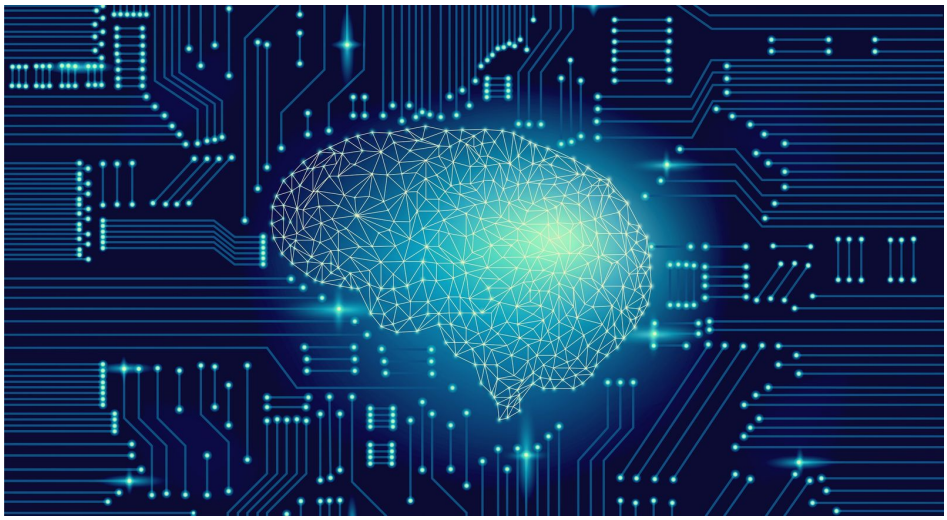


Thesis: DRL

Julian Fransen

December 2022

Abstract

In new physics, the models have large parametric spaces. Finding the islands of correct solution is an important problem. This thesis attempts to solve this problem using deep reinforcement learning. The parametric space problem is modelled as the game Battleship, where there are 'islands' of correct solutions, where the agent gets a reward, and in the remainder of the field the agents gets no reward. An algorithm is trained using q-learning, and it creates its own strategy from scratch. The goal of the algorithm is to maximize the reward by finding the islands as fast as possible. It was found that it is possible, at least in a small parameter space, and that a combination of supervised learning and reinforcement learning yields the best result.

Contents

1 Standard Model

The standard model involves describing all fundamental particles as waves in fields, where every type of particle has its own associated field. This is all described by quantum field theory. Particles can have interactions with each other, where both energy and momentum can be exchanged. The standard model takes a total of 19 parameters [9], which are for example the masses of particles and the strength of their interactions. The values for most of these parameters are known precisely, but not all. Regardless of that, the standard model is extremely successful in making predictions. It is able to predict particle physics experiments very precisely and it predicted the existence of undiscovered particles [11].

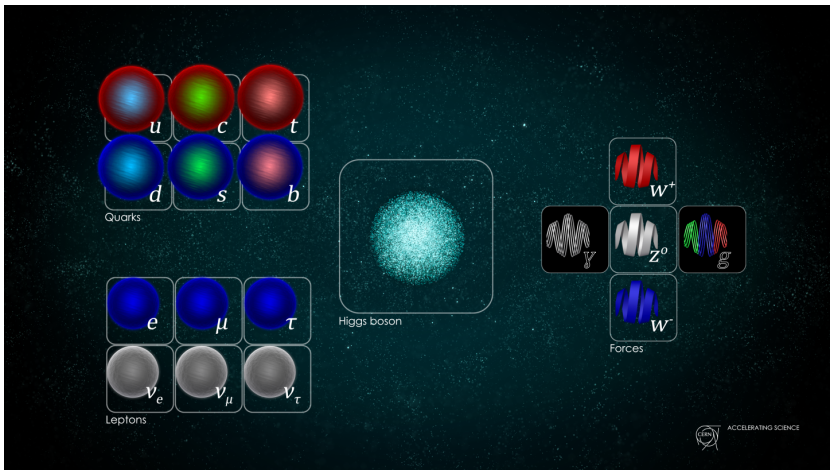


Figure 1: The standard model of particle physic

Although successful, the standard model is believed to be an incomplete description, due to certain experimental reasons. For example, the imbalance between matter and antimatter is not explained. Moreover, the existence of dark matter is not described by the standard model. Hence, physicists search beyond the standard model. One of the most investigated extensions to the standard model is Supersymmetry (SUSY). Supersymmetry predicts the existence of a symmetry operator $\hat{Q}$, that transforms fermions to bosons and vice versa. The idea is that this transformation only alters the spin of the paricles. The existence of the operator $\hat{Q}$ would imply the existence of new particles and fields, hence it is an extension to the standard model. There are multiple models in supersymmetry theories, for example MSSM, pMSSM and mSUGRA [5].

