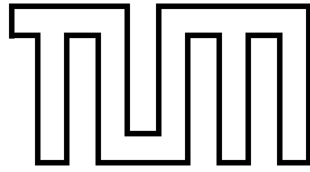


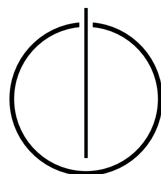
FAKULTÄT FÜR INFORMATIK

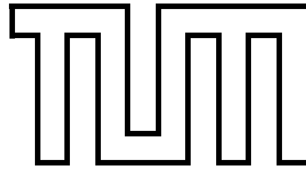
DER TECHNISCHEN UNIVERSITÄT MÜNCHEN

Bachelor's Thesis in Informatics

Can Reinforcement Learning be used to improve the autotuning process within AutoPas?

Leonhard Laumeyer





FAKULTÄT FÜR INFORMATIK

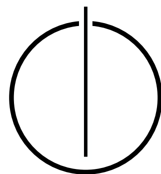
DER TECHNISCHEN UNIVERSITÄT MÜNCHEN

Bachelor's Thesis in Informatics

Can Reinforcement Learning be used to improve the autotuning process within AutoPas?

Kann Reinforcement Learning zur Verbesserung des Autotuning Prozesses in AutoPas eingesetzt werden?

Author: Leonhard Laumeyer
Supervisor: Univ.-Prof. Dr. Hans-Joachim Bungartz
Advisor: Fabio Alexander Gratl, M.Sc., Samuel Newcome, M.Sc.
Date: 15.09.2022



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

Munich, 15.09.2022 Leonhard Laumeyer

Abstract

This thesis presents a new tuning strategy for the node-level auto-tuned particle simulation library AutoPas. The strategy uses reinforcement learning to predict the best configuration for the simulation to use to achieve the fastest calculation time. An implementation of a modified version of the SARSA algorithm is shown. Furthermore, the hyperparameters: learning rate, discount factor, and exploration rate are fine-tuned trough grid search to produce the best possible results. The reinforcement learning tuning strategy is then tested according to different criteria. These criteria are then used to compare it against the full search and predictive tuning strategies. The reinforcement learning tuning interface shows considerable improvement compared to the already implemented options.

Contents

Abstract **vii**

I. Introduction and Background **1**

1. Introduction **2**

2. Background **4**
2.1. AutoPas 4
2.1.1. Data Layouts 4
2.1.2. Particle Containers 5
2.1.3. Newton3 5
2.1.4. Traversals 6
2.1.5. Tuning Procedure 6
2.1.6. Existing Interfaces 7
2.2. Machine Learning 8
2.2.1. Reinforcement Learning 8
2.2.2. Temporal Difference 9
2.3. Selection of the Reinforcement Learning Algorithm 10
2.3.1. Advantages and Disadvantages 10
2.3.2. Why Sarsa 10

II. Implementation **12**

3. Actual Implementation **13**
3.1. TuningStrategyInterface 13
3.1.1. addEvidence() 13
3.1.2. tune() 14

III. Results and Analysis **15**

4. Hyperparameter Tuning **16**
4.1. Learning Rate 16
4.2. Discount Factor 19
4.3. Initial States 20
4.4. Random Exploration 21

5. Comparison with Full Search and Predictive Tuning **23**
5.1. Exploding liquid vs Falling drop 23
5.2. Performance with OpenMP 26

IV. Future Work and Conclusion **28**

6. Future Work **29**

7. Conclusion **30**

V. Appendix **31**

A. Yaml Configurations **32**

B. Simulation parameters **34**

Bibliography **37**

Part I.

Introduction and Background

1. Introduction

In recent years a majority of applications in computer science have gotten more complex and detailed [JWCW15]. This resulted in the need for more computing power, which also applies in Molecular Dynamics. Contrary to other fields, particle simulations have not necessarily gotten more complex since the underlying physics law didn't suddenly change in the last decade. But the simulations have gotten more extensive and more realistic. Usually, there are two ways to approach this problem of having more and more complex calculations.

First, just a faster or better computer to run these simulations. But this comes with its own problems, like a limit to the speed of modern CPUs or the pricing going up exponentially at higher clock speeds.

The second way is to optimize the software itself. The AutoPas library would be one example of this. AutoPas dynamically changes the configuration of the simulations, such as the data layout or the particle container. The way it works is that after a set number of iterations, a selection of configurations is tested, and the best one is chosen at the end. This configuration is then used until the next tuning phase. There are predefined strategies for the tuning of options, such as Full Search or Predictive Tuning.

As the name suggests, full search tests every possible combination and then selects the one with the best results. This comes with a few problems, such as a considerable overhead in tuning time since it is a brute-force method and might test many bad configurations. The Predictive Tuning strategy, on the other hand, uses old tuning results to predict the current run time by extrapolation through linear regression and then selects and only tests those configurations. The problem with this is that the run time of a simulation is not just a linear function like the number of particles times calculation time. It is hard and maybe even impossible to represent the actual values by a polynomial function of the first degree with acceptable accuracy.

Other fields, like real-time image recognition, also run into similar problems trying to determine the best of something with the shortest calculation time or knowing the least about the underlying environment. Most of them turned towards machine learning to solve this problem. Machine learning is a big field that unites a variety of algorithms and methods that allow the computer, as the name suggests, to learn. The idea is that instead of giving the computer a set rule or procedure to follow for the decision-making, it understands it by itself. Machine learning splits into three categories: supervised, unsupervised, and reinforcement learning.

Supervised and unsupervised learning try to make sense or find a pattern in existing data. On the other hand, reinforcement learning tries to learn a policy by trial and error. Reinforcement learning can be seen as putting a toddler in a situation and letting him try everything out, supervised and unsupervised learning would be a person discovering the problem through books.

Reinforcement learning improves over the brute force method by not needing to run everything every time again since it can use old data to predict new data. On the other

hand, it also handles complex data better than linear regression since it is not limited by the linearity of the data.[KOK05] Which would make it an excellent candidate for an AutoPas tuning strategy. In this paper, we are exploring the potential benefits of using Reinforcement Learning as the selector strategy in the AutoPas library.

2. Background

2.1. AutoPas

AutoPas is an open-source node-level performance library that can be used for iterative tuning different configurations for short ranged pairwise force calculations. The options include Newton's third law of motion, different data layouts, particle containers, and traversals. Another option that can be set at the beginning of the simulation is which tuning strategy should be used. AutoPas also includes MD-flexible, an example implementation of a molecular simulations software that utilizes AutoPas to tune the configurations. All simulations are run with the help of this demo software.

2.1.1. Data Layouts

Data Layouts refer to how particle objects are stored during the simulation. AutoPas supports two different Data Layouts: SoA and AoS. They refer to Structure of Arrays and Array of Structures, respectively. Array of Structures saves the complete object particles sequentially, which allows for the fast loading of all the values of a single particle as seen in 2.1. On the other hand structure of Arrays saves the same attributes of all particles in the same array, which makes loading the values into the same vector registers faster.[GSBN22] This can be used by modern CPUs since they can apply the same instruction to multiple values, also called single instruction multiple data (SIMD). This feature allows swift vector operations such as scalar multiplication, which is used very often in the main calculation of a molecular dynamics simulation.[GSBN22]

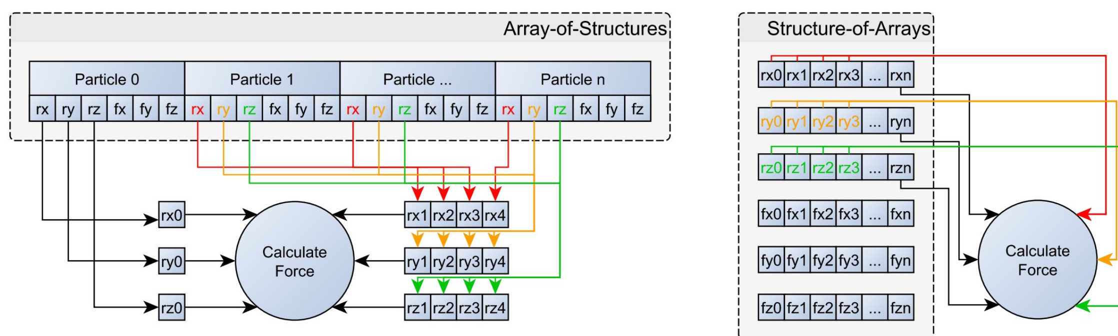
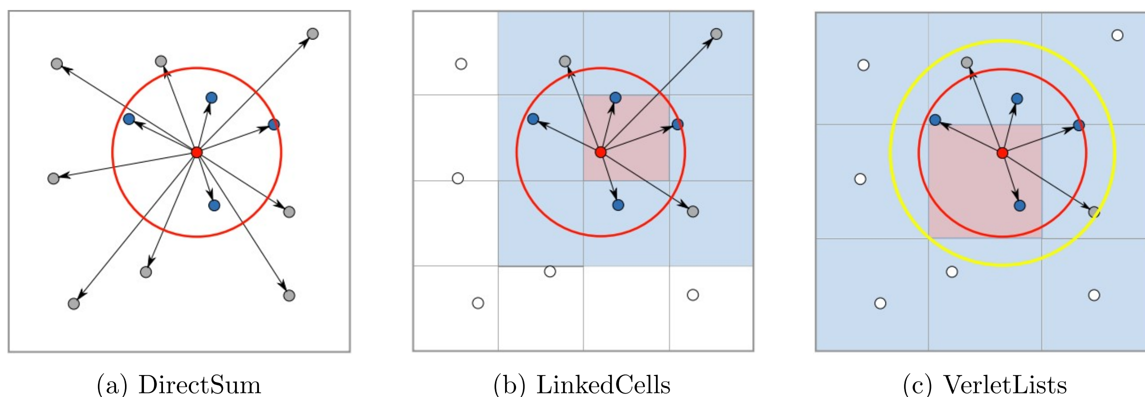


