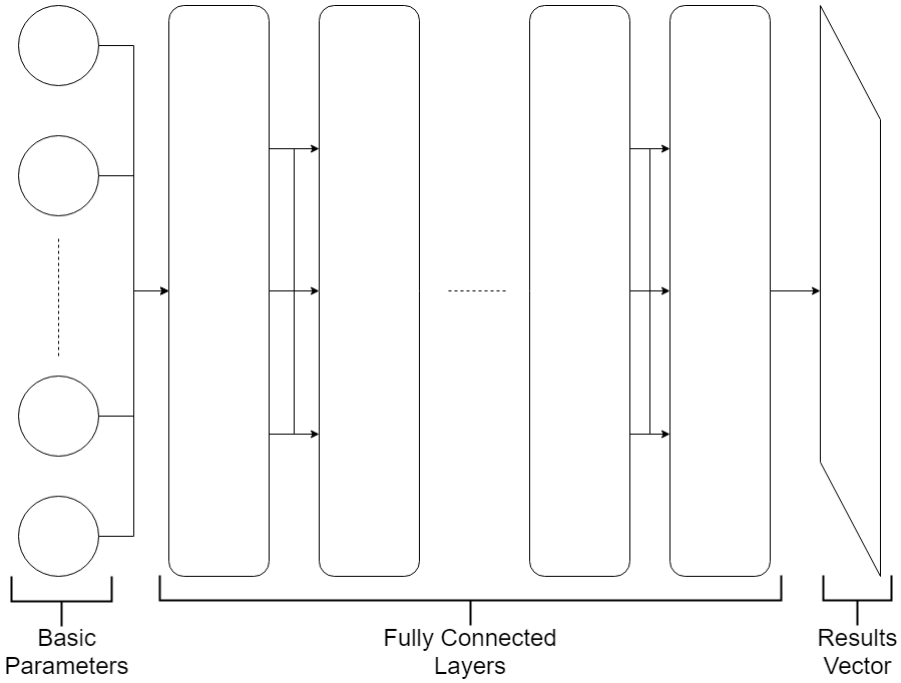


Figure 5.1.: Layout of a neural network using classification, which uses only basic parameters as input and one branch of fully connected layers

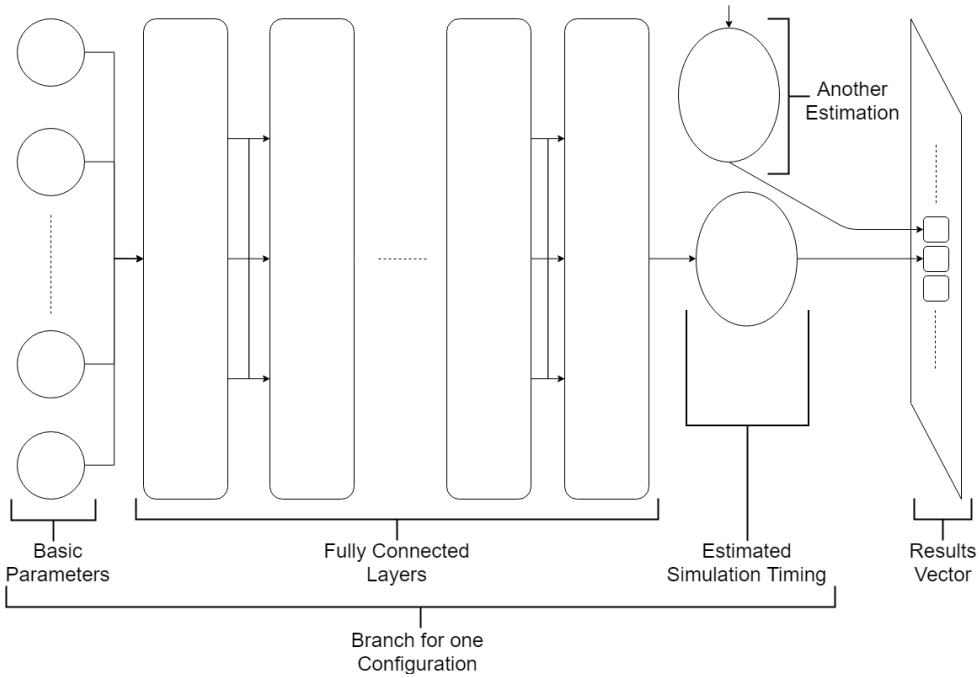


Figure 5.2.: Layout of a neural network using regression, which uses only basic parameters as input and one branch of fully connected layers

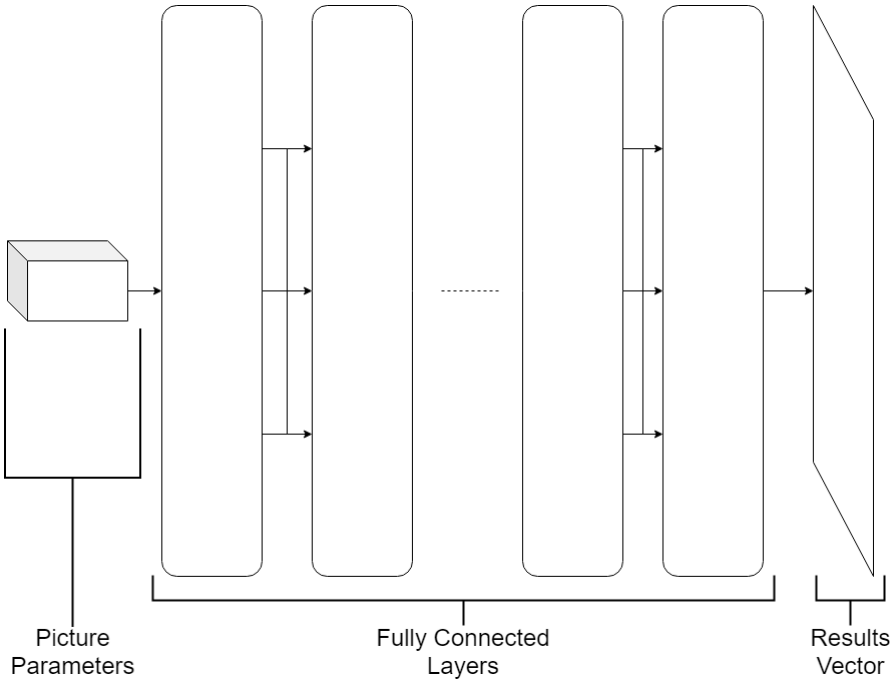


Figure 5.3.: Layout of a neural network using classification, which uses only picture (distribution) parameters as input and one branch of fully connected layers

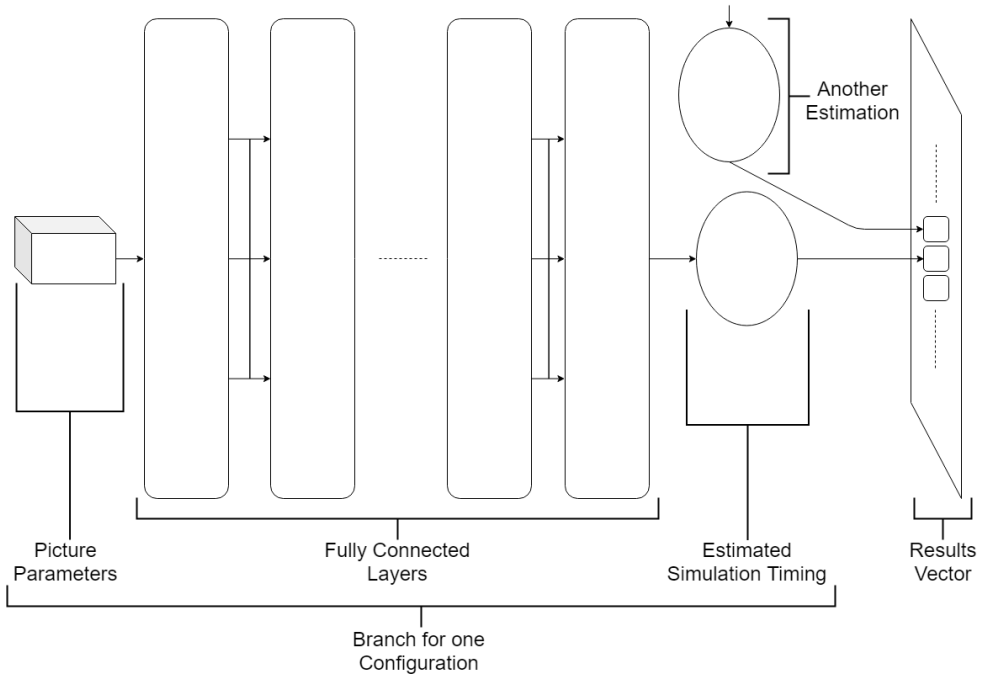


Figure 5.4.: Layout of a neural network using regression, which uses only picture (distribution) parameters as input and one branch of fully connected layers

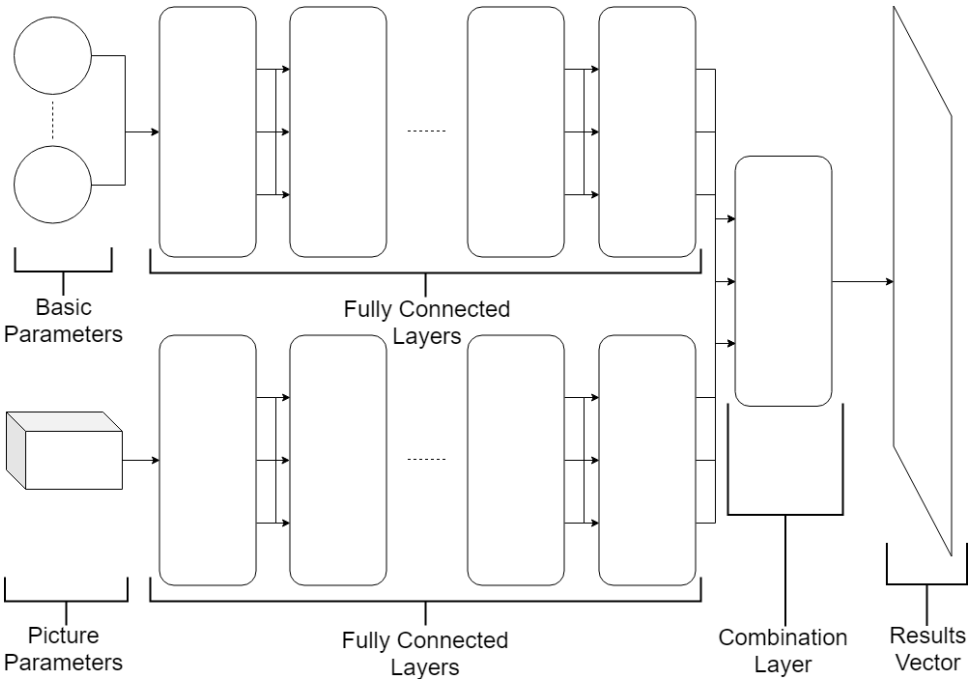


Figure 5.5.: Layout of a neural network using classification, which uses both parameters as input and two branches of fully connected layers

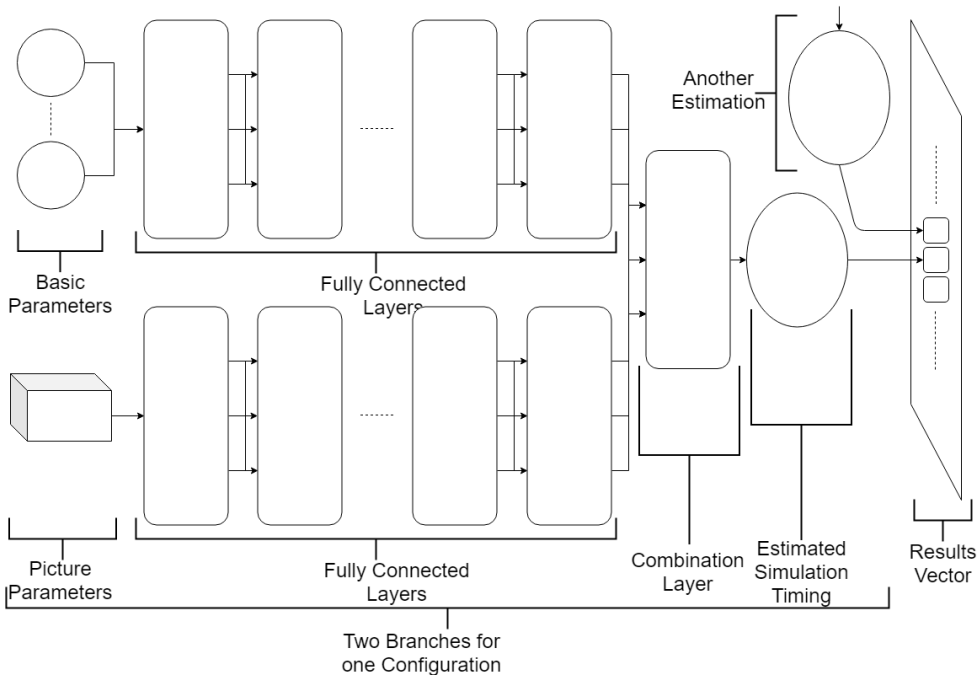


Figure 5.6.: Layout of a neural network using regression, which uses both parameters as input and two branches of fully connected layers

5.4. Choosing Activation Strategies

There are many activation strategies that can be used to define what types of operations each layer uses. The models have been primarily built using rectifiers, which are functions that are defined by their positive counterpart. Relu, or rectified linear units [NH10] are the most popular activation functions in use for the past few years [LBH15], and are also the most commonly used ones in tutorials offered by tensorflow. Experimentation with convolution layers for picture parameters have also been tried for this thesis. Additionally, other activators which are modifications of relu or activators in their own right have been tested. Finally, softmax is used for categorical cross-entropy classification. On the other hand, regression modules require no extra activation at the end to minimize mean squared error. Below, the formulas of all activators and their values in graphs are given.

Note: crelu, or concatenated relu has two parts, a positive part like relu, and a negative part that activates with negative values

$$
\begin{aligned}
relu(x) &:= max(x, 0) \\
softmax(x) &:= \frac{e^x}{\sum e^{x_{other}}} \\
elu(x) &:= \begin{cases} x, & \text{if } x > 0 \\ e^x - 1, & otherwise \end{cases} \\
selu(x)[\text{Kla+17}] &:= \begin{cases} \lambda * x, & \text{if } x > 0 \\ \lambda * \alpha * (e^x - 1), & otherwise \end{cases} \\
softplus(x) &:= \ln(1 + e^x) \\
softsign(x) &:= \frac{x}{1 + |x|} \\
tanh(x) &:= \tanh x \\
sigmoid(x) &:= \frac{1}{1 + e^{-x}} \\
crelu(x) &:= crelu_{pos}(x), crelu_{neg}(x) \\
crelu_{pos}(x) &:= relu(x) \\
crelu_{neg}(x) &:= min(x, 0) \\
leaky\_relu(x) &:= max(x, 0.1 * x) \\
relu6(x) &:= max(min(x, 6), 0)
\end{aligned}
$$