But there is a problem. Many models describing new physics have large parameteric spaces, including different models of SUSY. They have a dimensionality of the parameter space ranging up to 100 [1]. Furthermore, for effective field theories this number can be even larger [6]. Certain regions in the parameter space yield correct results, such as the correct Higgs mass. These islands

are places of interest: the real solution for the parameters could be somewhere in these islands. Thus, the goal is to find the complete islands and to find all islands in the parameter space. In the figure below there is depicted a simplified version of the problem. Here there are only 2 theoretical parameters θ and the z-axis is the likelihood of the parameters θ given the data: $L(\theta|X)$. As you can see, some values of θ have a very low likelihood, because they yield wrong values, for example for the Higgs mass.

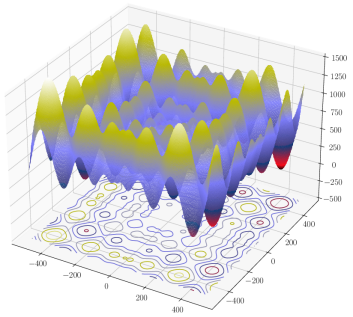
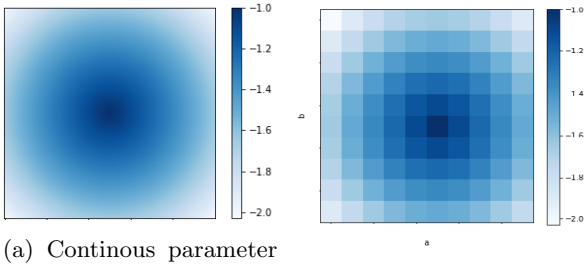


Figure 2: The likelihood of the parameters θ given the data: $L(\theta|X)$ for a simplified physical problem [1]

The first simplification of the problem is treating the continuous parameter space as a discrete one. This is done due to the fact that algorithms with a discrete output space are much simpler. By increasing the resolution of the parameter space, i.e. going for 100x100 instead of 10x10 in the figure above, a problem resembling original one is obtained. The islands of correct solutions



(a) Continous parameter space

(b) Discrete parameter space

Figure 3: Discretization of a function

would then consist of multiple discrete blocks close to each other, as in figure 4. In this figure the red blocks are the correct solutions and the white blocks are the rest of the field. As you can see, the correct solution blocks are clustered together as islands. At this point, the problem resembles the well-known game of 'Battleship'.

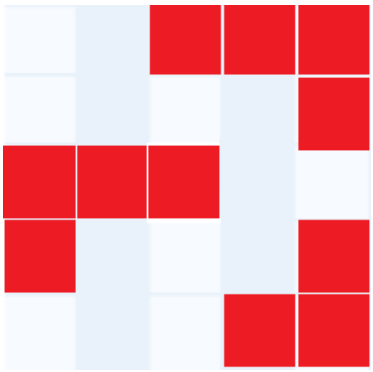


Figure 4: Discrete islands of correct solutions

1.1 Solving Battleship

Battleship is a board game in which the objective is to sink the other player's battleships [2]. At the start of the game, each player places their own ships in the 10 by 10 field. Then the players try to hit the opponents ship by naming a certain square, for example 'C6', which would be colums 3 and row 6. If the opponent has placed a part of a ship on this square, it's a hit. If not, it's a miss. The player who first sinks all ships of the opponent by hitting every square on which there is a ship, wins the game.

Figure 5: The Battleship gameboard

This game is related to the physics problem in two important ways. First, there are discrete 'islands of correct solutions': the ships. This means that if you shoot and it's a hit, its likely that there is another hit close by, just as in the physics problem, where correct solutions are clustered together. Furthermore,

the islands have roughly the same shape and size. Second, there are multiple islands of correct solutions: multiple ships. This means that after a target has been found the algorithm should search for new targets.