Figure 2.1.: The difference between AoS and SoA when loading values for force calculation. [GSBN22]

2.1.2. Particle Containers

Particle Containers define how the particles are mapped in 2D/3D space, which is important in order to find out for which particle the force calculation needs to be performed. The main forces that are being considered in such a simulation are van der Waals forces and Pauli repulsion. These two forces are combined in the simplified but still realistic Lennard-Jones 12-6 potential. Since the potential mentioned above quickly converges towards zero, it is helpful to introduce a cut-off radius that helps reduce complexity. So every particle outside this cut-off radius is not considered for the force calculation.[GSBN22] To take full advantage of this, three Particle Containers are implemented in AutoPas Direct Sum, Linked Cells, and Verlet Lists. Direct Sum is the brute force attempt, every particle is checked against all the other particles, and the force is calculated if the compared particles have a distance less than the cut-off radius. The linked cells algorithm is the first one that uses the cut-off radius to improve upon the direct sum version. This is done by introducing sectors, which should have at least the length of the cut-off radius. This allows for the reduction of the number of particles that need to be checked for distance since only the particles in the neighboring cells need to be considered now instead of all. The verlet lists further reduces the number of particles that need to be considered for force calculation by calculating a list of particles in proximity. This reduces the number of unnecessary distance calculations even further but introduces an overhead. The overhead is created by the need to rearrange the particles after each iteration and create a new neighbor list.[GSBN22]



(a) DirectSum (b) LinkedCells (c) VerletLists

Figure 2.2.: The red point is the selected main particle for the pairwise force calculation. The red circle is the cut-off radius for the Lennard Jones potential. The distance is calculated for every particle pair with an arrow, but the force calculation is only needed for the blue ones. The yellow circle is the Verlet-skin radius. The blue and red cells are being used in the neighbor list.[GSBN22]

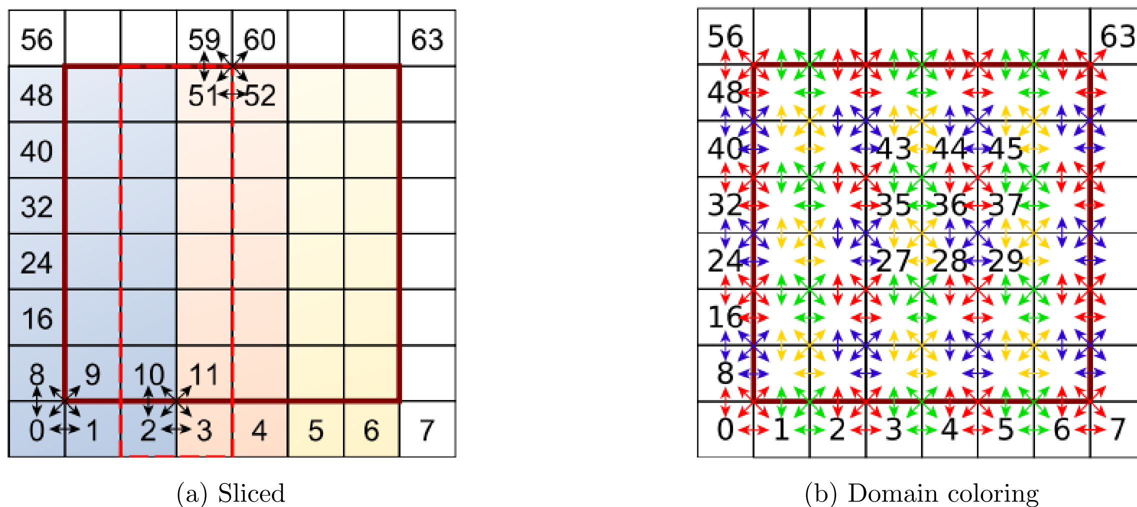
2.1.3. Newton3

Newton3 refers to the application of Newton's third law of motion or also commonly known as "action equals reaction". This law declares that every force has an opposing force that goes in the opposite direction. In molecular dynamics, that means that every force generated by particle A towards particle B also has an anti force from particle B towards particle A

with the opposing direction. The force is defined as $F_{AB} = -F_{BA}$[NC23] This law allows for an extensive optimization in the force calculations since the force between two particles now only needs to be calculated once and can be applied with the negation of the algebraic sign for the affected particle. This reduces the number of force calculations by half and only adds the negating of the force. The other side of the coin is that this performance optimization allows for race conditions when multiple threads work on the force calculation. This can be reduced or avoided by selecting an appropriate traversal strategy.[GSBN22]

2.1.4. Traversals

AutoPas also supports the use of OpenMP, which allows for shared-memory parallel programming. In order to use this efficiently, predefined traversal patterns are required to avoid deadlocks or bottlenecks in memory access. The idea is to split the work into parts that can run synchronized without the need to consider race conditions. The basic version is to lock all the neighboring cells in the linked cells algorithm. In a 3D space, instead of locking all 26 neighbors, it is now only necessary to lock 7 neighboring in a 2x2x2 cell. The second traversal option is slicing the domain in equally sized portions indicated by the different colors in Figure 2.3a. This has the advantage of very little overhead. To avoid race conditions on the border of two slices, the last row of the first domain is locked until the first of the second one is finished. Another option is domain coloring, which splits up the 8 cells in 3D and 4 cells in 2D intelligently so they can be worked on simultaneously. The Figure 2.3b shows the different domains that can be handled at the same time indicated by the different colors.[GSBN22]



(a) Sliced (b) Domain coloring

Figure 2.3.: Traversal patterns that prevent race conditions. The brown line indicates the end of the space and the outside cells are used for halo particles.[GSBN22]

2.1.5. Tuning Procedure

The tuning process of AutoPas works in such a way that the simulation is halted after a preset amount of iterations and starts a tuning phase. The tuning phase begins with the

tuning interface selecting the first configuration to test. This configuration is now used to run the simulation for one iteration while the time is recorded. After the iteration, the time measured is stored for this specific configuration. Then the next configuration is selected and measured. This goes on until every selected combination is tested. After finishing the collection of timings, the tuning interface sets an optimal configuration based on the new results. This best configuration is now used until the next tuning phase, where everything beings from the start again.[GSBN22]

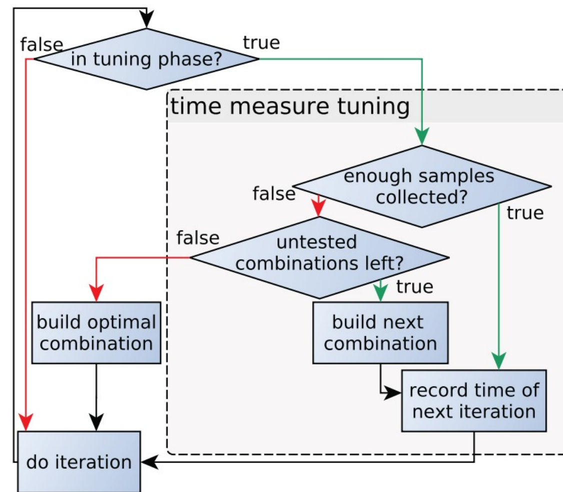


Figure 2.4.: Model of the tuning process of the AutoPas library[GSBN22]

2.1.6. Existing Interfaces

AutoPas uses polymorphism to provide the implementation of different tuning algorithms to the user, which all inherit from the TuningStrategy Interface. The Interface provides the basic functionality of adding evidence, getting the next configuration, and getting the best one at the end of the tuning phase. Before the start of the simulation, there is the option to select one out of these tuning strategies:

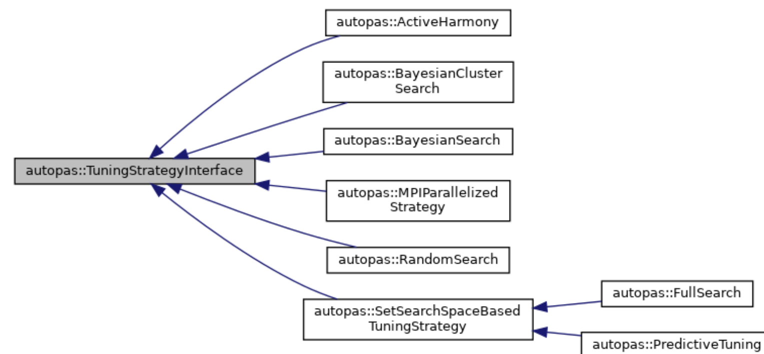


Figure 2.5.: The already existing tuning interfaces within AutoPas.