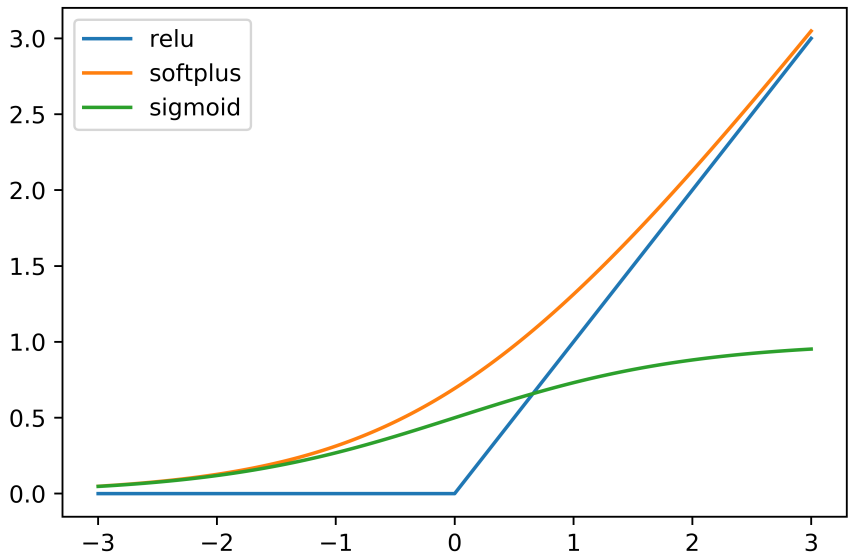


Figure 5.7.: Comparison of activation strategies

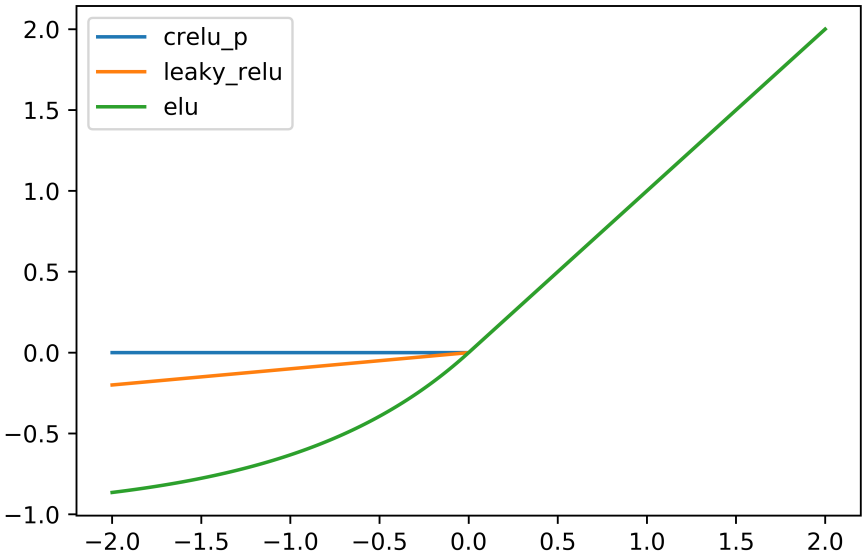


Figure 5.8.: Comparison of activation strategies

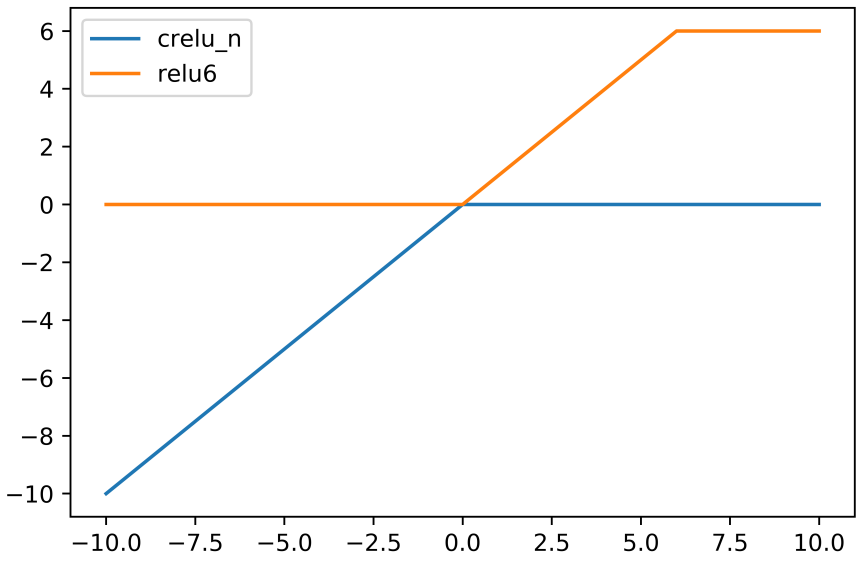


Figure 5.9.: Comparison of activation strategies

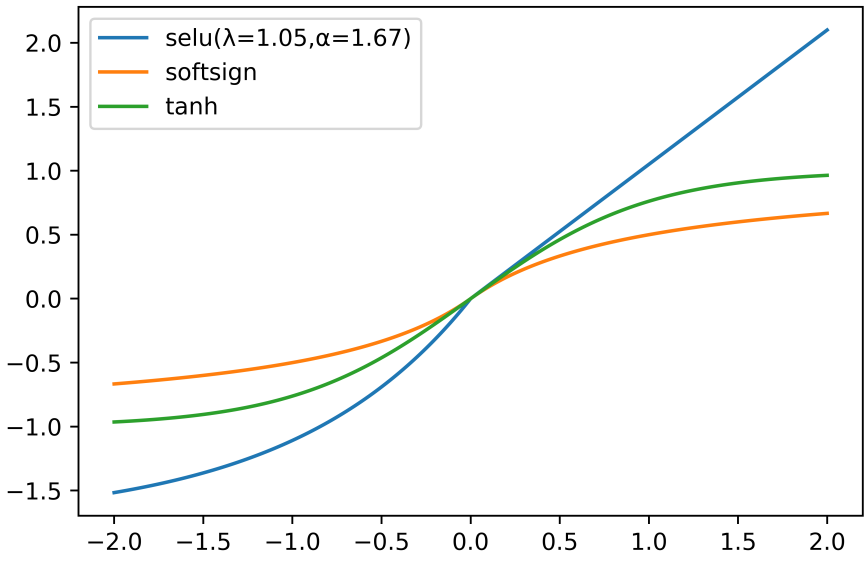


Figure 5.10.: Comparison of activation strategies

6. Model Modification

6.1. Number of Layers and Nodes

The first possible modification for the model is of course changing the number of layers and how many nodes there are per layer. Sources show that countless experiments have yielded some rules of thumb which have been tested by the author, namely that having one hidden layer is enough and the number of nodes of that said layer should be between the number of input nodes and output nodes [htta].

6.2. Learning Rate and Optimizers

Learning rate and the choice of optimizer are important hyper-parameters that effect how effective a model may learn from given data. Learning rate determines how a model reacts to error and corrects its values. A bigger learning rate allows the model to choose a configuration for itself quicker; however, this configuration is also probably sub-optimal. On the other hand, a small learning rate leads to longer learning times, perhaps even causing the model to never fully learn. [Zul18]
Optimizers are processes that minimize or maximize a target function. There are many methods to achieve this, such as stochastic gradient descent (SGD) or adaptive moment estimation (adam). The results of using different optimizers have been reported later in this thesis, under Section 9.4.

6.3. Normalization Strategies

Four methods for normalization have been used, two of which are based on Min-Max Feature scaling, one on standard score and the last one is taking the logarithm of the values.

$$
\begin{aligned}
Normalization1 : X' &= \frac{X}{X_{max}} \\
Normalization2 : X' &= \frac{X - X_{min}}{X_{max}} \\
Normalization3 : X' &= \frac{X - \mu}{\sigma} \\
Normalization4 : X' &= \log X
\end{aligned}
$$

Part III.

AutoPas Integration

7. AutoPas changes and goals

AutoPas did not support integration with a machine learning model. For this purpose, strategies were written in a generic fashion to access parameters from the simulation, a module was built to read configuration files, a method was added to create pictures (distributions) of the domain, with pixels representing sections of the domain and their numbers corresponding to how many particles there are for a given pixel. Bash scripts were used to generate and parse data B. A library called frugally deep [Dob19] was also used to integrate the model generated from tensorflow seamlessly.
Currently, it is possible to use a configuration file and the trained model under the strategy called MachineSearch. The strategy consults the model at the start of each tuning phase and creates the suggested configurations by using the configuration file. The ML model can correctly identify the optimal configuration within five predictions with a probability of over 99%; hence, the strategy MachineSearch tests five most likely configurations during tuning and the simulation is run using the most efficient one until the next tuning phase.

8. Python Notebooks for Machine Learning

Machine learning is done using Tensorflow and Keras. Keras is a high level API that is built on top of TensorFlow. Because Tensorflow has inherited the API of Keras, it is possible to simply use tensorflow.keras to use utilities as defined by Keras. The workbooks written by the author were tested on tensorflow 1.13.1 and 1.14.0. Windows with Anaconda was used to set up the environment required by tensorflow, using the command *conda create -n tensorflow_env tensorflow*. Then it was possible to start ipython notebooks, which provide an effective environment to use the machine learning tools provided by tensorflow. The workflow is defined as follows: first the necessary libraries are imported, namely tensorflow, numpy, and if one wants to create graphs, also pyplot. Then, the user is responsible for giving the path to the experiment results file and the number of the parameters (with each picture pixel counting as an additional parameter). It is assumed that the experiment results are a proper csv file (for example created using the provided parser) and that everything after parameters are an ordered list of simulation times. The data is copied inside the notebook to avoid rereading the csv file again. Then, the data is shuffled, and a default percentage of the data is put away as a testing set. Later, when the model attempts to fit the data, it will split the training set into a training set and validation set on its own. Then, the user chooses a normalization strategy for the data, which is applied before it is received by the model. The default normalization strategies are the ones to be found most effective by the author. Afterwards, it is shown how the single data lines have been normalized and what percentage of the labels belong to which configuration. This is an important metric, as a model is usually biased towards configurations that come more often. Then the model is built, either using keras.Sequential which is a quicker way to build a model or layer by layer which allows mixed data types to be handled in different branches and combined by a following layer. The model is compiled using an optimizer, a loss function and a metric. Then, model.fit is called to fit the data into the compiled model. Finally, the rest of the notebook shows how accurate the fitted model was. Due to the way the notebook is built, one can quickly change hyper-parameters, data, or model layouts to quickly test out other methods that might bring better results.

Part IV.

Results and Comparison of the Models

9. Overview of Results

9.1. General Overview

The tests were done on the linux cluster provided by LRZ, namely CoolMuc-2. The maximum possible amount of 28 cores were used during the experiments. Each experiment batch was given three hours to complete, and the experiments were done in batches of up to forty. A Gaussian particle generator provided by the AutoPas library was used to generate the experiments. For each experiment batch the number of particles, cutoff radius and verlet skin radius were permuted. The values that changed between batches include box length, standard deviation, mean, and the range of the particle count. Maximum possible experiments with possible configurations for a simulation are shown below.

box length	4, 8, 12, 16, 20, 32, 64, 96, 128, 256
standard deviation	box / 4, box / 2, box * 3 / 4, box
mean	0, box / 4, box / 2, box * 3 / 4, box
particles per dim	11-30, 31-40, 41-45, 46-50 (Total of 40)
cutoff radius	1, 2, 3, 4
verlet skin radius	1, 2, 3
possible experiments	96000

Table 9.1.: Particles per dim must be multiplied with its value three times

The following tables give an overview of how efficient each model was. The occurrence counter is a basic estimator that returns in order the most occurring experiments. This estimator is used as a base line to see how effective the other methods are. Five values are shown in each cell of the table, which stand for how much of the test data has been predicted correctly.

Best Results Of	Occurrence Counter	Classification	Regression
Basic Parameters	42.4%/72.0% /78.0%/83.3%/88.3	78.1%/92% /96.6%/98.4%/99.3%	69.0%/86.7% /91.2%/92.8%/94.6%
Picture Parameters		42.4%/72.2% /80.6%/86.4%90.9%	42.4%/48.7% /55.1%/69.5%/77.8%
Combined Parameters		79.2%/93.2% /97.2%/98.6%/99.2%	69.5%/86.6% /90.8%/92.7%/94.2

Table 9.2.: Best results of different methods and parameter sets, with cumulative percentages of correct predictions given (i.e. first percentage is correct in one prediction, second percentage is correct in first two predictions, and so forth)

As test results show, classification is much more accurate than regression, regardless of which parameter set is used. This is probably due to the need of using log normalized values for experiment durations, which can vary wildly with very small differences. Classification is superior even with much shorter training times, which can be a few minutes for classification compared to almost half an hour for regression. Picture parameters are insufficient for both models on their own, as shown by low rates of accuracy with classification and regression. On the other hand, combined parameters can slightly outperform basic parameters, which means picture parameters can be used as supporting values. A final disadvantage of the classification method is the need to correctly repeat experiments and going through complicated processes to combine new testing data with old ones, especially if new configurations were added. On the other hand, the regression model trains the label for each configuration separately. This means, the same experiments for new configurations don't have to be repeated exactly, which can be cumbersome with randomized experiments. Instead, each configuration has its own model, that is combined to a super model, so new configurations can be trained with their own data, as long as the parameters they use are the same.

ID	Container	Traversal	Data Layout	Newton 3
0	VerletListsCells	verlet-c01	AoS	disabled
1	VerletListsCells	verlet-c18	AoS	disabled
2	VerletListsCells	verlet-c18	AoS	enabled
3	VerletListsCells	verlet-sliced	AoS	disabled
4	VerletListsCells	verlet-sliced	AoS	enabled
5	VerletLists	verlet-lists	SoA	disabled
6	VerletLists	verlet-lists	SoA	enabled
7	LinkedCells	c08	SoA	disabled
8	LinkedCells	sliced	AoS	disabled
9	LinkedCells	c08	SoA	enabled
10	LinkedCells	c08	AoS	disabled
11	LinkedCells	sliced	SoA	enabled
12	LinkedCells	c08	AoS	enabled
13	LinkedCells	sliced	SoA	disabled
14	LinkedCells	c18	AoS	disabled
15	LinkedCells	c18	SoA	enabled
16	LinkedCells	sliced	AoS	enabled
17	LinkedCells	c18	SoA	disabled
18	LinkedCells	c01	AoS	disabled
19	LinkedCells	c18	AoS	enabled
20	LinkedCells	c01	SoA	disabled
21	VerletLists	verlet-lists	AoS	enabled
22	VerletLists	verlet-lists	AoS	disabled

Table 9.3.: The configuration of each experiment with their ID

When looked at individual experiments, it can be seen that the configurations Container: VerletListsCells , Traversal: verlet-sliced , Data Layout: Array-of-Structures , Newton 3: enabled, which make about 42 percent of 81542 experiments, and Container: LinkedCells , Traversal: c01 , Data Layout: Structure-of-Arrays , Newton 3: disabled, which equates to an additional 30 percent, are the most optimal two configurations for the chosen training and test data sets.

ID	Count	ID	Count	ID	Count
0	4066	1	2771	2	4742
3	4312	4	34431	5	0
6	0	7	652	8	13
9	28	10	16	11	255
12	1065	13	411	14	0
15	34	16	10	17	77
18	1	19	33	20	24520
21	3218	22	887	ALL	81542

Table 9.4.: Number of times each configuration was the most optimal

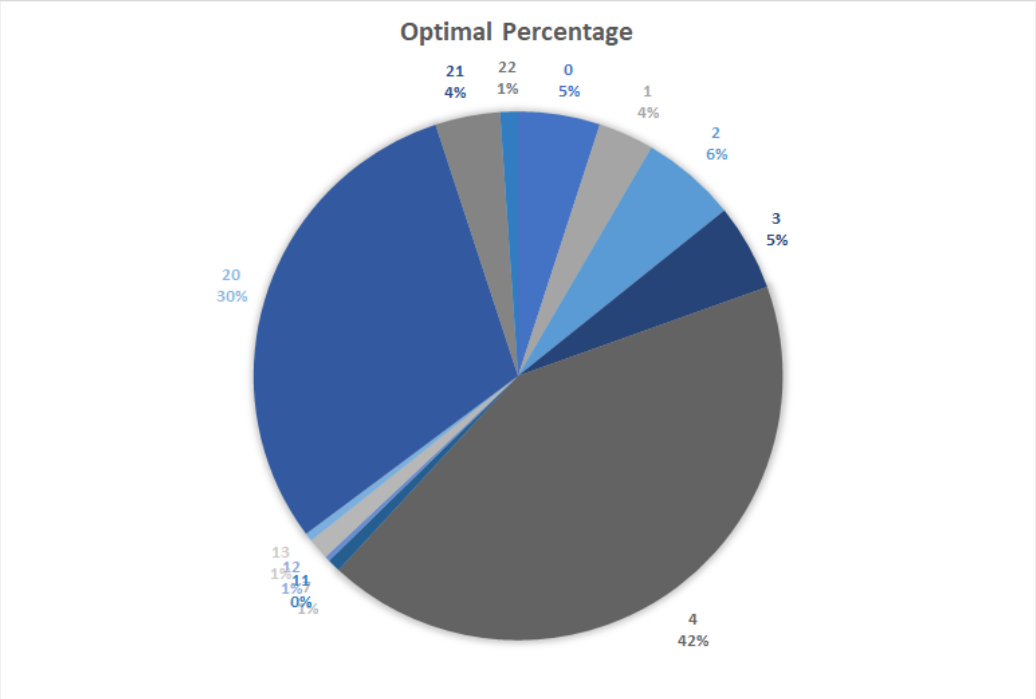


Figure 9.1.: Experiment distribution as a percentage of all experiments

9.2. Correlations of variables before machine learning

In this section, it will be shown how each variable is connected to a configuration for the whole dataset (i.e. including training and testing data sets). First, the parameters will be

compared with how likely they are to belong to a certain class. Only the most common seven classes (plus rest) will be discussed, which will cover at least 95 percent of results. In the following figures, two ways to properly portray the data has been used. One method is to analyze different attributes individually (by comparing configurations that are the same except one attribute) and the other method is to do this process as a whole (by grouping all configurations by one attribute). The following graphs compare performance results of configurations that have only one differing attribute for data layout and whether newton 3 was enabled. Traversals and containers are difficult to analyze this way because few comparable experiments exist and there are multiple containers and traversals, which is why only more general graphs for these two attributes have been created. It is important to note that the more general graphs have differing results compared to configuration by configuration comparison, because they take into account all configurations, irrespective of whether a comparable setting exists.