If we find an algorithm that can solve Battleship, it should be able to solve the problem in higher dimensions as well, which potentially means it could solve the original parameter space problem. The aim of this thesis is to solve a simplified version of this general problem. We will attempt to solve a simple version of Battleship with a small field to keep the computation time low. If N is the number of blocks in the field, we assume that the problem will scale with N or faster. We will be using Deep Reinforcement learning to train this algorithm. Hence the research question:

Can an algorithm trained by deep reinforcement learning find the global maxima in fewer iterations than a random strategy in a non-differentiable problem?

2 Approaches

To find these islands of partially correct solutions, there are many possible methods. In this section a few are mentioned, and some more information about Machine Learning is given.

2.1 Evolutional Strategies

Evolutional algorithms are population-based optimization algorithms which implement phenomena from biology, such as survival of the fittest, mutation and selection. The individuals in the populations are the possible solutions to the optimization problem, and the fitness function or loss function determines the quality of the solutions.

One example of an evolutional algorithm is the Particle Swarm Optimization [12]. This computational method lets a population of particles navigate the search-space. Each particle has a position and a velocity, and its direction is influenced by its best position and by the best position known by the swarm. In this context the 'best position' means the highest likelihood. It is expected that the swarm moves toward the direction of the correct solution.

Another approach is the use of Artificial Bee Colonies. The basic idea behind the Bee Colony Optimization is to create a multi-agent system, namely the colony of artificial bees, capable to solve complex combinatorial optimization problems. The algorithm is inspired by the behaviour of honey bees as they are searching for their food: nectar. In the Artificial Bee Colony algorithm there are N active data points which are tracked [1]. Initially the locations of these points are uniformly distributed. All function values for the data points are evaluated. In the bee-analogy the locations of nectar are the data points and their function value is the food gain from these sources. The goal of the optimization algorithm is , of course, to find the data point with best function value. Or, in the bee-

analogy, to find nectar source with highest food gain. The process of finding this point is iterative, and every iteration consists of three phases. First, new locations are explored and their function values are calculated. The locations of the new data points are determined by starting at an original active data point and then traveling some distance to one of the other active data points in all dimensions. If the function value of this new location is better than the original one, it gets replaced. However, the old location and function value of the data point are saved in case the test fails. This is part of the second phase, where locations that fail too many times are reinitialised. In the final step all active data points are assigned a fitness-score, which is dependent on the functional value. Based on this fitness-score an update probability is calculated for each data point. Similarly to the first phase, the data points are only updated if the new point improve the function value.

One variation of one of these algorithms is the Artificial Bee Colony (ABC) algorithm developed by Karaboga [10]. After comparing the performance of the ABC algorithm with different algorithms, the authors concluded that this algorithm has the ability to get out of a local minimum and that it can be efficiently used for multivariable, multimodal function optimization. This algorithm was applied to train feed-forward artificial neural networks, where the authors compared performances with the back propagation algorithm. It showed that the ABC algorithm could be good addition to the existing algorithms.

2.2 Machine learning

The idea behind machine learning is that algorithms learn and improve themselves, instead of being explicitly programmed behave in a certain way. This enables machine learning to be broadly applicable in many different fields of expertise. There exist three main branches: reinforcement learning, supervised learning and unsupervised learning. In this thesis only the first two techniques are used.

2.2.1 Neural nets

Artificial neural networks or neural nets (NNs) are algorithms inspired by the dynamics of networks of neurons found in animals. The neural nets consist of a number of layers of neurons, and every neuron usually connects with all neurons in the next layer. The biases in every neuron and the connection strength between neurons known as weights are trainable parameters. In training the neural networks are presented an input-output pair. Using the input, the NN makes a prediction. The error between this prediction and the true output is calculated using the error function. Then, using backpropogation, the gradient of the error E with respect to every trainable parameter p is calculated:

$$\frac{\partial E}{\partial p}, p \in P$$

with P defined as the collection of all trainable parameters. Finally, we utilize the gradient together with an optimization algorithm to change the trainable parameters, reducing the error and improving the prediction done by the NN.