All the other tuning strategies 2.5 besides PredictiveTuning and FullSearch will be excluded

since they are not used in this paper.

Full Search

Full search is akin to brute force since it tests every possible configuration one after another and then returns the one with the shortest run time. It only takes the current iterations into account and disregards any previous evidence. It has the problem of having a considerable overhead in the form of spending a long time tuning, depending on the number of possible configurations. Out of which, especially the really bad performing configurations take a substantial amount of time.

Predictive Tuning

Contrary to full search, predictive tuning uses only a fraction of all the possible configurations to reduce run time by ignoring inefficient configurations. It uses one of the three methods, linear regression, newton polynomial, or Lagrange polynomial, to predict the upcoming run times based on old evidence. It also allows for the option of a blacklist which is based on the assumption that inefficient configuration can not improve enough to beat the best one and therefore disregard this arrangement of options for the rest of the simulation. This allows for the testing of only relevant configurations depending on these predictions. Which can reduce the run time by 74% in the spinodal decomposition simulation by using linear regression and blacklisting. Even though it has faster run times than full search, it not always chooses the best configuration to use. [Pel20]

2.2. Machine Learning

The definition of machine learning is "the use and development of computer systems that are able to learn and adapt without following explicit instructions, by using algorithms and statistical models to analyze and draw inferences from patterns in data."[oxf00] So the goal is to imitate the human brain by not giving it a set or fixed list of instructions that need to be finished but to learn the process by itself.

The field can be split into supervised, unsupervised, and reinforcement learning. In supervised learning, the goal is to map the input to outputs, like if the input is "A", the output needs to be "1". This is done by giving a computer a set of example data, and it needs to find the function to get the correct output.[RN10] Unsupervised learning, on the other hand, also uses a set of example data, but this time the data has no labels. This forces the computer to find hidden patterns by itself.[HS99] Reinforcement learning goes even further toward the idea of simulating the human brain.

2.2.1. Reinforcement Learning

Reinforcement Learning is the name for computational learning by experience. What differentiates reinforcement learning from other areas of machine learning is that it learns from experience rather than labels or predefined instructions. It achieves that by learning in an actual environment, that can be simulated or natural, in which an agent takes action and is rewarded or punished based on the action taken. Those actions align with reaching a

specific goal, such as winning a game or finishing a particular task. Reinforcement learning algorithms consist of a policy, a value function, a reward, and sometimes an environment.

Policy The policy contains the allowed or possible actions an algorithm or agent can choose from in each state. It can be the cardinal direction one can move in or the different probabilities of the next dice throw. There are two different classes on-policy and off-policy. On policy improves the policy itself. Off policy creates a completely new one.

Value Function The value function annotates the value of each state, meaning how much a state is worth compared to others. This means a state with a high value will give you a higher reward in the end. In a Maze solving problem, it would stand for the distance to the exit.[SB18]

Reward The reward signifies the beneficiality of an action. It is given by the environment to the agent to signal good or bad behavior regarding reaching the end goal. It is then used to update the value of each state. The reward can be compared to information in the hot and cold game, where hot stands for a good thing and indicates being near the end, and cold means the opposite. So if you get the call hot, it means you are near the goal, and the place you are should have a high reward.[SB18]

Environment The environment is a model based on predictions or experience that simulates the environment or at least tries to guess how the environment would behave. Not all reinforcement learning systems use models since they can also just explore the problem through trial and error.[SB18]

2.2.2. Temporal Difference

The Difference between other Reinforcement Learning algorithms and Temporal Difference is that they do not require the final outcome but can already learn while interacting with the environment. This is achieved by bootstrapping, i.e., predicting based on prediction. The way of solving a problem does not change from other algorithms in that it still uses the acquired experience to update its estimated value for a specific state. There are a few different algorithms for TD learning, most notably Q-learning, Sarsa, and TD(λ); each has multiple modified versions to improve or change the original function.[SB18]

Q-Learning

Q-Learning is an off-policy TD control algorithm, meaning it evaluates all q values without following the policy in each state. It learns a value for all existing state action pairs independent of it being a possible combination and uses the max out of all of them.

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha[R_t + \gamma \max_a Q(S_{t+1}, a) - Q(S_t, A_t)]$$

Sarsa

Sarsa (State-Action-Reward-Sate-Action) is an on-policy temporal difference algorithm that learns the reward for each state and action pair rather than just the state value.[SB18] The algorithm:

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha[R_t + \gamma Q(S_{t+1}, A_{t+1}) - Q(S_t, A_t)] \tag{2.1}$$

It updates the Q value of the state and action of t based on the old value modified by the q-value of the next step multiplied with the discount factor gamma subtracted by the old value. The new q-value is calculated by the value of the next step multiplied by the discount factor gamma. This is then subtracted by the old value and multiplied by the learning rate alpha. This is then subtracted from the old value and results in the newly calculated value for this state-action pair.[SB18] Both algorithms contain the same parts but differ in the way these are used and combined. $Q(S_t, A_t)$ is the value for the state and action pair at the time t. α is the learning rate that regulates at which rate newly acquired information is used to influence already learned values. It ranges from zero to 1, both included where zero means no learning at all and one means relying very heavily on newly discovered information. R_{t+1} is the reward for the action taken in the state s. γ is the factor that decides how much influence the future reward has on the current state. This parameter also starts at zero but can exceed one, which introduces the problem of exploding q values. The value zero implies that the algorithm is short-sighted and only considers current rewards. Higher values make the function look more into future rewards and even prioritize higher long-term rewards.

2.3. Selection of the Reinforcement Learning Algorithm

As already mentioned in 2.2.2, a few different algorithms will be considered, mainly SARSA and Q-Learning and their variants.

2.3.1. Advantages and Disadvantages

Q-Learning is often used if the goal is selecting an optimal path, but it suffers from variance and, therefore, may take longer to converge.[SB18] There are a few modified Q-Learning algorithms that could also be an option, such as Deep Q-learning, which uses a neural network to represent the q values, but it adds a lot of complexity and a need for extensive data to train on, which makes it unusable in AutoPas as there is currently no option to transfer data between different simulations.[MKS+15] Another option would be the QQ-Learning algorithm which trains two on each other dependent Q-Learning models at the same time to help in noisy environments.[Has10] SARSA has the advantage over Q-Learning in that it might not learn the best path, but it converges faster and with fewer errors.

2.3.2. Why Sarsa

All in all, the SARSA algorithm is a better fit for the AutoPas use case since, even though always getting the best configuration is preferred, making a bad choice can and will cost a lot. Furthermore, SARSA also has the advantage of converging faster, which helps with the limited amount of tuning phases in a simulation. The example of the cliff walking situation

2.6 is an excellent example to highlight the difference. Where it shows that SARSA is better and faster in not choosing a bad path.

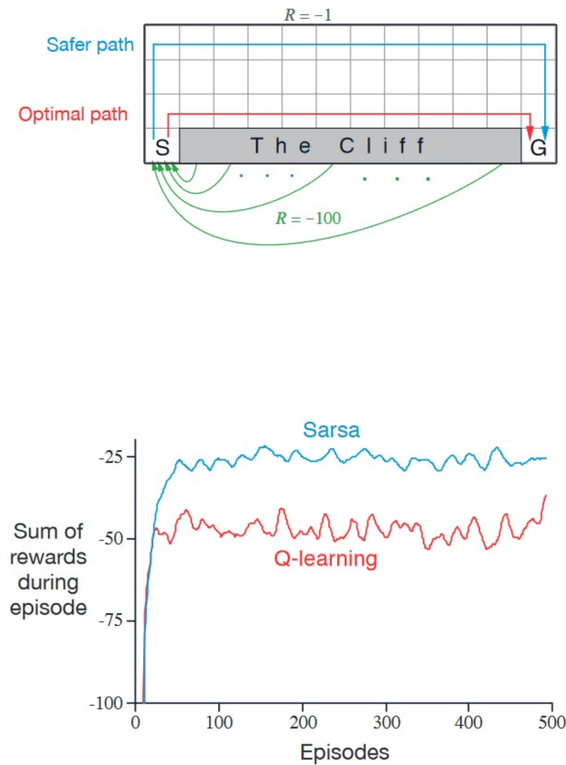


Figure 2.6.: SARSA vs Q-Learnig in high risk high reward situations.[SB18]

Part II.