In Figure 9.2, it can be seen that increasing the verlet skin radius is correlated with a few classes. The 20th class, which uses linked cells and doesn't depend on skin radius, has a higher testing accuracy as the verlet skin radius expands. Classes 0 to 4 all use verlet lists cells and perform worse as the skin radius increases. The skin radius must always be larger than twice the longest length of particle movement within a time-step. A higher skin radius always leads to more computations because more particles have to be compared; however, in two edge cases, a higher skin radius can increase performance. The first case is that cells can be bigger thanks to a higher skin radius, which means they will be able to hold more particles and lead to better vectorization. The second case is when the density of an experiment is relatively low (huge box length and few particles), which leads to a scenario where iterating through cells is computationally more expensive than iterating through particle pairs.

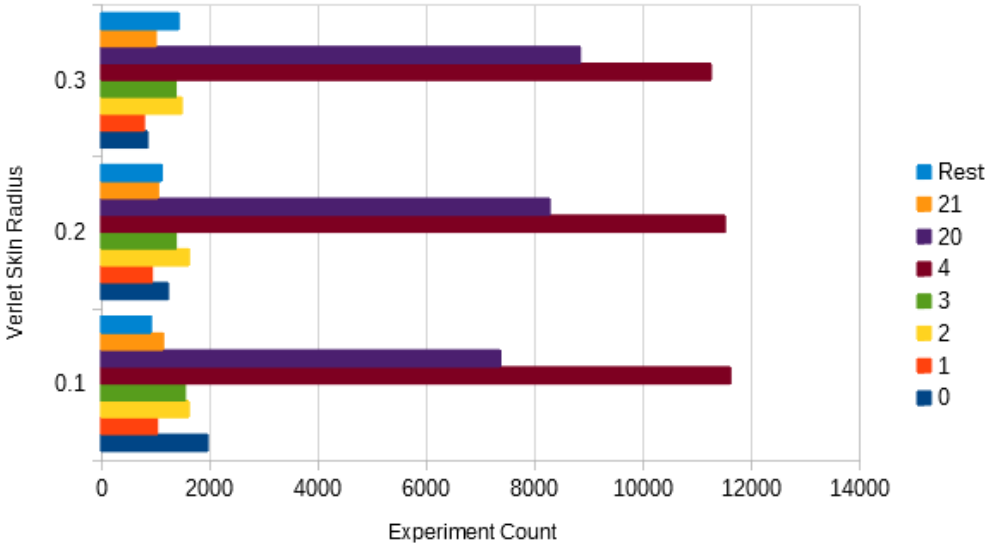


Figure 9.2.: The quickest configurations grouped by verlet skin radii

It can be read from Figure 9.3 that higher cutoff radius means that more computations must be done and parallelization will be harder, because each particle must interact with more particles, i.e. further ones that were once not inside the interaction range. The configurations that do not use Newton3, such as 0 and 20, have increased their percentage of being the quickest experiments. On the other hand, 2, 4 and 21 have all continually fallen as the cutoff radius has increased.

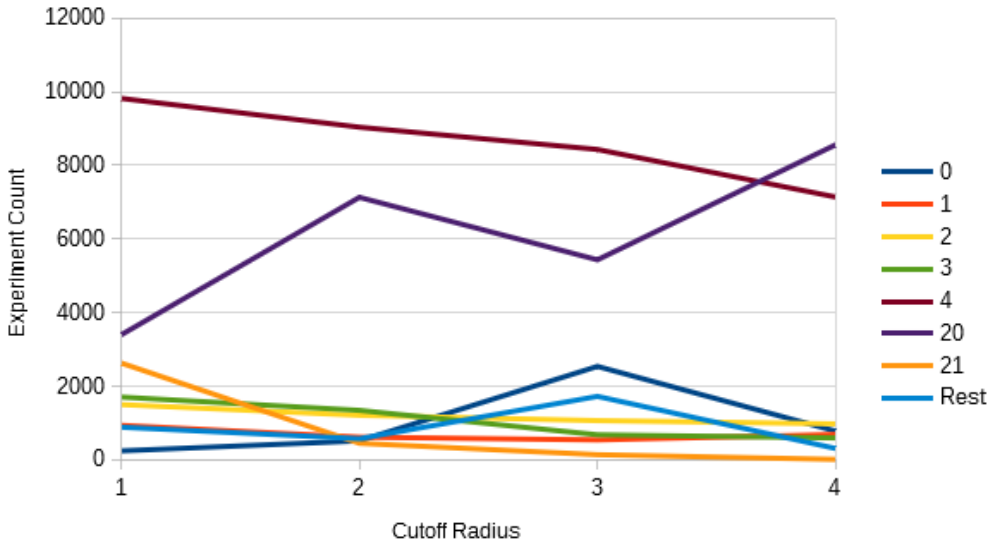


Figure 9.3.: Cutoff radius given in x axis and how many times each configuration is quickest for the given cutoff radius

Although low box lengths until 16 have a mostly mixed distribution, where the category of rest is relatively high and the 20th configuration shows increasing its rate of being the quickest among configurations, it can be seen clearly in Figure 9.4 after box length 16 until 128 that configurations 0, 1, 2 and 20 are falling, 4 is rising quickly, 3 and 21 are increasing their shares slightly. On the other hand, after box length 128 it can be seen that the 21st configuration becomes much more efficient, overtaking the experiment share of the 4th configuration. The 20th configuration uses a type of traversal for linked cells called c01 that is very effective for dense environments, hence increasing the box length dilutes the particles and removes the advantage of 20. On the other hand, 3, 4 and 21 all use verlet structures, which are better in environments where the environment isn't as dense. 4 and 21 both use array of structures and newton3; hence, for bigger environments with not so high density of particles, enabling these settings increases the simulation performance.

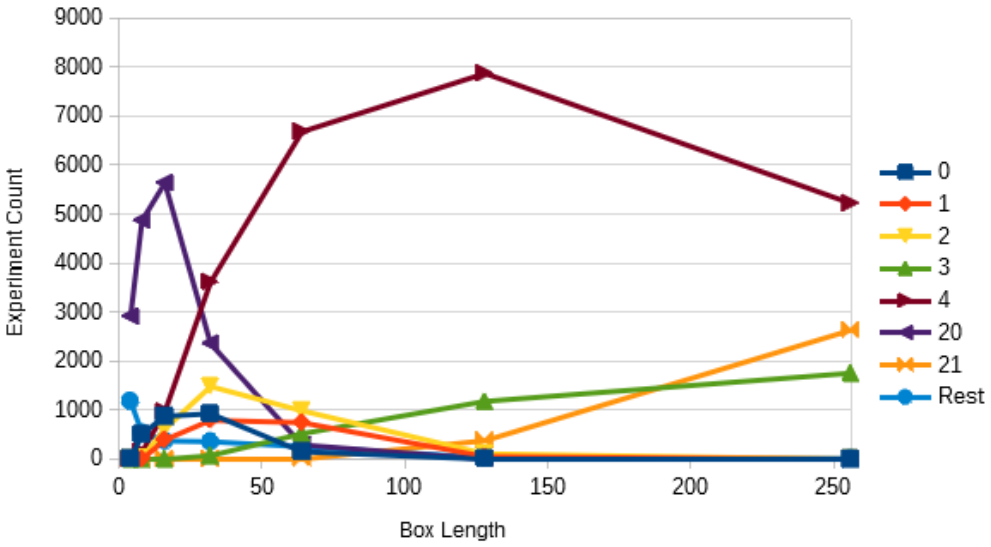


Figure 9.4.: How box length influences which experiment is quickest

Due to the way the experiments were structured, there are less experiments that have high number of particles (between 41-50 particles per dimension) and more experiments that have between 11-20 and 21-30 particles per dimension. This is because some traversals would overload the computation center with over the limit memory consumption. Sometimes, the reason was due to a set time limit of three hours; though, this was less often the case. This weakness of the data set has been noted in the possible improvements part of the thesis.

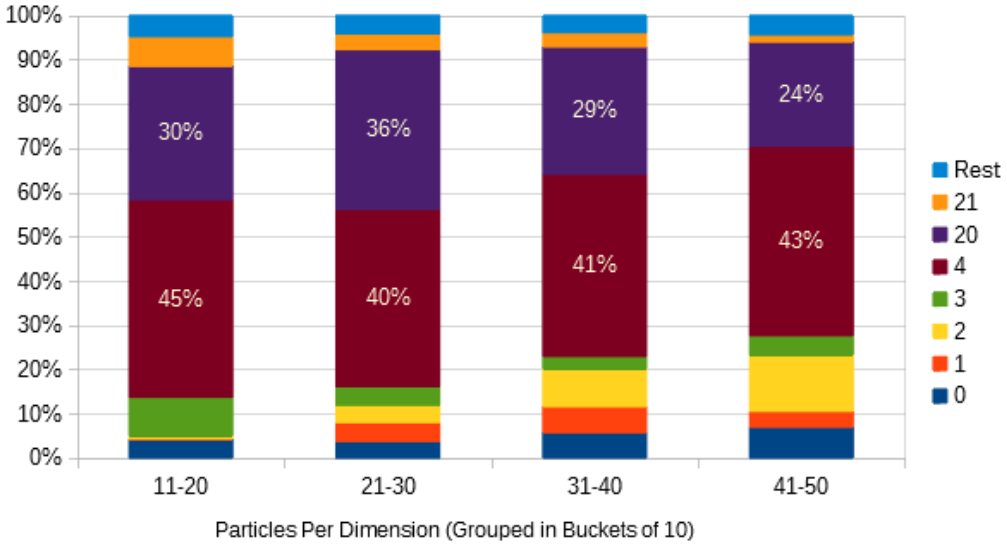


Figure 9.5.: Quickest experiment classes grouped by number of particles per dimension

In Figure 9.5, almost no clear patterns can be seen, as configurations are almost never continuously worse or better. If one ignores the first bucket of particles per dimension between 11 and 20, it an be seen that the fourth configuration slightly improves its standing as the number of particles increases, while the 20th configuration get worse quite quickly. Configurations 0 and 2 continually get better, while configuration 21 gets slightly worse. The rest of the classes are mostly not correlated with the particle count parameter. An explanation could be that the number of particles on its own is a bad parameter to consider, when trying to figure out which configuration will be the quickest. This is because the number of particles is not equally distributed and doesn't have much meaning on its own without the box length. Because the experiments are done with 28 cores and the parallelization is a bigger issue than a lack of resources, adding more things to calculate slows most configurations equally.

In the following figures, i.e. 9.6 and 9.7, the relative accuracy of two configurations with certain attributes are shown. For example, the 8th configuration uses array of structures and the 13th configuration uses structure of arrays, and the 8th configuration was in about 35000 experiments faster and in about 45000 experiments slower. By taking the average of these values over configuration comparisons, an average value is found that reflects in how many cases, all else being equal, is aos faster and in how many cases is it slower. As it can be seen from the graph, aos is slightly quicker in two cases, somewhat slower in 2 cases and much quicker in 3 cases.

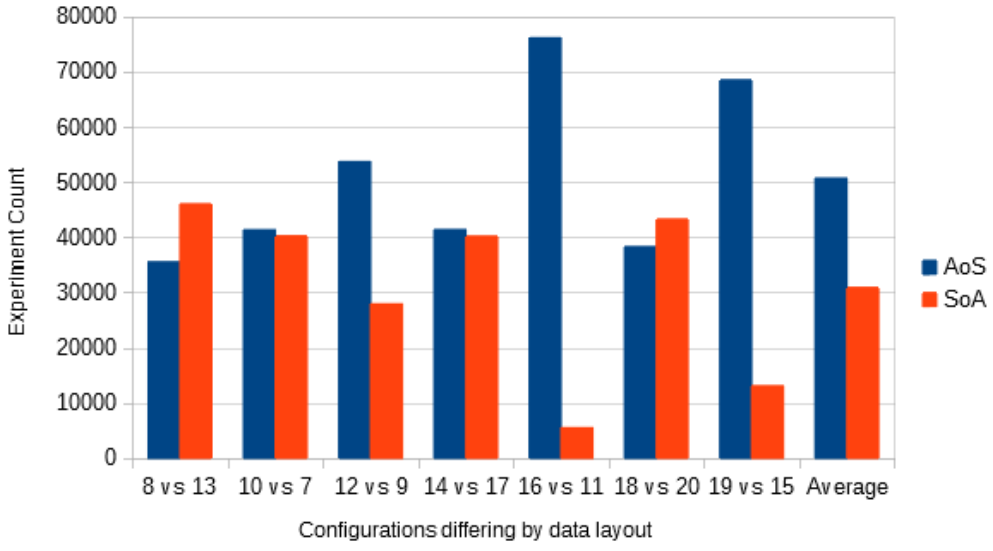


Figure 9.6.: Performance comparisons of experiments that only differ in data layout

On the other hand, the difference between using and not using Newton 3 is not as clear cut. Enabling Newton 3 leads to a massive positive difference in three experiments, a significant negative difference in another three, a significant positive difference in one. The rest can be seen on the graph. In the end, enabling Newton 3 does increase the chance of having a

simulation that runs quicker.

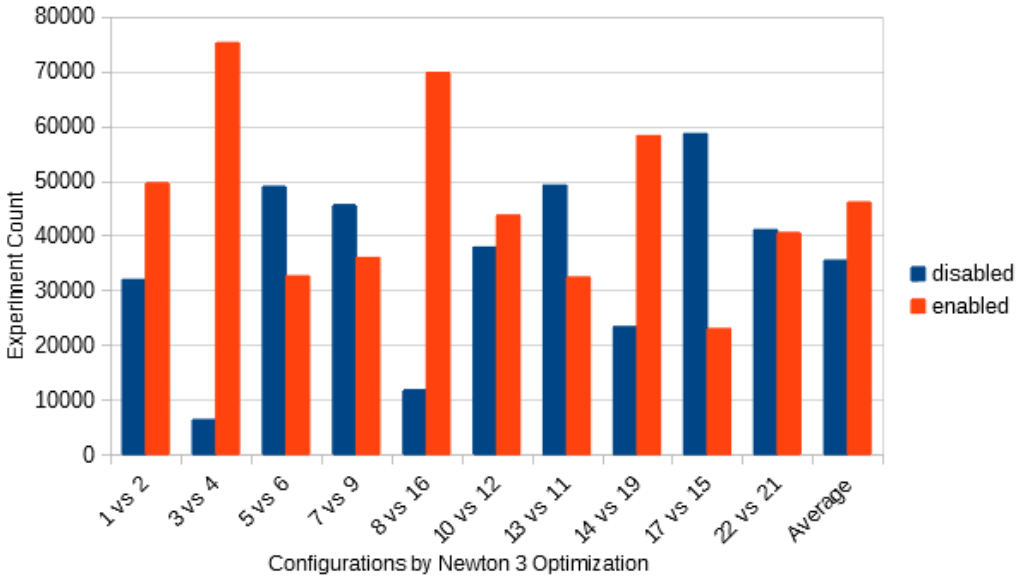


Figure 9.7.: Performance comparisons of experiments that only differ in whether Newton 3 is enabled

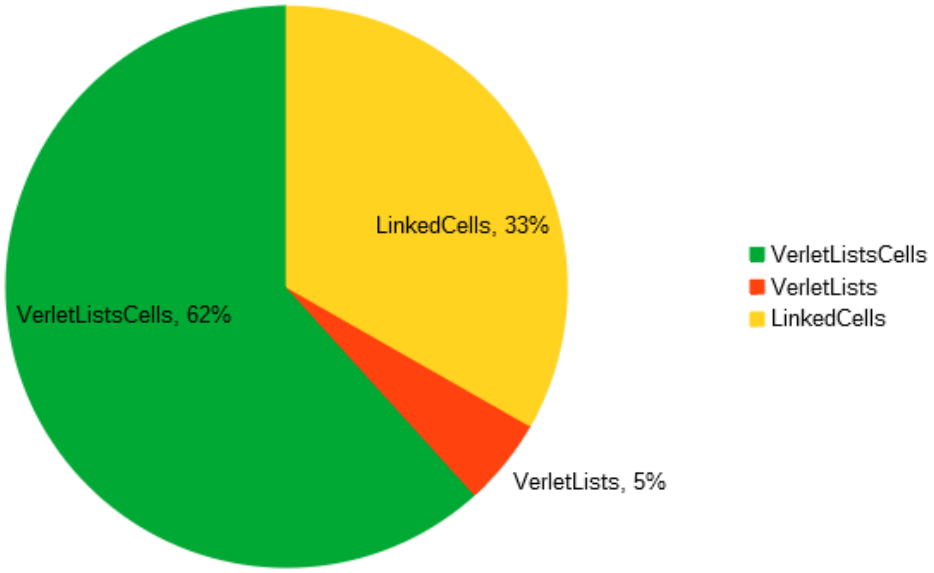


Figure 9.8.: Share of experiments for which container type was the quickest

Due to always performing worse than other settings, direct sum was removed from the dataset, because training times were magnitudes longer. It can be seen that verlet lists cells and verlet lists outperform linked cells most of the time. This is because verlet lists use an implementation that lowers the number of interactions by making sure only particles that

are somewhat close to each other are compared instead of all particles that can be found inside a neighbor. This speedup can only be achieved if there are not too many particles inside neighbors that already are within the interaction range of a given particle.

It can be seen that for the given experiments, the traversal type verlet-sliced and c01 are the most effective ones, which are used and mainly bolstered by configurations 4 and 20, respectively. Traversal type is one of the main contributors to whether a given configuration will be the quickest one; however, because there are more attributes that influence the performance of an experiment, just predicting the correct traversal type is not enough to estimate which configuration type would be the correct one.

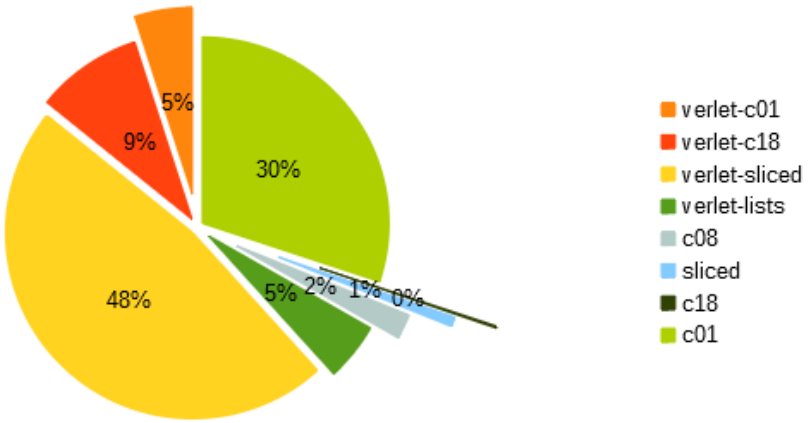


Figure 9.9.: Share of experiments for which traversal type was the quickest

Array of structures, which is a vectorization of particle values such as location and velocity, outperforms structure of arrays in two thirds of all cases. This is because cache efficiency and locality of data can be maintained easier with such a strategy.

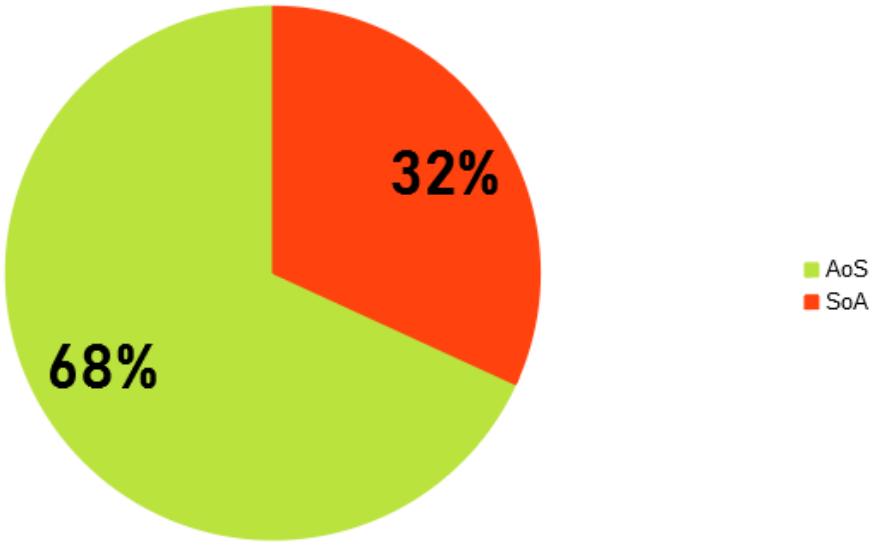


Figure 9.10.: Share of experiments for which data layout was the quickest

Enabling Newton 3, on the other hand, is only 8 percent better than not enabling it. This is probably due to synchronization problems encountered by configurations that were measured in many experiments with relatively low box lengths and high cut off radii. Not being able to utilize all 28 cores can mean lower performance, especially if some configurations utilize less than half of all cores to enable Newton 3.

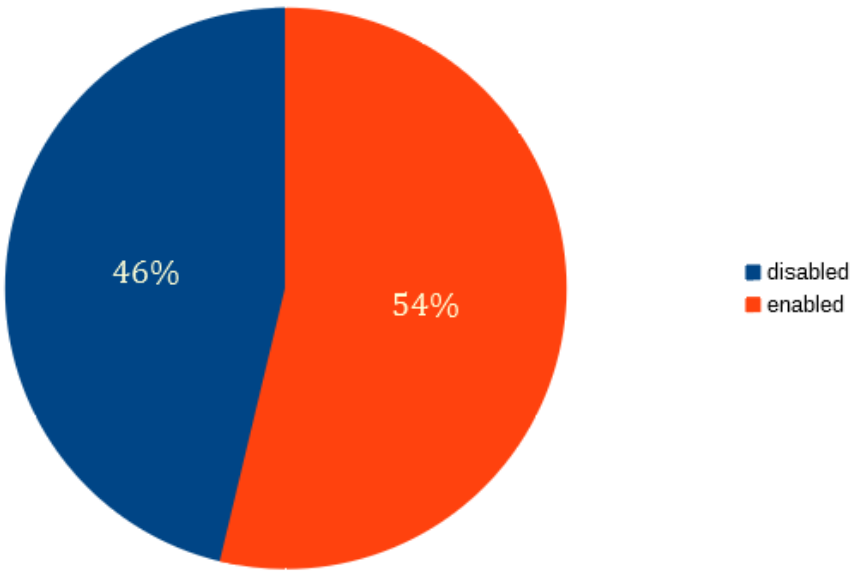


Figure 9.11.: Share of experiments for which whether enabling Newton 3 was the quicker

9.3. Detailed overview of the accuracy of different configurations

In the following figures, the relationship between attributes of an experiment and the accuracy of predictions are shown. For example, Figure 9.12 shows that if an experiment has a higher verlet skin radius, it is more likely that the model's first prediction is correct, but also more likely that the model will not be able to predict correctly by fifth prediction.

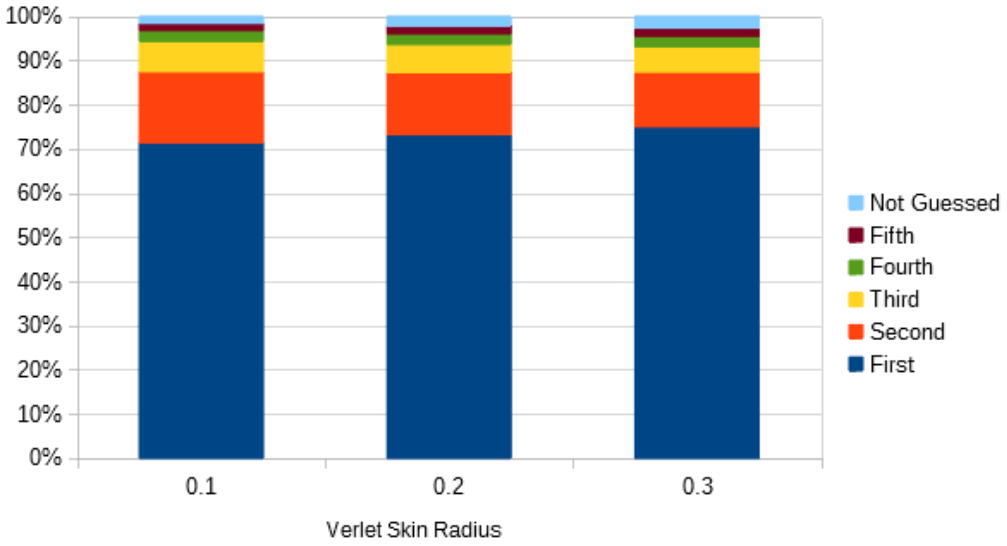


Figure 9.12.: The predictions for configurations grouped by verlet skin radii

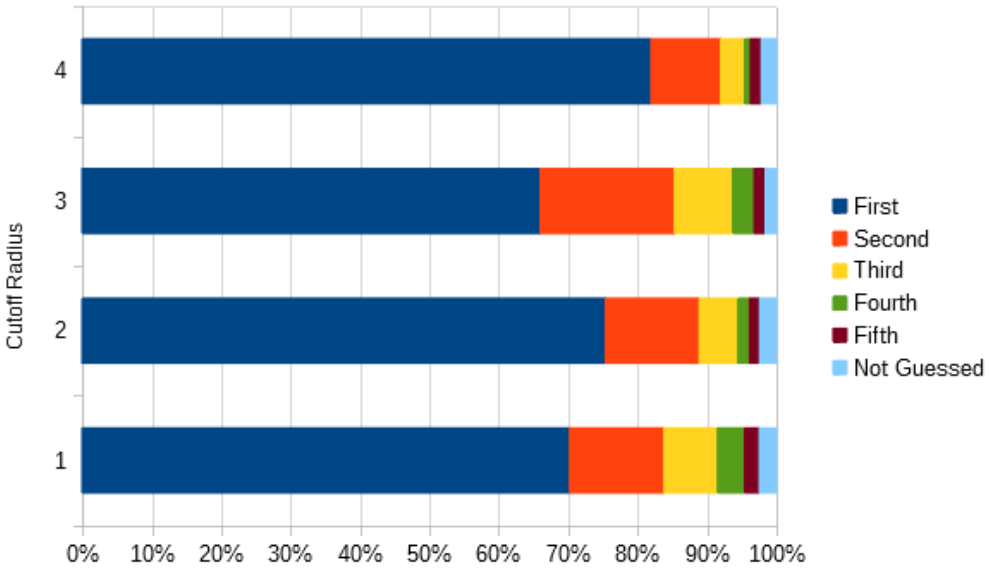


Figure 9.13.: Accumulated predictions of which configuration is quickest for the given cutoff radius

Other significant patterns include, that a cutoff radius of 4 (Figure 9.13) leads to worse estimations after second prediction, that experiments which have box lengths between 16 and 64 are generally harder to optimize for, and increasing the number of particles does not significantly influence how well the estimations of the model are.

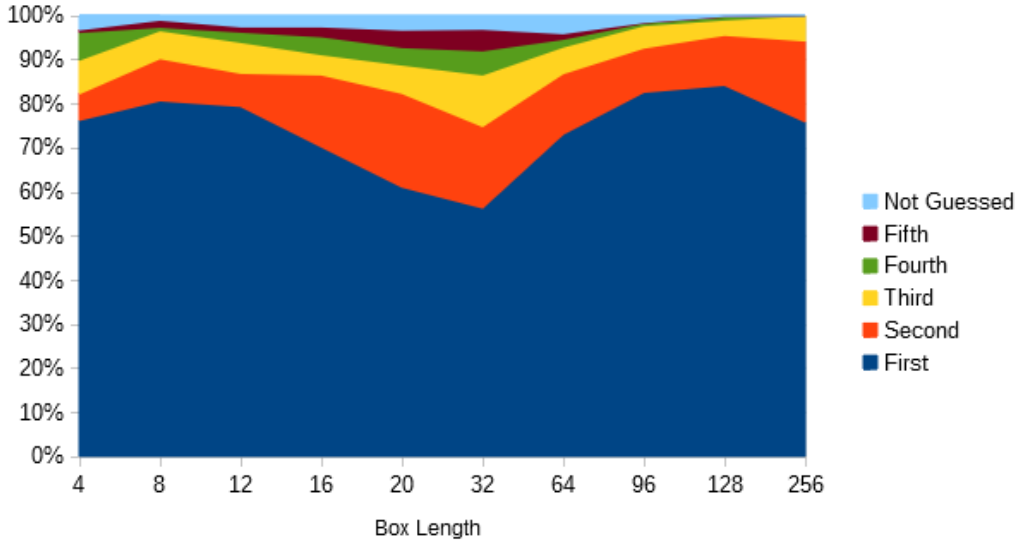


Figure 9.14.: How box length influences which experiment is thought to be quickest

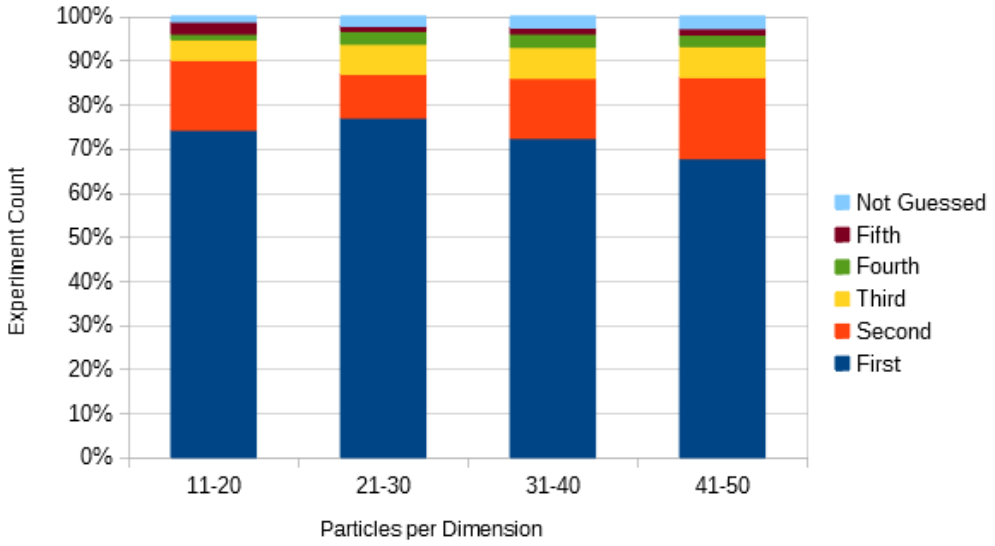
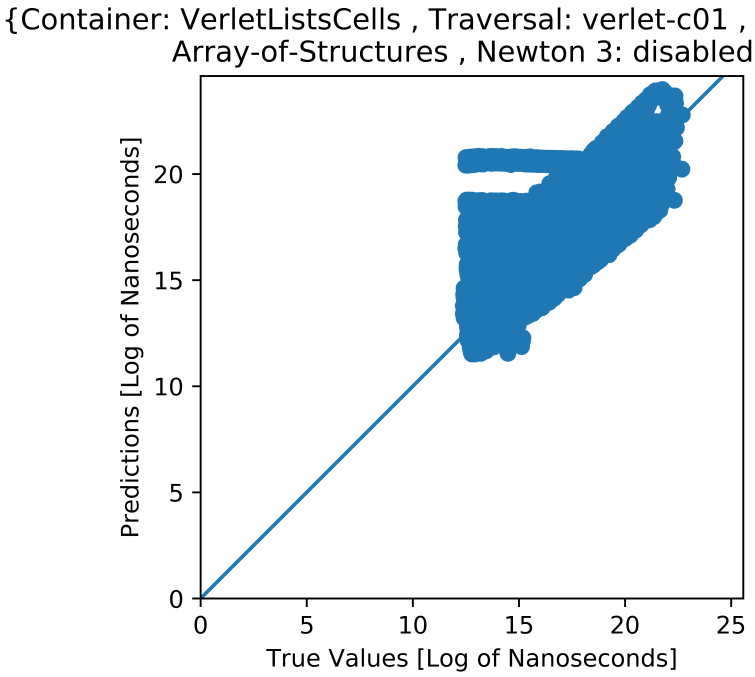
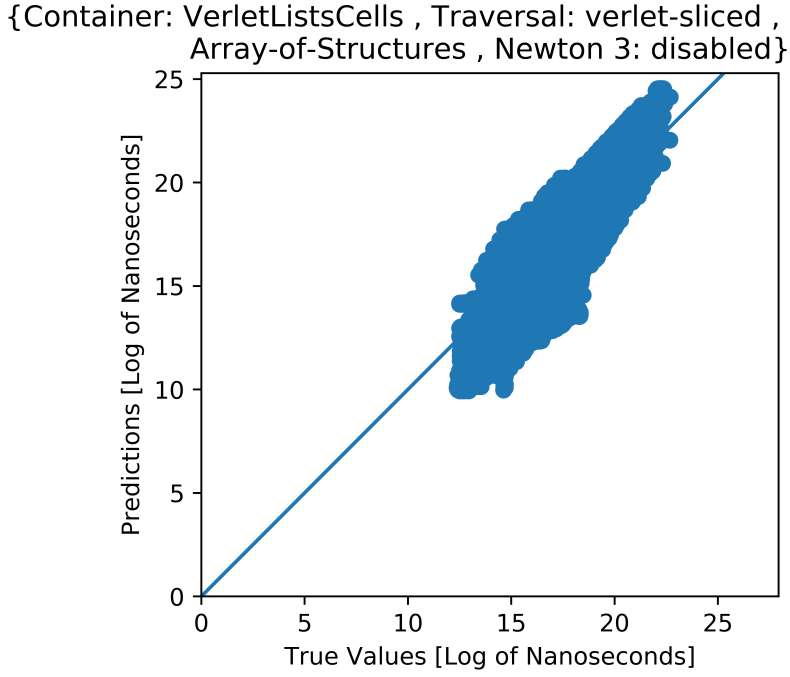