Implementation

3. Actual Implementation

3.1. TuningStrategyInterface

All tuning strategies need to inherit from the TuningStrategyInterface. The interface provides a set of functions that can or need to be implemented by the actual tuning strategy.

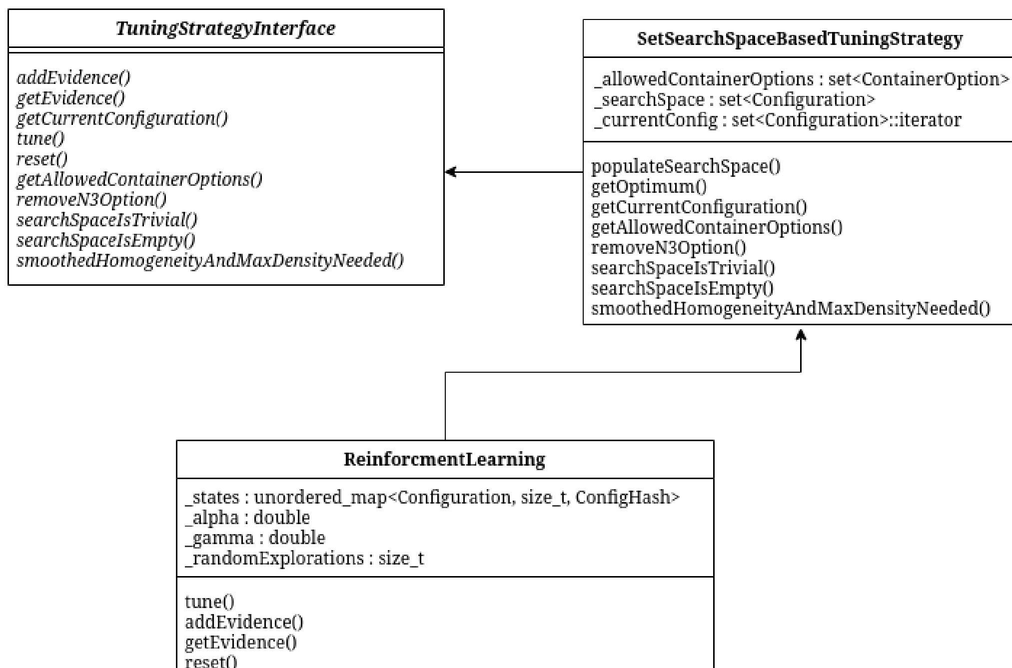


Figure 3.1.: Inheritance of ReinforcementLearning. Helper function, function parameter, and return values are excluded to avoid cluttering.

The reinforcement learning strategy actually inherits from the SetSearchSpaceBased-TuningStrategy, which already implements all functions except addEvidence, getEvidence, tune and reset, as well as adding the two additional functions populateSearchSpace and getOptimum as shown in Figure 3.1. The two functions getEvidence and reset are trivial.

3.1.1. addEvidence()

This function is used to save a certain configuration's run time and update the Q value. On the first tuning phase, the state is initialized with the time measured it took to run on iteration. From the second phase onwards, the value is updated following this algorithm:

```
1    if (_firstTuningPhase) {
2      _newState = -time;
3    } else {
4      _newState = _oldState + _alpha * (-time - _gamma * _oldState);
5
6    }
```

Listing 3.1: Reinforcement Learning part in addEvidence().

The used reinforcement learning algorithm is based on the SARSA algorithm but modified to suit the AutoPas environment.

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha[R_{t+1} + \gamma * Q(S_{t+1}, A_{t+1}) - Q(S_t, A_t)] \tag{3.1}$$

$$_NewState \leftarrow _oldSate + \alpha[-time + \gamma * _OldState]$$

There are two notable changes to the original version 2.1. First, the Reward of the next state is substituted with the negative of the run time. First, there is no next state in the AutoPas version since each tuning phase is its own problem. So instead of the usual m x n Matrix with m,n¿ 1 it only consists of a matrix with n=1. This also removes the q value in the latter half of the bracket. The reward function is just the negative time, as suggested in this paper.[MVC+10] The second change is that instead of the difference between the next and current q value, just the present, not updated value is taken since there is no next state.

3.1.2. tune()

The tune function is responsible for the selection of the configurations to test during this tuning phase, getting the next configuration to try, and selecting the best configuration at the end.

Collection of Configurations

On the first call of the tune function on each tuning phase, the getCollectionOfConfigurations() helper function is called. On the first run, it returns every possible configuration. However, on the second and following calls, it returns just a selection of all possible configurations based on the minimum of the _randomExplorations attribute or possible configurations. The new selection of configurations always contains the best possible from the last tuning phase and a number of random samples out of the rest.

Optimal Configuration

After the last configuration has been tested, the optimal configuration is selected based on the maximal value out of all the state values, i.e., the configuration which should take the least amount of time.

Part III.

Results and Analysis

4. Hyperparameter Tuning

As stated in 2.2.2 all Temporal Difference algorithms have a learning rate α and a discount factor γ. The modified version of the SARSA algorithm also uses both parameters, even though the use and function differ from their original use. In order for them to be actually helpful, they need to be selected carefully or tuned through trial and error.

4.1. Learning Rate

The learning rate is still the same in that it controls how much new results weigh in during the update of the state value. A common technique used for the learning rate in machine learning is to adapt the learning rate during the training process to archive a better accuracy or a faster training time. A changing learning rate can get better end results since it allows for smaller update steps the further the training continues. It can also be faster since it allows for bigger update steps in the beginning while still keeping a good accuracy by having smaller steps towards the end.[Smi15] But since the results of the AutoPas tuning phase change between each iteration, it only serves as a disadvantage since it assumes that the results are constant and don't change over the training. So the best option is to use a constant predefined learning rate.[SB18]

The function needs to go through hyperparameter optimization to find the best value for α. There are different techniques for this process. One of them is grid search, which manually selects a value from a predefined range and then measures its behavior according to a performance metric and repeats that for all numbers in the range. In the case of AutoPas, the range is the whole range of allowed values from zero to one, both included in 0.05-sized steps. The performance metric is the time spent per non-tuning iteration during a simulation, which is calculated by dividing the total time spent non-tuning through the non-tuning iterations.

Figure 4.1.: Learning rate tuning with $\gamma = 0.8$ on the fallingDrop1 simulation A.1.

Figure 4.2.: Learning rate tuning with $\gamma = 0.7$ on the explodingLiquid simulation A.2.

Both Figures 4.1 and 4.2 show a clear indication that the learning rates between 0.8 and 0.9 produce the best results. The best testing timing for fallindDrop1 is at $\alpha = 0.85$ with $0.003468s$ spent per iteration and for explodingLiquid the best value is $\alpha = 0.8$ with a timing per iteration of $0.000462s$. This can be explained by the fact that if you have a deterministic environment, a learning rate of $\alpha = 1$ will give you the best results.

A learning rate of near zero is often seen in a stochastic environment since each result is only a partial truth of the complete answer. Take the example of a coin throw. If the learning rate is set to one during each iteration, the state values will change and favor or even only accept the current result. But since both possibilities are equally likely to occur, this function will never converge to the right solution. But if the learning rate is set to near zero, each iteration will lower the distance toward the correct answer of 50% for tails and heads, as seen in Figure 4.3.

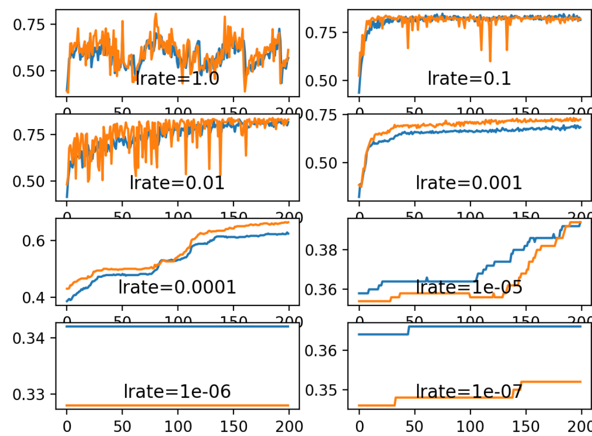


Figure 4.3.: Line Plots of Train and Test Accuracy for a Suite of Learning Rates on the Blobs Classification Problem[Bro]

Returning to AutoPas, which in theory is a deterministic environment since each instruction should always take the same amount of time in optimal conditions. The problem is that many factors influence the final runtime, such as CPU workload, CPU clock boost, memory response time, and many more. AutoPas doesn't have these kinds of information when saving results since, first of all, most of them have very little influence, and they would introduce a big overhead to collect and store. Therefore it can happen that the two same configurations at the same iteration of the same simulation produce different timings, which makes the result appear stochastic to AutoPas. For example, the fur color of a rabbit, as defined in Table 4.1, is always the same no matter how often it is checked.