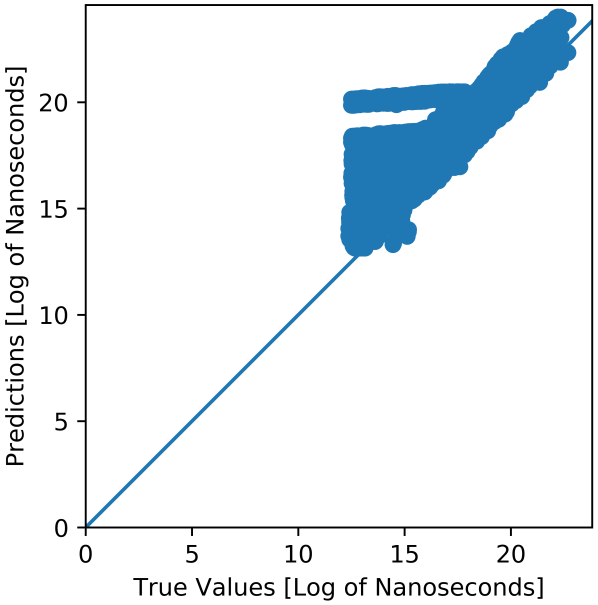
Figure 9.15.: Predictions for quickest experiment classes grouped by particle count ranges of 10

Here, the following figures show how the regression model estimates how long the given configuration will take for all experiments, plotted against their true values, both in log scales. It can be seen that the model often overestimates how long a simulation will take.

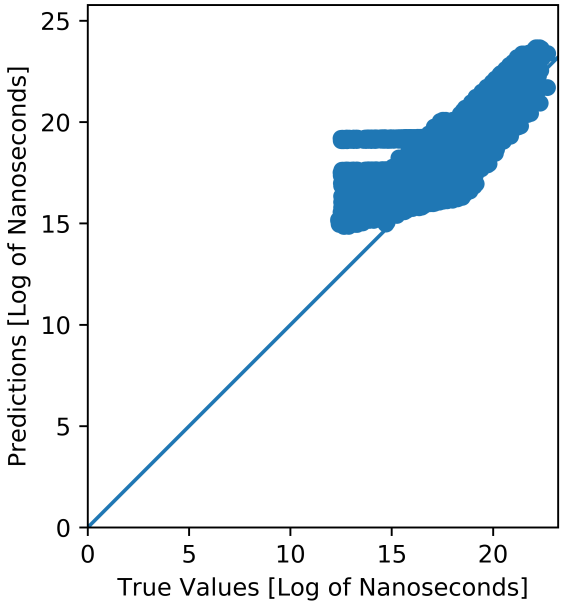




{Container: VerletLists , Traversal: verlet-lists , Data Layout: Structure-of-Arrays , Newton 3: enabled}



{Container: LinkedCells , Traversal: sliced , Data Layout: Structure-of-Arrays , Newton 3: enabled}



9.4. How changing hyperparameters influence the results

As mentioned in model modifications, it is possible to change hyper parameters about the model that may influence how well the data may be learned. For this purpose, a random seed of 42 was set to shuffle the data and the model was trained with different hyperparameters using the first half of the data, a default setting (Adam optimizer, learning rate of 0.001, 3 layers (input layer, one hidden layer, output layer), 16 nodes in intermediary layers, relu as activator for hidden layers, and always softmax for classification, first normalization method), only classification and only the basic four parameters. The median results of each change was noted and is summarized in the following graphics. The default value is marked with a star and each type of experiment was repeated five times. The values for previous sections were collected from the default model, except it was trained for 50 epochs instead of 5. Due to resource constraints, all comparisons here are made with a default value of 5 epochs, unless number of epochs are compared. Median values are calculated first by comparing the testing accuracy of the first predictions, and then first five predictions. If multiple experiments tie for these comparisons, then the overall better performing experiment is chosen as the median value.

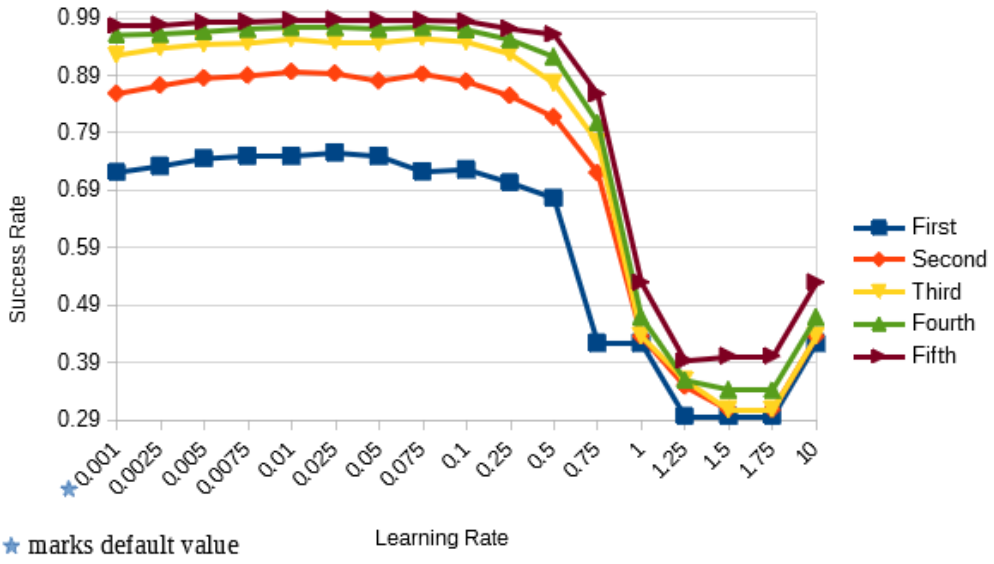


Figure 9.16.: How changing the learning rate influences the testing accuracy for the different number of predictions in a classification model

The default learning rate for Adam Optimizer is 0.001, and the author noted that a higher learning rate was usually more accurate. Figure 9.16 shows that a slight improvement for first prediction can be attained until a rate of 0.025, which quickly falls of for higher learning rates. The default value of 0.001 attains a median testing accuracy of [0.721 0.858 0.925 0.96 0.976] while the best learning rate 0.025 attains [0.755 0.893 0.948 0.974 0.987]. Yet, the rate 0.05, although in all experiments performed worse than 0.025 for first prediction, outperforms 0.025 if the values for first five predictions are compared. The median value for accuracy by fifth prediction of 0.05 is 0.988 and 0.986 for the learning rate 0.025.

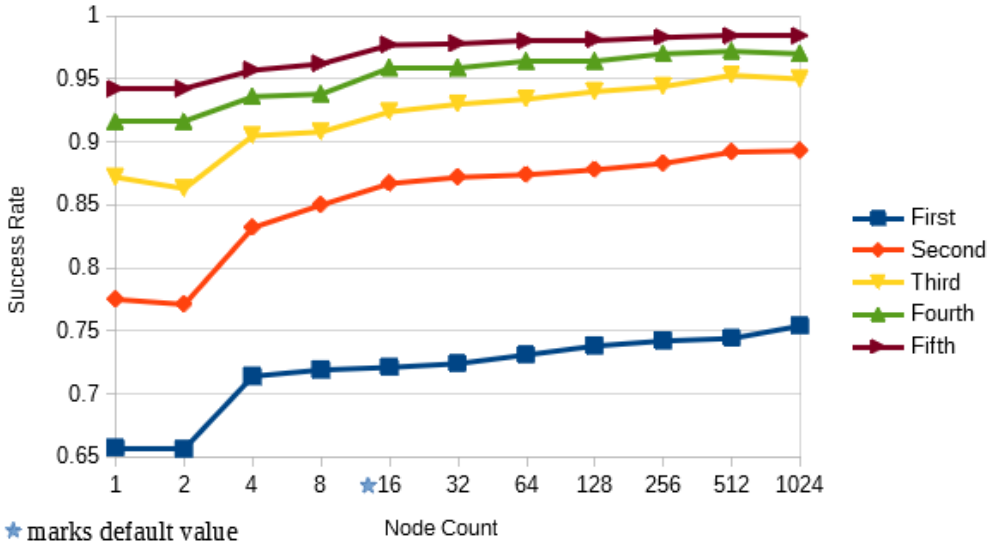


Figure 9.17.: How changing the number of nodes at the intermediary layer influences the testing accuracy for the different number of predictions in a classification model

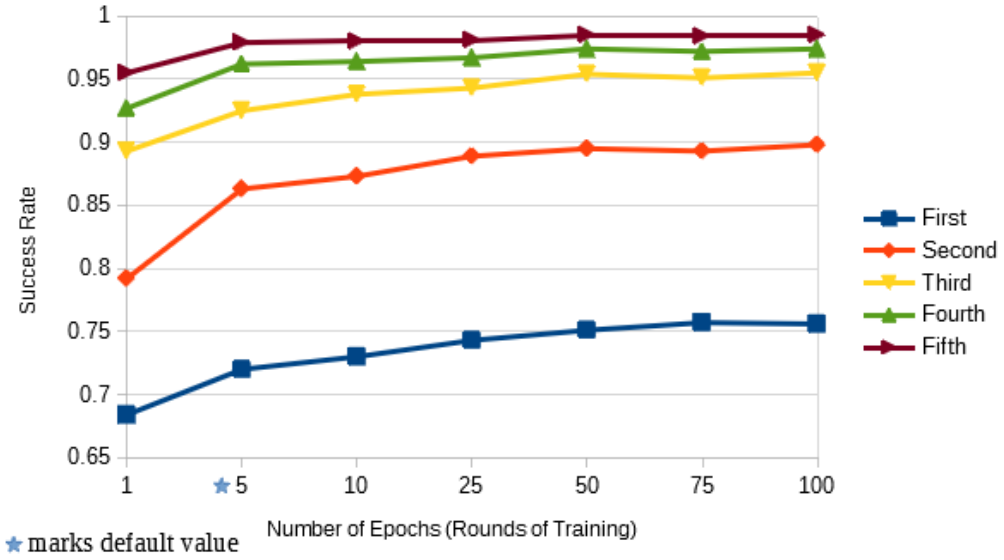


Figure 9.18.: How changing the number of epochs influences the testing accuracy for the different number of predictions in a classification model

It can be seen from Figure 9.17 that higher number of nodes has almost always a positive effect on the testing accuracy. After 512 nodes, the training time however became quite high especially compared to low rates of return. The median value of 512 nodes is only 1 percent worse (74.4% to 75.4%) than 1024 nodes at the first prediction, and by first five predictions, their testing accuracies are equal at 98.4%. Increasing the number of nodes

almost always means that the testing accuracy is higher for all predictions; however, the almost exponential increase in training time makes this change less desirable.

The number of epochs for the classification tutorial provided by tensorflow was only 5, compared to the number of epochs of 1000 for the regression tutorial. For the model used in thesis, regression is also magnitudes more costly to train than classification, because each configuration has to be trained separately to estimate times, which is then compared in a final array. It can be seen in Figure 2.18 that until 50 epochs, there is a steady increase in testing accuracy, which then stabilizes for additional epochs.

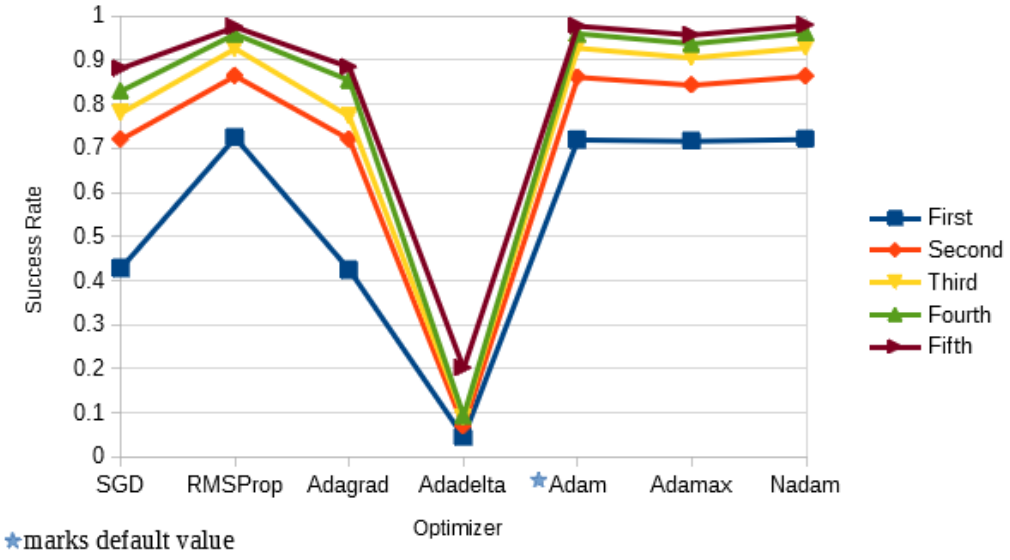


Figure 9.19.: How changing the optimizer influences the testing accuracy for the different number of predictions in a classification model

There are 7 possible optimizers provided by tensorflow, which were all tried out for the default model. SGD is meant for regression models, which performed relatively poorly compared to three Adam variants and RMSProp. The default model, Adam, has a median value of [0.719 0.861 0.927 0.96 0.977] which is only surpassed by Nadam with a median value of [0.72 0.863 0.928 0.962 0.979].

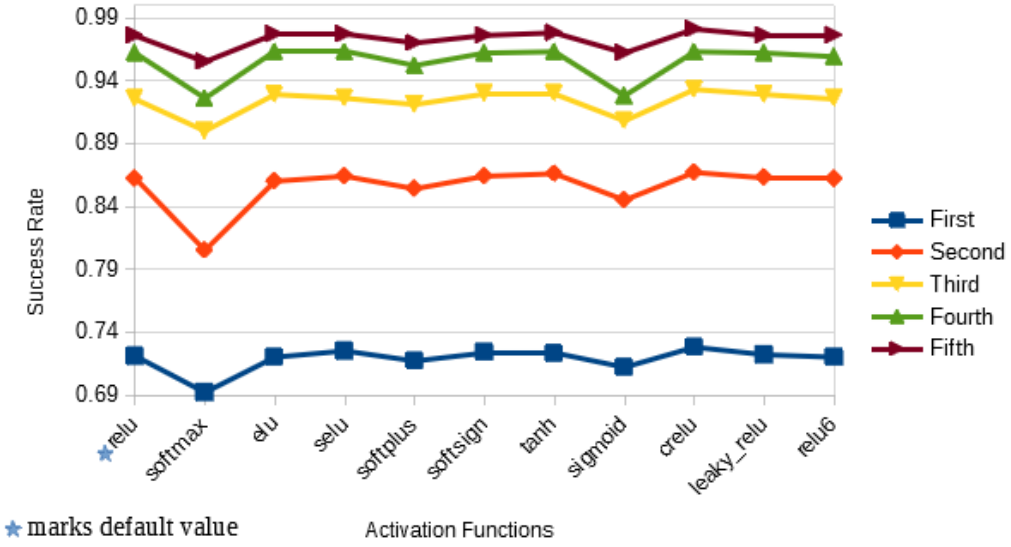


Figure 9.20.: How changing the activation function influences the testing accuracy for the different number of predictions in a classification model

Activation functions also have varying degrees of accuracy, with softmax performing the worst with values [0.692 0.805 0.9 0.926 0.955] and crelu the best with [0.728 0.867 0.933 0.963 0.981].

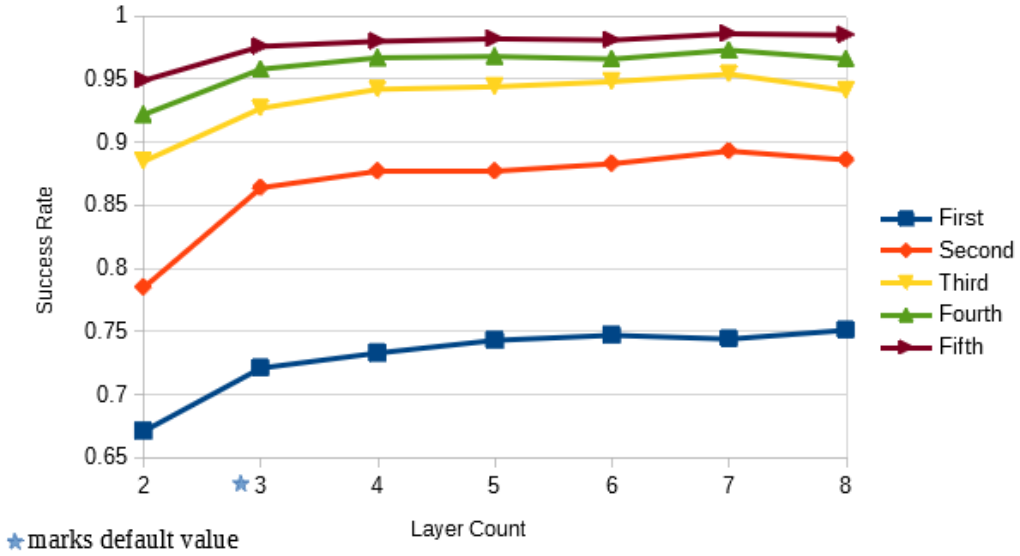


Figure 9.21.: How changing the number of layers influences the testing accuracy for the different number of predictions in a classification model

Increasing the number of layers up until five increases the testing accuracy for all predictions. For the configuration with six layers the value of the fifth prediction drops compared to five

layers, and for the configuration with seven layers the value of the first prediction drops. It can be inferred that with default settings, having more than five layers doesn't impact any testing accuracy more than 1%, and having more layers does not lead to steady gains for the testing accuracy of all number of predictions.

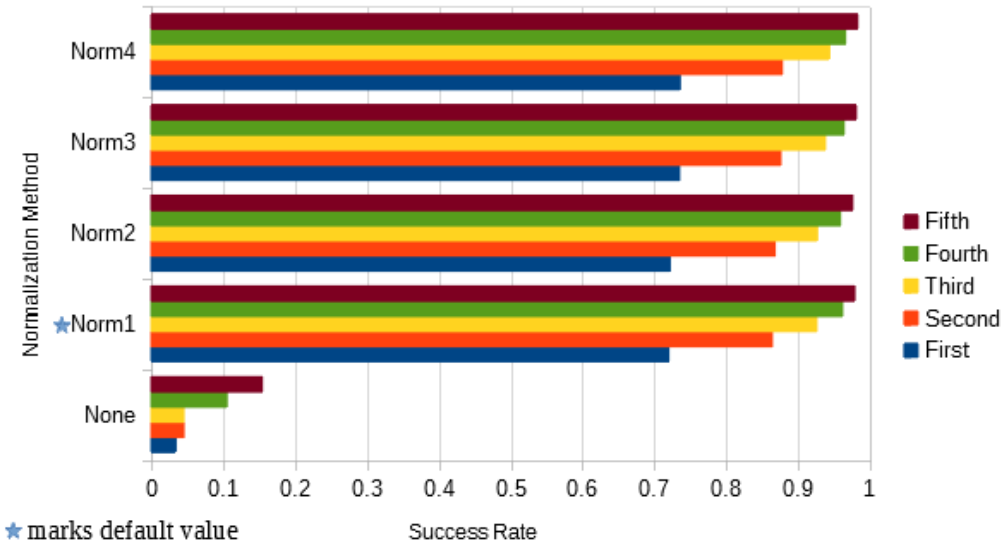


Figure 9.22.: How changing the normalization method influences the testing accuracy for the different number of predictions in a classification model

As the data shows, not normalizing the data is not an option, which is caused by highly varying parameters, such as verlet radius, which is between 0.1 and 0.3, cutoff radius, which is between 1 and 4, and particle count, which can go up to 125000. Surprisingly, the best normalization is not the min-max normalization or standardizing the data but taking the log values of the parameters. This is probably due to number of particles highly influencing the result, and the number of particles increase with a factor of N^3.

9.5. Benefits of using MachineSearch over FullSearch

At the start of testing, FullSearch supported 27 different configurations, 4 of which were DirectSum configurations that almost always under-performed (especially for over 1000 particles). In contrast, MachineSearch chooses 5 configurations, that include the optimal configuration with 99% likelihood. First, 18 tests (not including DirectSum) are avoided, which can take up a lot of time. Calling the ML model happens within few milliseconds, and loading it is also a one-time cost that takes less than few seconds. Looking at predictions that are not optimal, but the best within the five configurations the ML model suggests, it can be seen that the worst case is a sub-optimal configuration that takes 2.76 times longer to complete. There were only 10 sub-optimal configurations that took 1.5 times longer (less than 0.25%). On the other hand, testing out all the configurations, the strategy of FullSearch, takes 100 times longer in 101 cases compared to testing out the configurations suggested by MachineSearch. Assuming an interval of 100 steps that use the optimal configuration,

and a tuning phase that tests out configurations 3 times, a series of experiments conducted on the dataset collected by the author show that FullSearch requires an average of 26.399 seconds for a tuning phase and 100 steps with the optimal configuration, whereas in average MachineSearch requires only 5.656 seconds. This is due to the average tuning phase of FullSearch, which is 22.791 seconds (with three tests per configuration) compared to 2.048 seconds taken by MachineSearch. Hence, the costs of calling the model is negligible (few milliseconds) to the average benefits gained by MachineSearch, which is over 11 times quicker during the tuning phase. Likewise, if tuning phases happen seldom, e.g. every million steps instead of one hundred, the average time for FullSearch is again only slightly longer, with 36096.149 seconds compared to MachineSearch's 36084.890. These results are also summarized below as a table.

Strategy	Total Time in seconds	Average Time in seconds	Average Tuning Time in seconds
FullSearch	1076305.864	26.399	22.791
MachineSearch	230610.791	5.656	2.048

Table 9.5.: The time of various statistics for both strategies, when relevant configurations are tested three times (default value) and the chosen solution is used for 100 time steps, given in seconds

10. Summary

In this thesis, the author attempted to use machine learning techniques to reduce the search space for simulations during tuning phases. To achieve this end, parameter discovery was done, different layouts for the neural network were tested and the solution was integrated into the C++ Library AutoPas with the help of its developers. It was shown how different parameters influence the timings of configurations and the results were interpreted for the reader. Main methods for machine learning (classification and regression) with different choices for parameters (basic, distribution-based, combined) were compared and the results showed that classification was superior compared to regression and basic parameters outperformed the distribution-based picture parameters. Combined parameters were only slightly better than basic ones. With high precision rates and highly reduced times for the tuning phases (with MachineSearch being in average 11 times faster), the theoretical and practical benefits of using this strategy instead of a full discovery of the whole configuration domain were introduced and advocated.

11. Outlook

There are many ways to improve the whole process. First of all, one can use decision trees instead of classification or regression methods to see if they are more accurate. Second, there are other types of activation nodes that might fare better with the simulation data. Thirdly, more basic parameters can be utilized such as the verlet rebuild frequency, and more parameters can be extracted from the picture, that can be more useful than what convolution could extract. Additionally, it is possible to use better normalization methods to improve the efficiency of the models. Finally, a possible way to further train the model with real world data and benefit end users is to set up a remote system which can be quarried with parameter data. This system can answer the end user as to what configurations are likely to be optimal, and then receive the results of the user after the simulations are completed. Hence, such an online learning system can be utilized to train and use the model.

Part V.

Appendix

A. Problems with Calculations During a Time Step

The complication arises from the fact that a particle's value may be changed by an actor which does not operate the particle. This will be explained more thoroughly in this section.

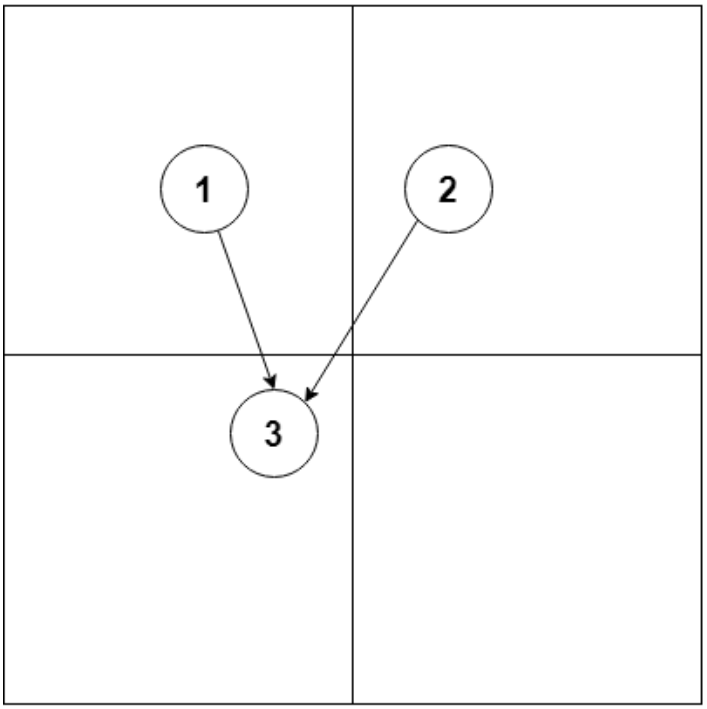


Figure A.1.: Possible problems with parallelization

A simulation is the process and result of interactions and changes that happen over time. To model time, it is divided into tiny time steps. In each of these time steps, forces are calculated, new velocities are found and afterwards particles are moved in a direction. This can be followed by balancing measures between force calculation and movement that normalize or stabilize the speed of particles. After each of these phases are complete, the time is increased, and new calculations based on the new locations and velocities of particles

are possible.
A good way to increase the performance of simulations is to use more computation units that can individually execute a part of the simulation; however, doing so adds the overhead of communication and synchronization. First, as a communication problem, consider that each computation unit calculates a certain segment of the simulation. Although dividing up the particles by identification or spatial relation —such as having two computation units that divide up the simulation domain in two from the middle of the x-axis— is a simple operation, calculating the short-ranged pair interactions between particles become messier. For instance, having enabled Newton 3, a problem such as the one in the figure below might occur: two processing units reading the force value of an interaction partner simultaneously and overwriting each other's results.
Another synchronization problem is due to load balancing. Take a domain where particles are distributed as shown below. The first computation unit will have to wait for the second one because the number of particles they have to handle is not close in numbers. Different traversals approach these problems differently but using more dynamic sharing algorithms have other overhead costs which sometimes do not justify their gains.

B. Bash pipeline: From Cluster Simulations to CSV Files

Multiple Bash scripts have been written to send experiments to the linux cluster, parse data, clean up errors, resize pictures and allow new experiments to be added. In the following sections, the latest version of the scripts are introduced in the order they have been written as needed.

B.1. Parser

The parser is a combination of unix commands but mainly sed. It's aim is to use a log file of a series of experiments to create a csv file that can be fed into the python notebook.

B.2. Job Creator

Job creator is a set of three files that can create as many separate jobs as required for the LRZ cluster.

B.3. Resolution Reducer

Resolution reducer is an awk script that reduces the number of pixels from source dimension to the target dimension. Three different versions of this program have been written with the following average durations per experiment.

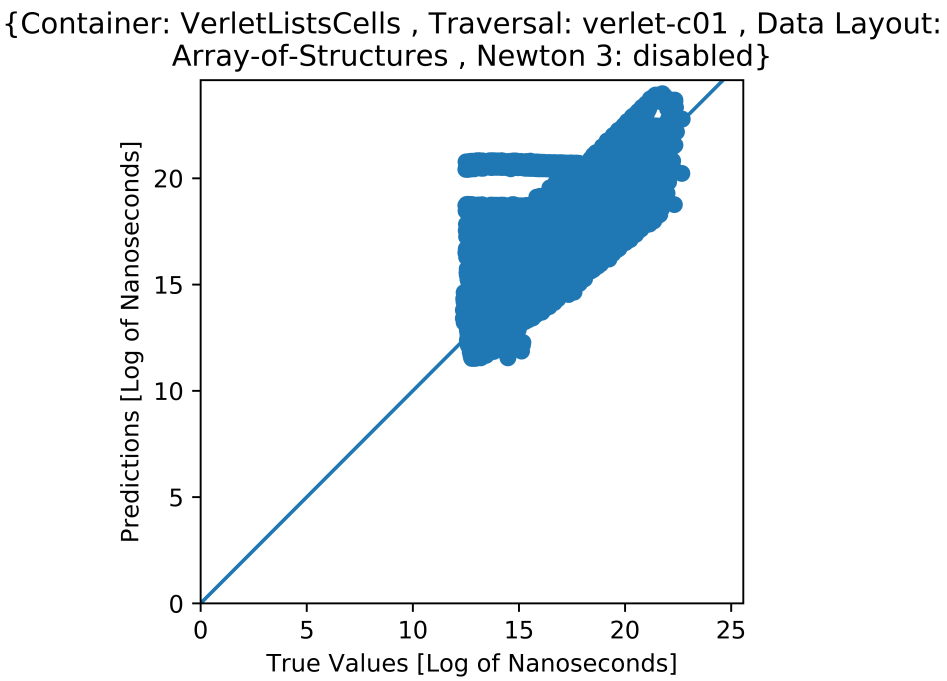
B.4. Error Cleaner

Some experiments cancel early due to time constraints or badly acting configurations that may not complete with some parameters. In these cases, the experiment should be ignored even though it is a part of the parser. The error cleaner ensures that each experiment which has the correct amount of entries are accepted and the rest are removed. This is also done with an awk script.

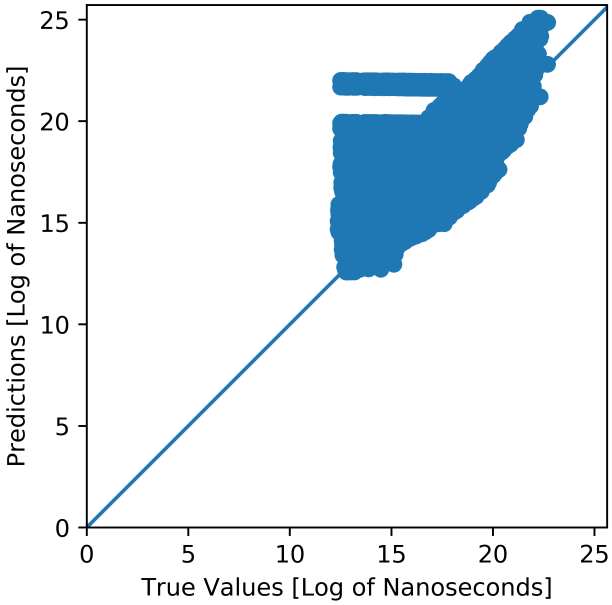
B.5. Experiment Joiner

Adding new configurations means repeating the old experiments. Even though it is possible to repeat the experiments with the same parameters (up to picture parameters), it is still not possible to manually combine experiments. For this purpose, an awk script which joins two results files with each other, using parameters as keys, has been written.