Animal	Name	Colour of Fur
Rabbit	Anton	Black
Rabbit	Oskar	White

Table 4.1.: Color of the rabbits separated by name

But if you stop including the name in the table - which would be all the additional parameters in AutoPas - the answer changes from being deterministic to stochastic since the color of either Anton or Oskar is being checked and reported as seen in Table 4.2.

Animal	Colour of Fur	Percentage
Rabbit	Black	50%
Rabbit	White	50%

Table 4.2.: Color of a random rabbit

This allows for the explanation of the learning rate value of roughly 0.8, as the environment in itself is deterministic. Still, due to missing factors and the high complexity, the timings appear slightly stochastic.

4.2. Discount Factor

The discount factor γ serves a different purpose in the AutoPas version of the SARSA algorithm since the original use case of limiting the influence of future decisions does not apply in AutoPas. Because the Equation 3.1 does not subtract the current state from the next state but only multiplies the current state with γ, its purpose moves towards limiting the importance of previous timings. Since the discount factor is also a hyperparameter, it needs to be tuned to achieve the best result. So the same technique as for the learning rate is applied with the parameters from zero to one with 0.05-sized steps. Even though higher numbers than one can be used and may provide a better result this time, it may cause the value of the state to explode, meaning it will continue to diverge from zero instead of converging on a specific number. The performance metric is also the time spent per non-tuning iterations during a simulation.

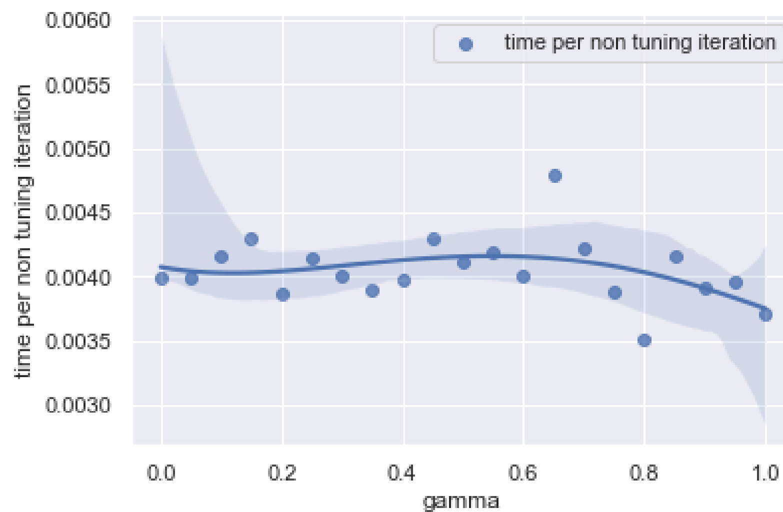


Figure 4.4.: Learning rate tuning with $\alpha = 0.9$ on the fallingDrop1 simulation A.1.

Even though it has a high variance, it still suggests a clear downwards trend for higher γ's. The best timing was measured with $\gamma = 0.8$ with 0.003504[1]. The linear regression plot

[1]The timings are averaged over different α values, which introduces a difference in time measured between the best alpha and gamma timings.

would suggest that values higher than one would produce a better result, but it is not being considered because it can cause an inter overflow. The value of 0.8 can be explained by the fact that there are clearly better configurations that do not increase efficiency during a simulation.[Pel20] But there are still changes in the overall distribution of the particles in the room during a simulation which can make slight variations of configurations better while the simulation progresses.

4.3. Initial States

The choice of the initial condition can impact the time of learning as well as the initial bias towards specific actions. The initial bias never disappears for constant α's but only gets smaller. But having an initial bias is not necessarily bad since it acts as an additional parameter that can reflect the prior knowledge about an environment. So there are two ways to set the initial value for a q state.

First, set the value to something very optimistic, like a high number; this forces the algorithm towards a high exploration since, after the first testing, the value is going to be lower and needs to be adjusted to be a worse option. This encourages the algorithm to try all actions multiple times since the states need several updates to reach their optimal value.[SB18]

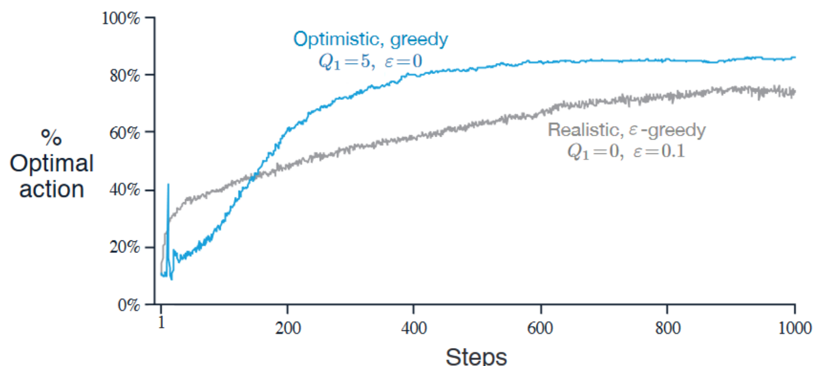


Figure 4.5.: The effect of optimistic initial action-value estimates on the 10-armed testbed. Both methods used a constant step-size parameter, $\alpha = 0.1$. [SB18]

The second option is to use the reward of the first test, which purposely introduces a bias towards certain actions. This allows, first of all, the introduction of prior knowledge about the environment if it is deterministic. In addition, it also speeds up learning and convergence since there is no need to update the value of the states multiple times to even get close to the optimal reward.[SNL12]

In AutoPas, since the focus is on getting the best possible result as fast as possible and not testing or choosing bad configurations, the first option does not fit very well. The second option offers various advantages as well, like not needing to know what an optimistic value for this specific simulation would be. Another benefit would be the immediate bias towards better-performing configurations. In addition, AutoPas tests every configuration in the first tuning phase to avoid the probability of only selecting bad or suboptimal configurations in

the first exploration phase.

4.4. Random Exploration

The problem with random exploration is that it is, first of all, it is arbitrary and does not magically select the best configurations to test. So there is a need for a broad enough coverage over all possibilities to ensure that it will include the optimal composition of options with a high enough certainty. But this comes with a trade-off between ensuring that the best configuration is tested and producing additional overhead.

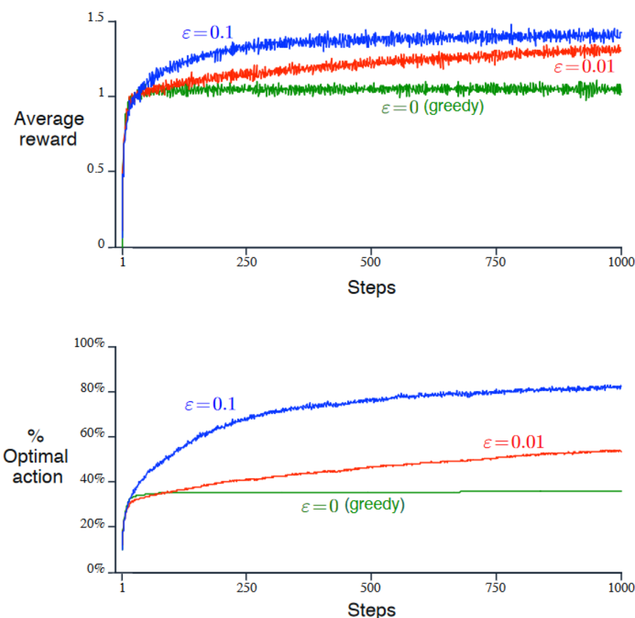
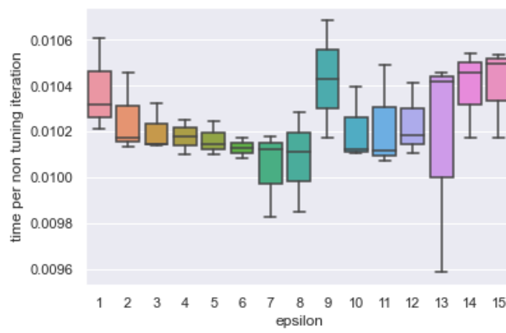


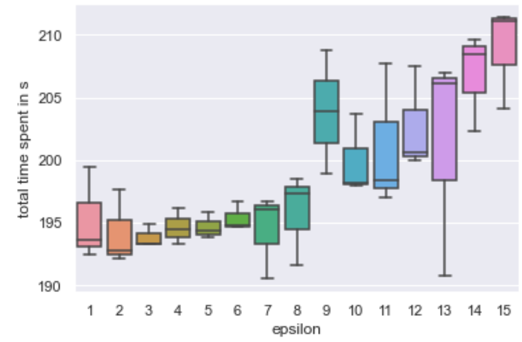
Figure 4.6.: Average performance of ” ϵ-greedy action-value methods on the 10-armed testbed. These data are averages over 2000 runs with different bandit problems. All methods used sample averages as their action-value estimates. [SB18]

As seen in Figure 4.6, the higher exploration rate selects better actions and reaches this faster, but the graph does not show how long each step takes. In AutoPas, the goal is to reduce time spent on the calculation of the simulation, which includes time spent in the tuning process. So there is a need to find a balance between selecting the best and spending less time on tuning.