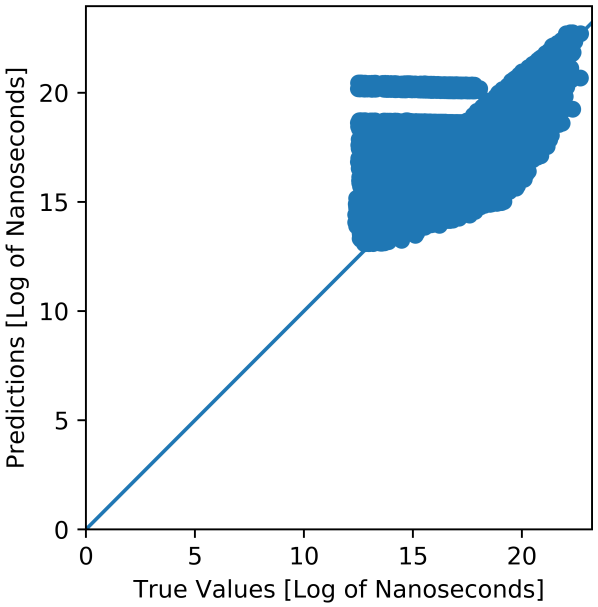
C. All regression graphs



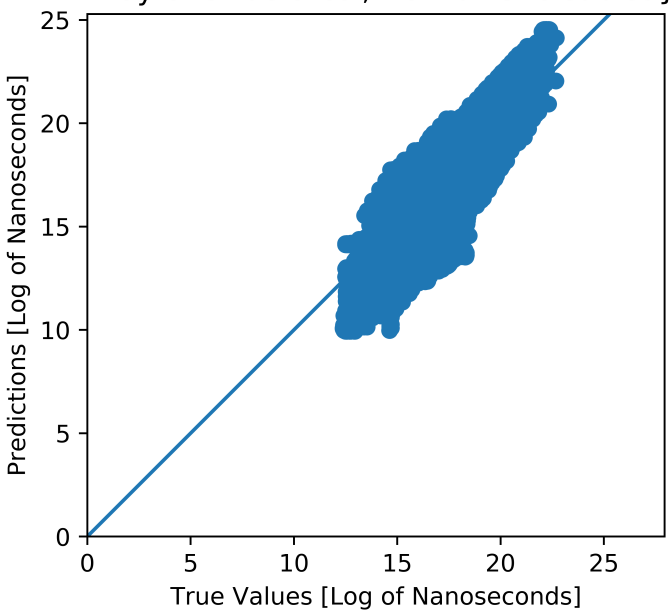
{Container: VerletListsCells , Traversal: verlet-c18 , Data Layout: Array-of-Structures , Newton 3: disabled}



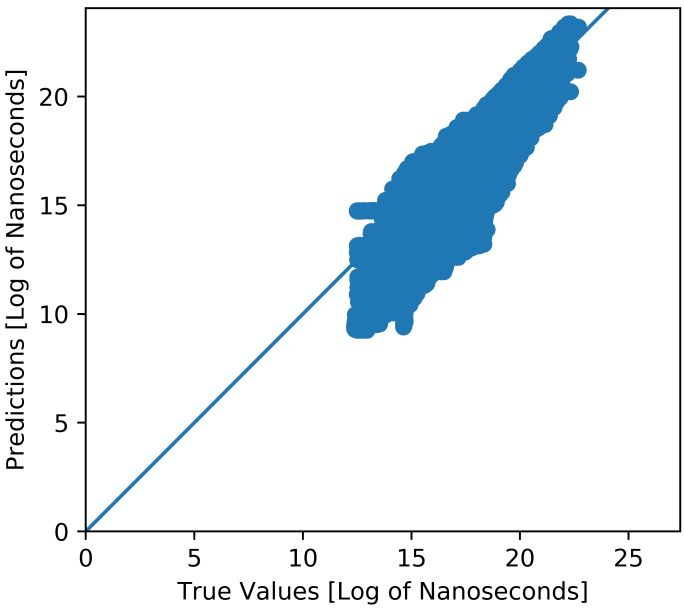
{Container: VerletListsCells , Traversal: verlet-c18 , Data Layout: Array-of-Structures , Newton 3: enabled}



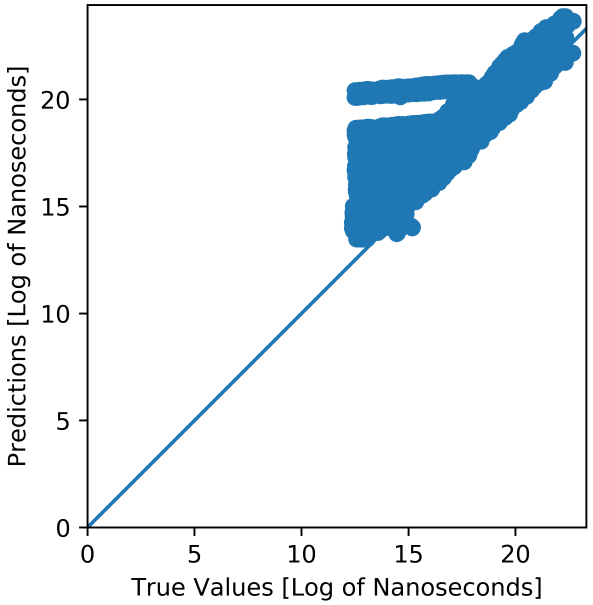
{Container: VerletListsCells , Traversal: verlet-sliced , Data Layout: Array-of-Structures , Newton 3: disabled}



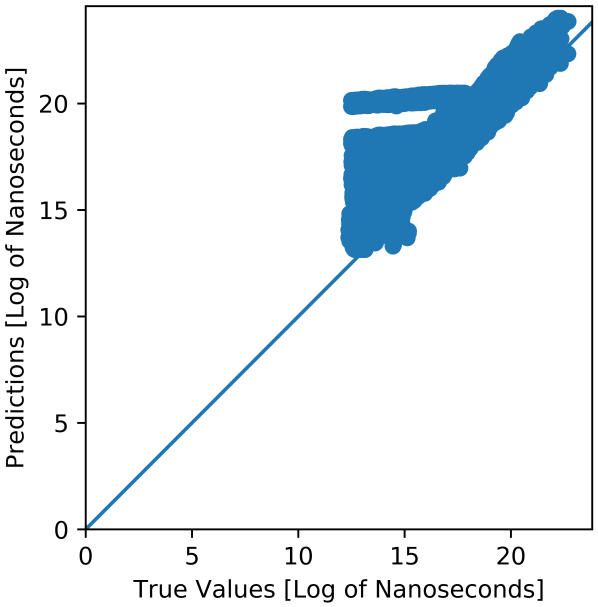
{Container: VerletListsCells , Traversal: verlet-sliced , Data Layout: Array-of-Structures , Newton 3: enabled}



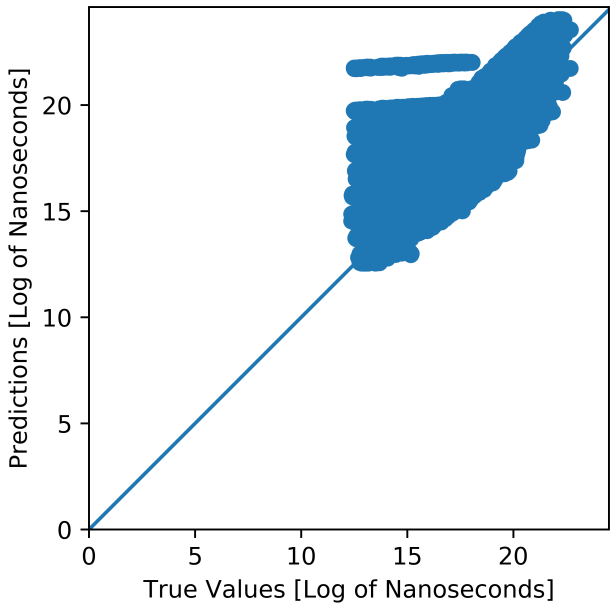
{Container: VerletLists , Traversal: verlet-lists , Data Layout: Structure-of-Arrays , Newton 3: disabled}



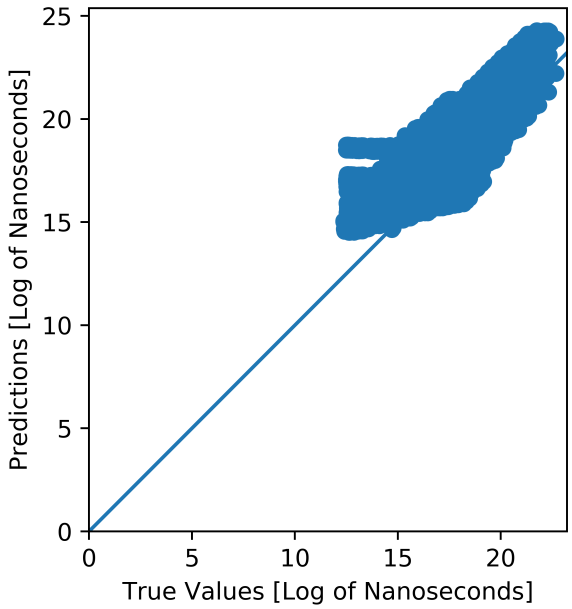
{Container: VerletLists , Traversal: verlet-lists , Data Layout: Structure-of-Arrays , Newton 3: enabled}



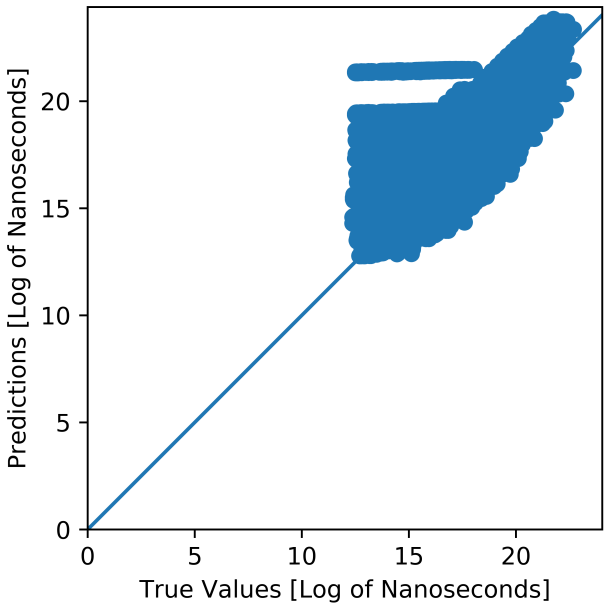
{Container: LinkedCells , Traversal: c08 , Data Layout: Structure-of-Arrays , Newton 3: disabled}



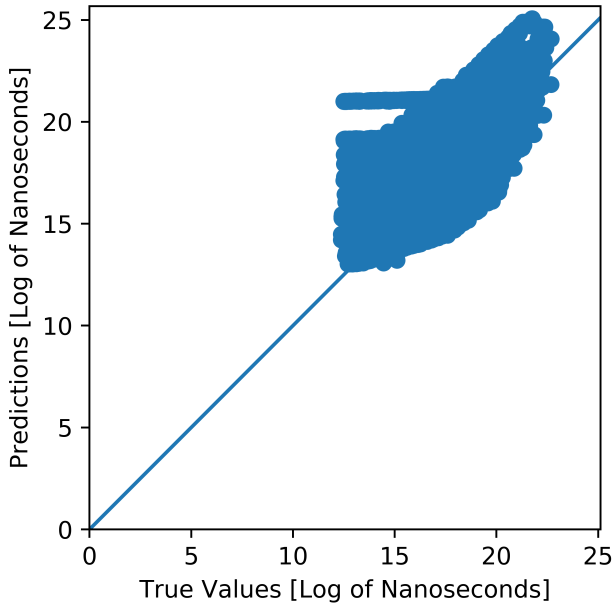
{Container: LinkedCells , Traversal: sliced , Data Layout: Array-of-Structures , Newton 3: disabled}



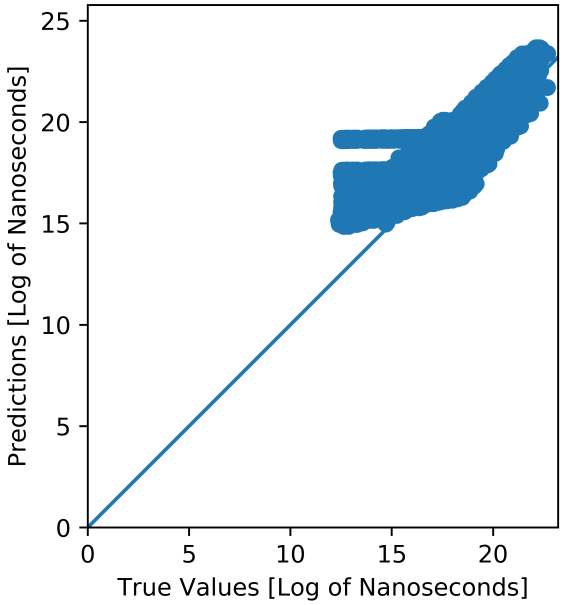
{Container: LinkedCells , Traversal: c08 , Data Layout: Structure-of-Arrays , Newton 3: enabled}



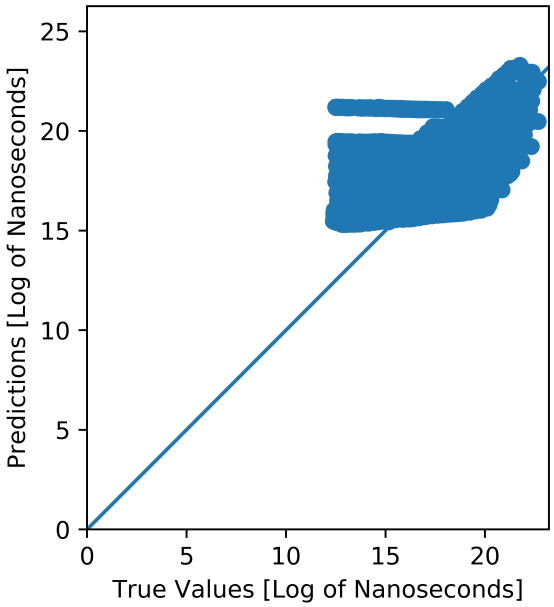
{Container: LinkedCells , Traversal: c08 , Data Layout: Array-of-Structures , Newton 3: disabled}



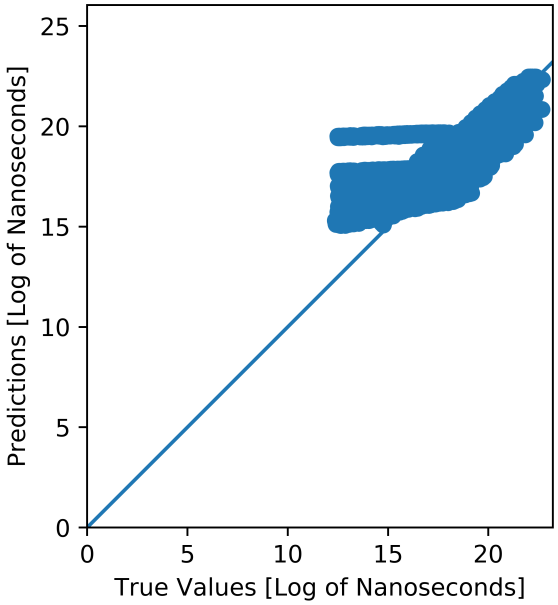
{Container: LinkedCells , Traversal: sliced , Data Layout: Structure-of-Arrays , Newton 3: enabled}



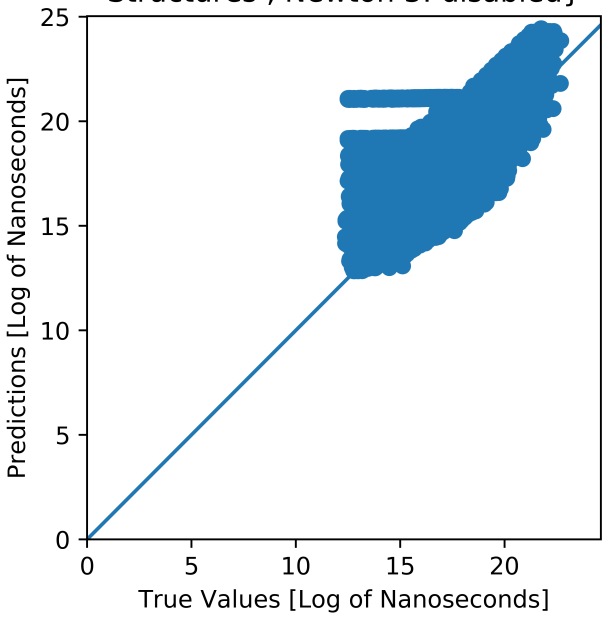
{Container: LinkedCells , Traversal: c08 , Data Layout: Array-of-Structures , Newton 3: enabled}