To find a good trade-off, the parameter needs to be tuned since not optimal tuning has a high chance of suggesting worse configurations, as seen in Figure 5.4b. So the easiest way is to perform another grid search, but this time with the focus on the exploration parameter.



(a) Time per non tuning iteration



(b) Total time spent

Figure 4.7.: Timings with different ϵ values averaged over three runs.

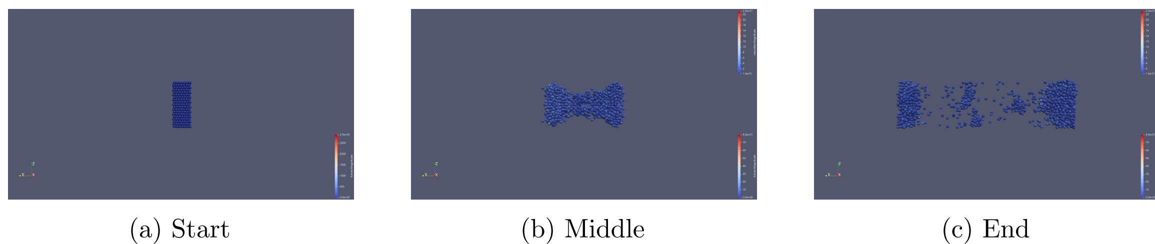
As seen in Figure 4.7a the exploration factor of seven offers the best timings for per non tuning iteration while not taking too long and even getting the best run once. Even though one to four have better timings overall, it can be explained by the fact that during the first run, all configurations are tested, and the best never being a bad choice during the whole simulation.

5. Comparison with Full Search and Predictive Tuning

All of the following simulations were run on my personal Desktop PC B.

5.1. Exploding liquid vs Falling drop

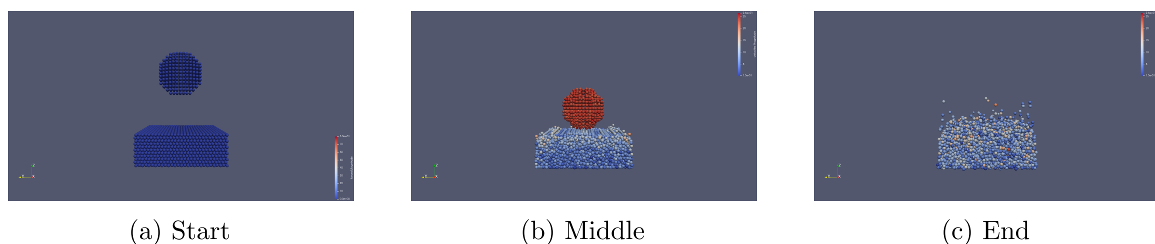
To compare the different tuning strategies, it is essential to know what exactly will be compared. The tests are done with two different types of simulations. First of all, the exploding liquid simulationA.2 simulates a compressed liquid in a rectangular box exploding outwards towards two opposite sides, as seen in Figure 5.1.



(a) Start (b) Middle (c) End

Figure 5.1.: Parts of the exploding liquid simulation

The second simulation is the falling drop simulation A.1 that recreates the falling of a drop into water as well as the following splash, as seen in Figure 5.2.



(a) Start (b) Middle (c) End

Figure 5.2.: Parts of the falling drop simulation

These two simulations were chosen because both of them have different characteristics. Falling Drop starts out with two bodies that then merge into one connected mass. Exploding liquid, on the other hand, starts with one entity that continues to separate into two significant and many small parts. These properties are essential for testing since they have varying requirements for the best configurations. Falling drop, for example, the optimal configuration only switches between two slightly different configurations 5.3.

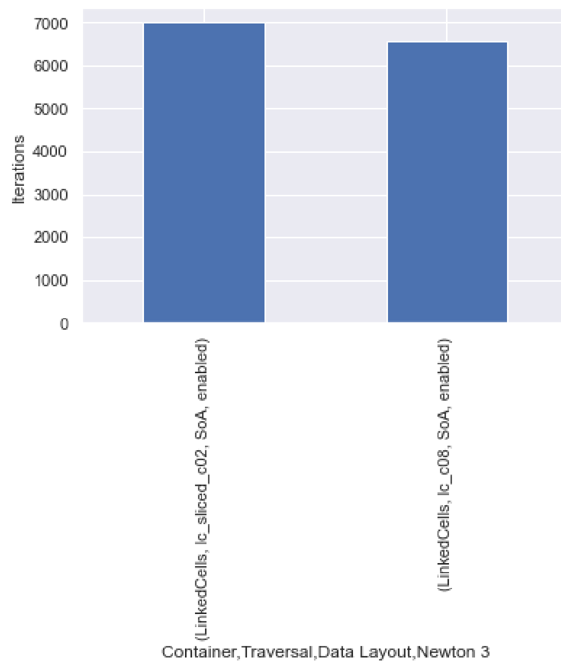
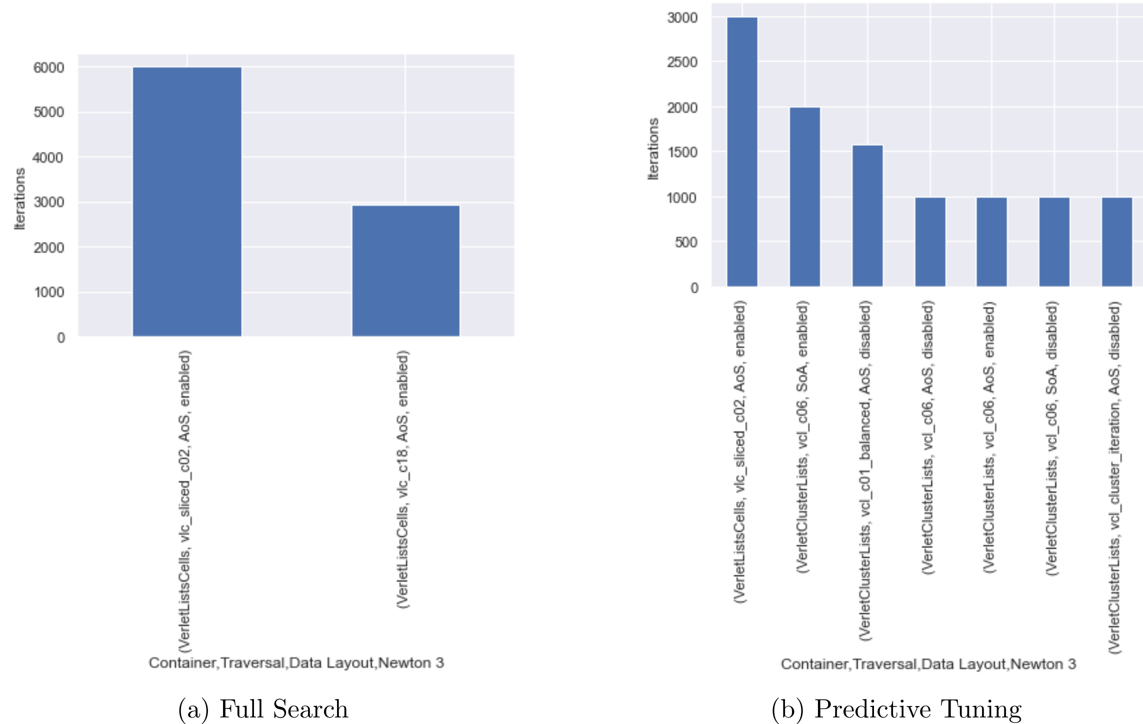


Figure 5.3.: Amount of iterations a specific configuration was used during the fallingDrop simulation

The exploding liquid, on the other hand, due to its chaotic nature, can suggest that a specific configuration is better even though this might only be the case for a few iterations or never since the composition changed beforehand 5.4. So why is that important for testing? Because it forces the tuning strategy to predict a wildly varying process and find the best configuration or a constant flow without requiring a considerable overhead for useless testing.



(a) Full Search

(b) Predictive Tuning

Figure 5.4.: Amount of iterations a specific configuration was used during the explodingLiquid simulation separated by the two different tuning strategies

The next step would be deciding what to test or compare the different tuning strategies. A good start would be the total time spent on the simulation, but this comes with a problem since it includes several constant factors that are not influenced by the tuning strategy, such as halo particle exchange or container updates.

The subsequent step would be the total time spent in the force calculation. This has the advantage of showing the overhead produced by tuning and can, therefore, especially in shorter or simulations with more tuning phases, favor strategies that choose a good enough configuration because time tuning is faster than the rest of the simulation.

Calculating the time spent per non tuning iteration would eliminate this favoring. Still, it would also show strategies that spent more time finding the best configuration in a better light. This shows the trade-off between less overhead during tuning phases by only testing a fraction of the configurations and finding the best configuration.

So both total time spent in the force calculation and time spent per iteration will be compared going forward.

explodingLiquid	fullSearch	predictiveTuning	reinforcementLearning
Total	15.159s	55.156s	12.157s
Tuning	8.721s	4.648s	4.430s
Non Tuning	6.436s	50.505s	7.725s
Per Non Tuning Iteration	$7.199 * 10^{-4}s$	$47.646 * 10^{-4}s$	$7.611 * 10^{-4}s$

Table 5.1.: Timings for the different tuning strategies with the explodingLiquid simulation.

The Table 5.1 shows that predictive tuning has a problem predicting and therefore selecting a good configuration. It even goes that far in choosing a terrible option. As seen in Figure 5.4b it decides to disable Newton3, which, as explained in 2.1.3, is very bad for performance. This could result from the nonoptimal tuning of the underlying extrapolating function since this can be recreated with a low exploration value. Full search already shows its good and bad points since it takes longer to tune than to actually run the simulation. Reinforcement learning is the fastest of the three regarding tuning time, but it shows that this is a trade-off with actual simulation time and tuning time since it takes 20% longer for the actual simulation.

fallingDrop	fullSearch	predictiveTuning	reinforcementLearning
Total	196.330s	170.509s	167.480s
Tuning	50.679s	19.136s	14.197s
Non Tuning	138.964s	144.794s	146.760s
Per Non Tuning Iteration	$1.024 * 10^{-2}s$	$1.006 * 10^{-2}s$	$1.010 * 10^{-2}s$

Table 5.2.: Timings for the different tuning strategies with the fallingDrop simulation.

The timings for the falling drop simulation in Table 5.2 further show the problems full search has since it spent more than two and a half times the time in the tuning phase than the other two strategies for an improvement of only about 5%. This time predictive tuning produces manages to not select terrible options and is only slightly behind reinforcement learning in total time needed. It even comes first in the time spent per non tuning iteration, which should be a full search since it should always select the best configuration in theory.

Predictive tuning shows that it can not handle some situations very well, especially the falling drop simulation. Even though full search has a considerable overhead through the testing of all possible configurations, it shows that selecting the best options can make a noticeable difference compared to selecting just a good one. But as seen in Table 5.2 choosing the configuration that is the fastest at the current iteration does not mean that it will still be true 100 iterations later, as shown by the time spent on non-tuning iterations, where reinforcement learning and predictive tuning beat out full search.

5.2. Performance with OpenMP

In recent years CPUs have hit a bottleneck regarding the speed of single core and therefore started to invest in increasing the overall speed of the computations by adding more cores and threads. Because of this, it is very uncommon for programs to run simulations without parallelization since it would just waste the rest of the CPU cores. The next test focuses on

the speed with OpenMP enabled in AutoPas, allowing the workload to be split into different threads.

explodingLiquid	fullSearch	predictiveTuning	reinforcementLearning
Total	17.015s	48.482s	11.542s
Tuning	10.499s	6.542s	4.634s
Non Tuning	6.514s	41.936s	6.905s
Per Non Tuning Iteration	$7.286 * 10^{-4}s$	$40.005 * 10^{-4}s$	$6,417 * 10^{-4}s$

Table 5.3.: Timings for the different tuning strategies with the explodingLiquid simulation and 12 threads.

Surprisingly full search is the only one that performs noticeably worse with 12 workers, which would suggest running into multiple race conditions, especially during the tuning phase. Predictive tuning also performs worse in the tuning phase but has improved timings since 12 threads can calculate the force updates faster, even with the hindrance of a bad configuration. Reinforcement learning is the only one that shows faster timings across the board and even manages to beat full search in time spent per non-tuning iteration.

fallingDrop	fullSearch	predictiveTuning	reinforcementLearning
Total	87.773s	64.398s	57.832s
Tuning	22.618s	8.291s	6.961s
Non Tuning	62.325s	53.655s	49.009s
Per Non Tuning Iteration	$0.460 * 10^{-2}s$	$0.371 * 10^{-2}s$	$0.337 * 10^{-2}s$

Table 5.4.: Timings for the different tuning strategies with the fallingDrop simulation and 12 threads.

Even though full search shows improvement in the multi-threaded version of falling drop, it has less improvement than the other two and this time took almost three times as long to find the best configuration. Even the time spent in non-tuning iterations gets beaten by both the predictive tuning and reinforcement learning strategies. Reinforcement learning also receives the most significant boost from using multiple threads in the falling drop simulation. It now is the fastest in total time spent as well as time spent per non-tuning iteration.

In conclusion, full search is terrible at handling race conditions and multi-threaded simulations in general. Even though predictive tuning shows clear improvement in the parallel version over the sequential one, it still suffers from the same problems. Reinforcement learning, on the other hand, only gets better through the introduction of multiple threads and increases its lead over the other two strategies.

Part IV.

Future Work and Conclusion

6. Future Work

There are multiple ways to further improve the implementation of this reinforcement learning. The first would be to substitute the algorithm with a more sophisticated one, like the previously mentioned QQ-learning method[Has10]. This variation provides faster converging while still finding the optimal path. This could be further refined by merging with the two other fields of machine learning.

The supervised field offers the option of deep Q-learning, which would train a neural network in the place of the q state values, that would even allow to pretrain a model and then skip the tuning phase exploration phase altogether.

On the other hand, the unsupervised field would be the merge with the, in predictive tuning proposed, linear regression, which did select better configurations during the testing 5.2. This algorithm is called GQ-Learning[MSBS10] and offers faster learning through the prediction of the action values.

Another point to improve the current algorithm would be to offer hyperparameter tuning. The current values are optimized for the tested simulations and might not be the best for other completely different scenarios. This could be done with different kinds of techniques such as gradient-based optimization, Bayesian optimization, or external frameworks[KO22].

Another point of improvement would be the collection of more data, such as the number of particles, a value representing the distribution of the particle in the room, and maybe the average force affecting a particle. This would allow, especially when also training a neural network for far better prediction and accuracy over the next iteration since the function now has insight into the simulation instead of selecting the optimal configuration based on a few values provided by a black box-like simulation environment.

7. Conclusion

This thesis aimed to determine if Reinforcement learning can be used to improve the autotuning process within AutoPas. To do that, it was first necessary to select the most fitting algorithm out of all the variations that reinforcement learning, especially the ones temporal difference, has to offer. Since none of them provided exactly what was needed, the SARSA algorithm was chosen due to its faster learning rate and safer prediction and was modified to fit better. The function was changed so that it no longer considers future rewards because these are not known yet to the tuner during the simulation process.

The next step was to tune the algorithm's hyperparameters to get the best possible results. The parameter that needed to be tuned were the learning rate, discount factor, and the exploration. The tuning was done by performing a grid search over all possible values.

To show that the reinforcement learning tuning strategy is actually an improvement, it was compared to the already existing tuning strategies, full search, and prediction tuning. The comparison was made with the two different simulations falling drop and exploding liquid, which offer a different particle distribution over a room. Falling drop is one continuous object, whereas exploding liquid provides a more random distribution of particles all over the possible space. The strategies were measured according to the total time spent and time spent in non-tuning iterations to update the force of the particles. In the single-threaded comparison, reinforcement learning showed a slight improvement over predictive tuning while being considerably better than full search. In the multi-threaded version, however, it managed to beat both tuning strategies by a considerable amount.

Part V.