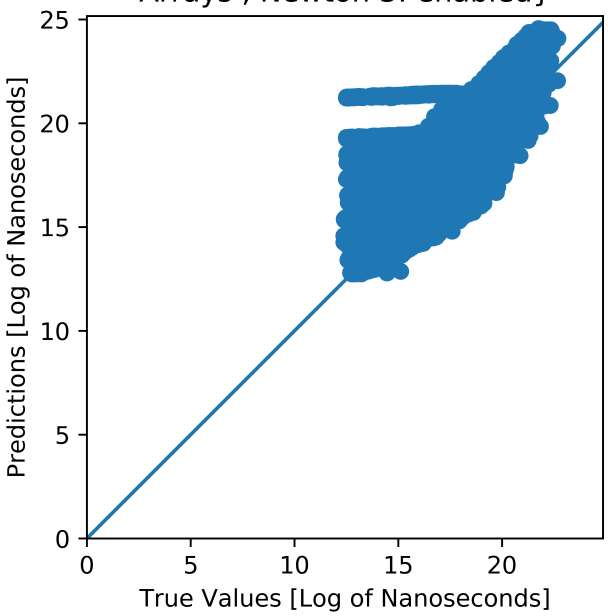
{Container: LinkedCells , Traversal: sliced , Data Layout: Structure-of-Arrays , Newton 3: disabled}



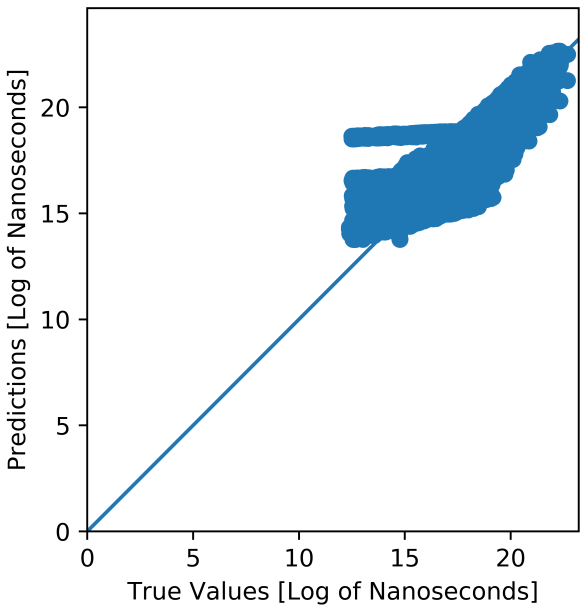
{Container: LinkedCells , Traversal: c18 , Data Layout: Array-of-Structures , Newton 3: disabled}



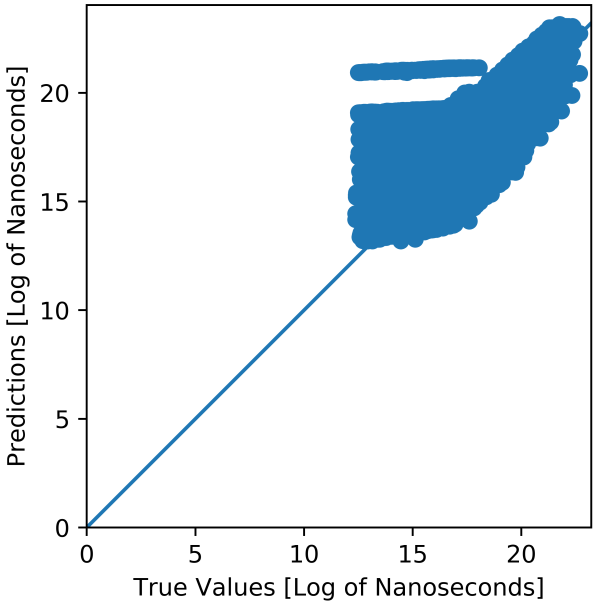
{Container: LinkedCells , Traversal: c18 , Data Layout: Structure-of-Arrays , Newton 3: enabled}



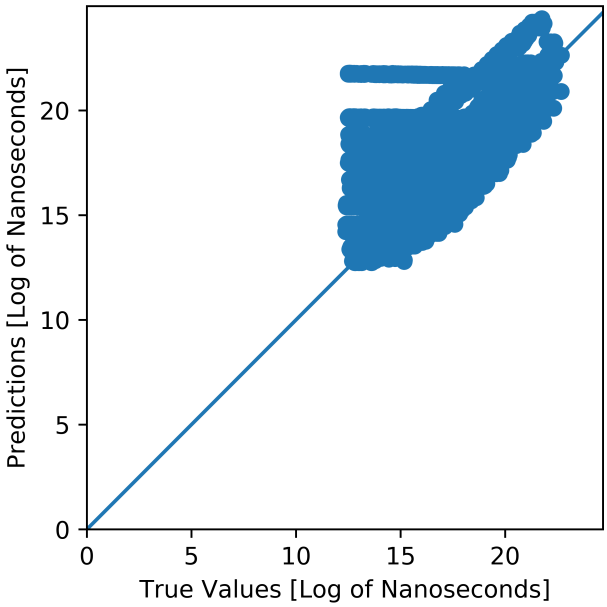
{Container: LinkedCells , Traversal: sliced , Data Layout: Array-of-Structures , Newton 3: enabled}



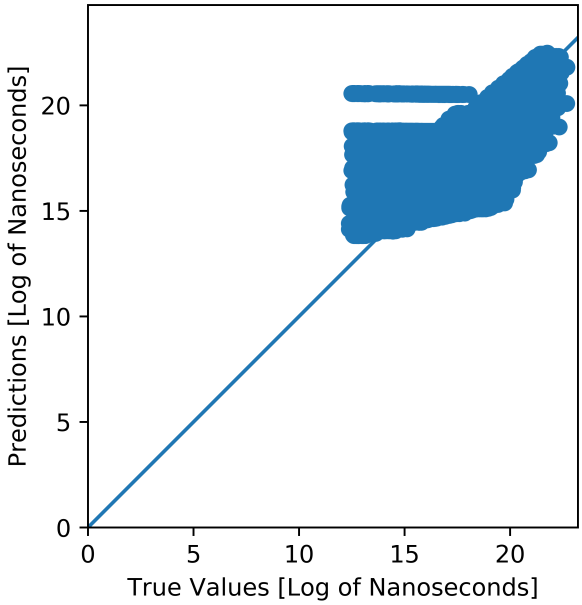
{Container: LinkedCells , Traversal: c18 , Data Layout: Structure-of-Arrays , Newton 3: disabled}



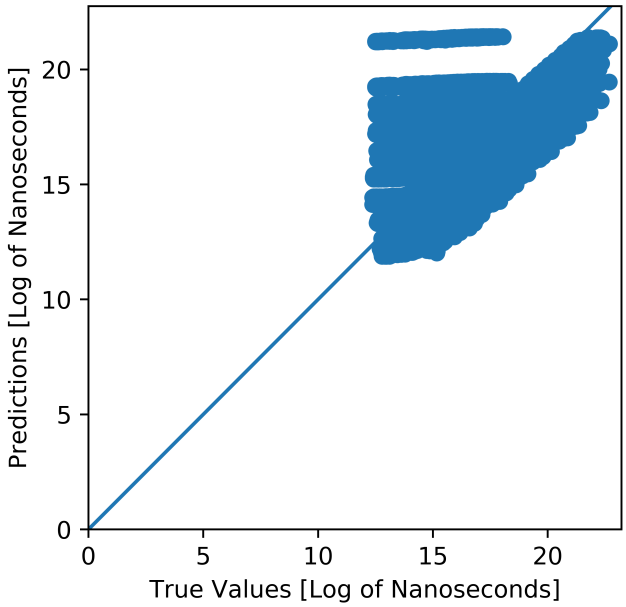
{Container: LinkedCells , Traversal: c01 , Data Layout: Array-of-Structures , Newton 3: disabled}



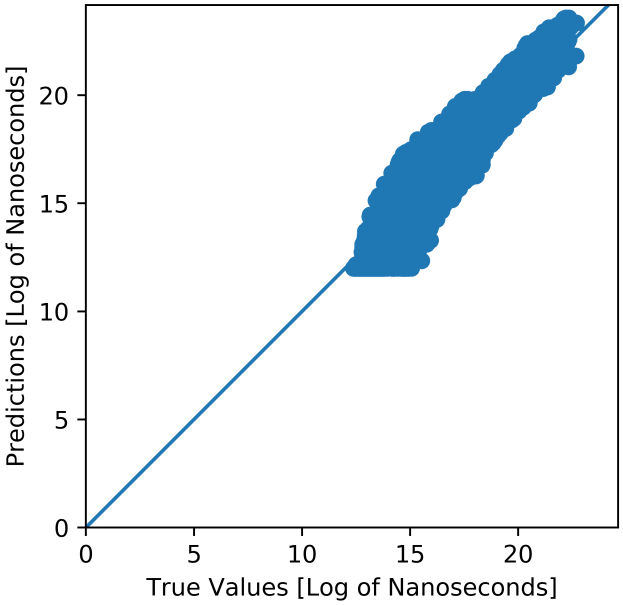
{Container: LinkedCells , Traversal: c18 , Data Layout: Array-of-Structures , Newton 3: enabled}



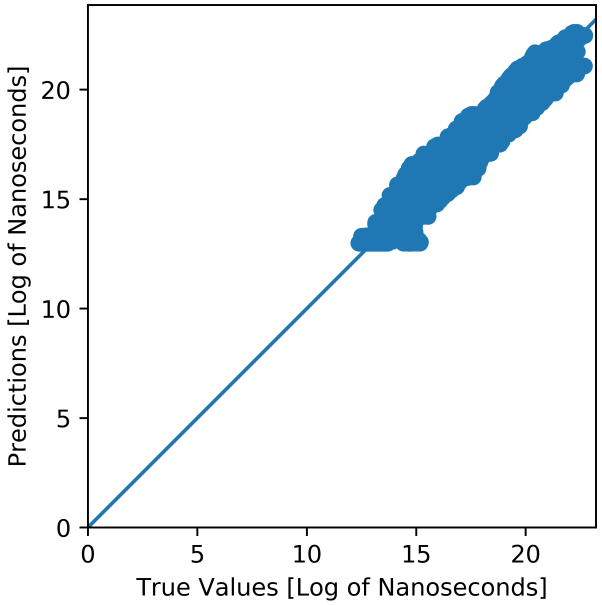
{Container: LinkedCells , Traversal: c01 , Data Layout: Structure-of-Arrays , Newton 3: disabled}



{Container: VerletLists , Traversal: verlet-lists , Data Layout: Array-of-Structures , Newton 3: enabled}



{Container: VerletLists , Traversal: verlet-lists , Data Layout: Array-of-Structures , Newton 3: disabled}



Bibliography

[Adm19] Admin. *How to Manipulate Data Structure to Optimize Memory Use on 32-Bit Intel® Architecture*. Feb. 2019. URL: https://software.intel.com/en-us/articles/how-to-manipulate-data-structure-to-optimize-memory-use-on-32-bit-intel-architecture.

[AW59] B. J. Alder and T. E. Wainwright. "Studies in Molecular Dynamics. I. General Method". In: *Journal of Chemical Physics* 31 (Aug. 1959), pp. 459–466. DOI: 10.1063/1.1730376.

[Bar+05] Marian Stewart Bartlett et al. "Recognizing facial expression: machine learning and application to spontaneous behavior". In: *2005 IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR'05)*. Vol. 2. IEEE. 2005, pp. 568–573.

[Dob19] Dobiasd. *Dobiasd/frugally-deep*. July 2019. URL: https://github.com/Dobiasd/frugally-deep.

[Dzy+92] Igor Ekhiel'evich Dzyaloshinskii et al. "The general theory of van der Waals forces". In: *Perspectives in Theoretical Physics*. Elsevier, 1992, pp. 443–492.

[Fur+11] Grigori Fursin et al. "Milepost gcc: Machine learning enabled self-tuning compiler". In: *International journal of parallel programming* 39.3 (2011), pp. 296–327.

[Gra+19] Fabio Gratl et al. "AutoPas: Auto-Tuning for Particle Simulations". In: *2019 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. 2019.

[HHS] David Harris, Harris, and Sarah. *Digital design and computer architecture*. it. 2nd ed. San Francisco, Calif: Morgan Kaufmann, p. 129. ISBN: 978-0-12-394424-5.

[htta] doug (https://stats.stackexchange.com/users/438/doug). *How to choose the number of hidden layers and nodes in a feedforward neural network?* Cross Validated. URL:https://stats.stackexchange.com/q/1097 (version: 2018-07-22). eprint: https://stats.stackexchange.com/q/1097. URL: https://stats.stackexchange.com/q/1097.

[httb] DeltaIV (https://stats.stackexchange.com/users/58675/deltaiv). *What is the essential difference between a neural network and nonlinear regression?* Cross Validated. URL:https://stats.stackexchange.com/q/345065 (version: 2018-05-08). eprint: https://stats.stackexchange.com/q/345065. URL: https://stats.stackexchange.com/q/345065.

[Ker] Keras. *Why use Keras?* URL: https://keras.io/why-use-keras/.

[Kla+17] Günter Klambauer et al. "Self-normalizing neural networks". In: *Advances in neural information processing systems*. 2017, pp. 971–980.

[LBH15] Y. Lecun, Y. Bengio, and G. Hinton. "Deep learning". In: *Nature* 521 (May 2015), pp. 436–444. DOI: 10.1038/nature14539.

[LHT11] Wei-Yang Lin, Ya-Han Hu, and Chih-Fong Tsai. "Machine learning in financial crisis prediction: a survey". In: *IEEE Transactions on Systems, Man, and Cybernetics, Part C (Applications and Reviews)* 42.4 (2011), pp. 421–436.

[M69] Master MMaster M. *Sparse_ categorical_ crossentropyvscategorical_ crossentropy(keras, accuracy)*. Apr. 1969. URL: https://datascience.stackexchange.com/questions/41921/sparse-categorical-crossentropy-vs-categorical-crossentropy-keras-accuracy.

[Mon+08] Concepcion A Monje et al. "Tuning and auto-tuning of fractional order controllers for industry applications". In: *Control engineering practice* 16.7 (2008), pp. 798–812.

[New] I. Newton. *The Mathematical Principles of Natural Philosophy*. URL: https://books.google.com.tr/books?id=Tm0FAAAAQAAJ&pg=PA20&redir_esc=y#v=onepage&q&f=false.

[NH10] Vinod Nair and Geoffrey E Hinton. "Rectified linear units improve restricted boltzmann machines". In: *Proceedings of the 27th international conference on machine learning (ICML-10)*. 2010, pp. 807–814.

[Ope18] OpenMp. *About Us*. July 2018. URL: https://www.openmp.org/about/about-us/.

[Sar94] Warren S Sarle. "Neural networks and statistical models". In: (1994).

[Sil+16] David Silver et al. "Mastering the game of Go with deep neural networks and tree search". In: *nature* 529.7587 (2016), p. 484.

[Tena] TensorFlow. *Train your first neural network: basic classification*. URL: https://www.tensorflow.org/tutorials/keras/basic_classification.

[Tenb] TensorFlow. *Why TensorFlow : TensorFlow*. URL: https://www.tensorflow.org/about.

[TH08] M. Trenti and P. Hut. "N-body simulations (gravitational)". In: *Scholarpedia* 3.5 (2008). revision #91544, p. 3930. DOI: 10.4249/scholarpedia.3930.

[Ver67] Loup Verlet. "Computer "Experiments" on Classical Fluids. I. Thermodynamical Properties of Lennard-Jones Molecules". In: *Phys. Rev.* 159 (1 July 1967), pp. 98–103. DOI: 10.1103/PhysRev.159.98. URL: https://link.aps.org/doi/10.1103/PhysRev.159.98.

[Zul18] Hafidz Zulkifli. *Understanding Learning Rates and How It Improves Performance in Deep Learning*. Jan. 2018. URL: https://towardsdatascience.com/understanding-learning-rates-and-how-it-improves-performance-in-deep-learning-d0d4059c1c10.