Appendix

A. Yaml Configurations

Listing A.1: Falling drop yaml file with Reinforcement as tuning strategy

```
container                        :  [LinkedCells, VerletLists,
    VerletListsCells, VerletClusterLists]
verlet-rebuild-frequency         :  10
verlet-skin-radius               :  1.0
verlet-cluster-size              :  4
selector-strategy                :  Fastest-Absolute-Value
data-layout                      :  [AoS, SoA]
traversal                        :  [ lc_c01, lc_c18, lc_c08,
    lc_sliced_c02, vl_list_iteration, vlc_c01, vlc_c18, vlc_sliced_c02,
    vcl_cluster_iteration, vcl_c01_balanced, vcl_c06 ]
tuning-strategy                  :  reinforcement-learning
tuning-interval                  :  1000
tuning-samples                   :  3
tuning-max-evidence              :  10
functor                          :  Lennard-Jones-AVX2
newton3                          :  [disabled, enabled]
cutoff                           :  3
box-min                          :  [0, 0, 0]
box-max                          :  [7.25, 7.25, 7.25]
cell-size                        :  [1]
deltaT                           :  0.0005
iterations                       :  15000
boundary-type                    : [reflective,reflective,reflective]
globalForce                      :  [0,0,-12]
Objects:
  # "water"
  CubeClosestPacked:
    0:
      particle-spacing           :  1.122462048
      bottomLeftCorner           :  [1, 1, 1]
      box-length                 :  [48, 28, 10]
      velocity                   :  [0, 0, 0]
      particle-type              :  0
      particle-epsilon           :  1
      particle-sigma             :  1
      particle-mass              :  1
  Sphere:
    0:
      center                     :  [18, 15, 30]
      radius                     :  6
      particle-spacing           :  1.122462048
      velocity                   :  [0, 0, 0]
```

```
      particle-type                : 1
      particle-epsilon             : 1
      particle-sigma               : 1
      particle-mass                : 1
no-flops                           : false
no-end-config                      : true
log-level                          : info
no-progress-bar                    : True
```

Listing A.2: Exploding Liquid yaml file with Reinforcement as tuning strategy

```
container                          : [LinkedCells, VerletLists,
   VerletListsCells, VerletClusterLists]
traversal                          : [ lc_c01, lc_c18, lc_c08,
   lc_sliced_c02, vl_list_iteration, vlc_c01, vlc_c18, vlc_sliced_c02,
   vcl_cluster_iteration, vcl_c01_balanced, vcl_c06 ]
data-layout                        : [AoS, SoA]
newton3                            : [disabled, enabled]
verlet-rebuild-frequency           : 10
verlet-skin-radius                 : 0.2
verlet-cluster-size                : 4
selector-strategy                  : Fastest-Absolute-Value
tuning-strategy                    : reinforcement-learning
tuning-interval                    : 1000
tuning-samples                     : 10
functor                            : Lennard-Jones-AVX2
cutoff                             : 2
box-min                            : [0, 0, 0]
box-max                            : [15, 60, 15]
cell-size                          : [1]
deltaT                             : 0.00182367
iterations                         : 12000
periodic-boundaries                : true
Objects:
  CubeClosestPacked:
    0:
      box-length                   : [15, 6, 15]
      bottomLeftCorner             : [0, 27, 0]
      particle-spacing             : 1.
      velocity                     : [0, 0, 0]
      particle-type                : 0
      particle-epsilon             : 1
      particle-sigma               : 1
      particle-mass                : 1
no-flops                           : false
no-end-config                      : true
no-progress-bar                    : true
```

B. Simulation parameters

```
CPU       : I7-8700k (6 cores\12 threads) 4.3ghz-4.7ghz boost
Memory    : 16gb DDR4 3600mhz c16
Storage   : Samsung 960 evo
Compiler  : Clang10
OS        : WSL2 Ubuntu 20.04 running on Windows 10
```

List of Figures

2.1. The difference between AoS and SoA when loading values for force calculation. [GSBN22] 4
2.2. The red point is the selected main particle for the pairwise force calculation. The red circle is the cut-off radius for the Lennard Jones potential. The distance is calculated for every particle pair with an arrow, but the force calculation is only needed for the blue ones. The yellow circle is the Verlet-skin radius. The blue and red cells are being used in the neighbor list.[GSBN22] 5
2.3. Traversal patterns that prevent race conditions. The brown line indicates the end of the space and the outside cells are used for halo particles.[GSBN22] . 6
2.4. Model of the tuning process of the AutoPas library[GSBN22] 7
2.5. The already existing tuning interfaces within AutoPas. 7
2.6. SARSA vs Q-Learnig in high risk high reward situations.[SB18] 11

3.1. Inheritance of ReinforcementLearning. Helper function, function parameter, and return values are excluded to avoid cluttering. 13

4.1. Learning rate tuning with $\gamma = 0.8$ on the fallingDrop1 simulation A.1. . . . 17
4.2. Learning rate tuning with $\gamma = 0.7$ on the explodingLiquid simulation A.2. . 17
4.3. Line Plots of Train and Test Accuracy for a Suite of Learning Rates on the Blobs Classification Problem[Bro] 18
4.4. Learning rate tuning with $\alpha = 0.9$ on the fallingDrop1 simulation A.1. . . . 19
4.5. The effect of optimistic initial action-value estimates on the 10-armed testbed. Both methods used a constant step-size parameter, $\alpha = 0.1$. [SB18] 20
4.6. Average performance of " ϵ-greedy action-value methods on the 10-armed testbed. These data are averages over 2000 runs with different bandit problems. All methods used sample averages as their action-value estimates. [SB18] 21
4.7. Timings with different ϵ values averaged over three runs. 22

5.1. Parts of the exploding liquid simulation 23
5.2. Parts of the falling drop simulation 23
5.3. Amount of iterations a specific configuration was used during the fallingDrop simulation 24
5.4. Amount of iterations a specific configuration was used during the explodingLiquid simulation separated by the two different tuning strategies 25

List of Tables

4.1. Color of the rabbits separated by name 18
4.2. Color of a random rabbit 19

5.1. Timings for the different tuning strategies with the explodingLiquid simulation. 26
5.2. Timings for the different tuning strategies with the fallingDrop simulation. 26
5.3. Timings for the different tuning strategies with the explodingLiquid simulation and 12 threads. 27
5.4. Timings for the different tuning strategies with the fallingDrop simulation and 12 threads. 27

Bibliography

[Bro] Jason Brownlee. Understand the impact of learning rate on neural network performance. `https://machinelearningmastery.com/understand-the-dynamics-of-learning-rate-on-deep-learning-neural-networks/`. Accessed: 2022-09-12.

[GSBN22] Fabio Alexander Gratl, Steffen Seckler, Hans-Joachim Bungartz, and Philipp Neumann. N ways to simulate short-range particle systems: Automated algorithm selection with the node-level library autopas. *Computer Physics Communications*, 273:108262, 2022.

[Has10] Hado Hasselt. Double q-learning. In J. Lafferty, C. Williams, J. Shawe-Taylor, R. Zemel, and A. Culotta, editors, *Advances in Neural Information Processing Systems*, volume 23. Curran Associates, Inc., 2010.

[HS99] Geoffrey Hinton and Terrence J. Sejnowski. *Unsupervised Learning: Foundations of Neural Computation*. The MIT Press, 05 1999.

[JWCW15] Xiaolong Jin, Benjamin W. Wah, Xueqi Cheng, and Yuanzhuo Wang. Significance and challenges of big data research. *Big Data Research*, 2(2):59–64, 2015. Visions on Big Data.

[KO22] Mariam Kiran and Buse Melis Ozyildirim. Hyperparameter tuning for deep reinforcement learning applications. *CoRR*, abs/2201.11182, 2022.

[KOK05] Takashi Kuremoto, Masanao Obayashi, and Kunikazu Kobayashi. Nonlinear prediction by reinforcement learning. In *Advances in Intelligent Computing*, volume 3644, pages 1085–1094, 08 2005.

[MKS+15] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A. Rusu, Joel Veness, Marc G. Bellemare, Alex Graves, Martin Riedmiller, Andreas K. Fidjeland, Georg Ostrovski, Stig Petersen, Charles Beattie, Amir Sadik, Ioannis Antonoglou, Helen King, Dharshan Kumaran, Daan Wierstra, Shane Legg, and Demis Hassabis. Human-level control through deep reinforcement learning. *Nature*, 518(7540):529–533, February 2015.

[MSBS10] H.R. Maei, Cs. Szepesvári, S. Bhatnagar, and R.S. Sutton. Toward off-policy learning control with function approximation. In J. Fürnkranz and T. Joachims, editors, *ICML*, pages 719–726. Omnipress, June 2010.

[MVC+10] Martin Midtgaard, Lars Vinther, Jeppe R. Christiansen, Allan M. Christensen, and Yifeng Zeng. Time-based reward shaping in real-time strategy games. In Longbing Cao, Ana L. C. Bazzan, Vladimir Gorodetsky, Pericles A. Mitkas,

Gerhard Weiss, and Philip S. Yu, editors, *Agents and Data Mining Interaction*, pages 115–125, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg.

[NC23] I. Newton and R. Cotes. *Philosophiae naturalis principia mathematica*. Sumptibus Societatis, 1723.

[oxf00] Oxford english dictionary, 2000.

[Pel20] Julian Mark Pelloth. Implementing a predictive tuning strategy in autopas using extrapolation. Bachelorarbeit, Technical University of Munich, Sep 2020.

[RN10] Stuart Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach*. Prentice Hall, 3 edition, 2010.

[SB18] Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning: An Introduction*. The MIT Press, second edition, 2018.

[Smi15] Leslie N. Smith. No more pesky learning rate guessing games. *CoRR*, abs/1506.01186, 2015.

[SNL12] Hanan Shteingart, Tal Neiman, and Yonatan Loewenstein. The role of first impression in operant learning. *Journal of experimental psychology. General*, 142, 08 2012.