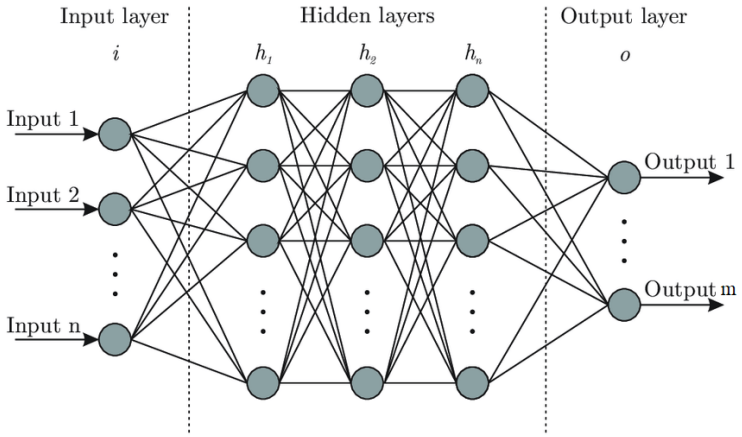


Figure 6: General neural network architecture.

2.2.2 Reinforcement learning

Reinforcement learning tackles problems which can be modeled as a Markov decision process (MDP) [4]. This is a process involving decision making at discrete timesteps, where outcomes aren't fully determined by the decision, but also partly random. Hence MDPs are discrete-time stochastic processes. It is very useful to model a problem as a MDP due to the mathematical framework that has been established [3]. The following list contains the most important terms for the general model of a problem.

- State(S_t): the state contains all observable information of the environment at time t.
- Action space(A): the set of all possible actions. The agent outputs an action a_t at time t with $a \in A$.
- Reward: The reward $R_a(S, S') = r_t$ is a function which gives the reward of going from state S at time $t-1$ to state S' at time t.
- Agent: the player of the game. The output is the chosen action and it essentially maps a (S_t, r_t) to a.

2.2.3 Q-learning

Q-learning is an algorithm of reinforcement learning where the agent in the prototypical problem is a q-table. This table is filled with q-values: the expected

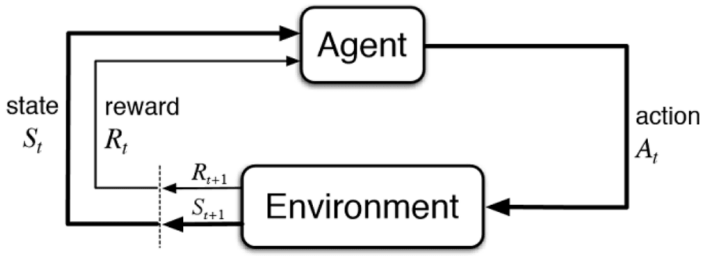


Figure 7: An overview of the mechanism of behind deep reinforcement learning.

reward of every action given a state. Every row describes a different state and every column describes the q-value for an action. Thus, for a problem with N possible states and M possible actions the Q-table will be of size $N \cdot M$. Typically the initial values of the table are equal, to avoid favouring certain actions over others. Every iteration information about one particular q-value is acquired: if the Environment is in state $s_t = s_i$ and the chosen action is $a_t = a_j$ then information about q-value q_{ij} is received. A complete Q-table contains the q of every action for every state. The Q-values are updated using the following formula [7]:

$$Q^{new}(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha \cdot (r_t + \gamma \cdot maxQ(s_{t+1}, a) - Q(s_t, a_t))$$

The new q-value is consist of the sum of three terms. The first one, $(1 - \alpha)Q(s_t, a_t)$, is the current q-value weighted by the learning rate α. $\alpha \in [0, 1]$, higher values for α change the q-value more rapidly. The second term, αr_t, is the reward received at time t weighted by the learning rate. Finally, the last term $\alpha\gamma \cdot \max Q(s_{t+1}, a)$: the maximum reward that can be obtained from the next state, (weighted by learning rate and discount factor γ). The discount factor is a measure of how important the future rewards are with respect to the current reward.

2.2.4 Deep reinforcement learning

Deep reinforcement learning is a powerful technique which is however rarely used in particle physics. This is due to the fact that this framework is essentially designed to solve games. By rephrasing the particle physics problem as a game, we obtain a unique approach to the problem. In deep reinforcement learning the agent is a neural net, where the given input is the state and the output are the q-values of all possible actions in that state. After the neural net is trained, the best strategy would be to always pick the action with the greatest q-value. The training data this NN is trained on is not labeled data. This is also the strength of DRL: it can tackle problems without knowing what the correct output is. The training data is generated through the algorithm playing the game. By tracking all observables, the NN learns to favour actions that lead to

a high reward and to avoid actions that lead to a small reward. The input to the NN is the state of the Environment. Then the output of the NN will be the predicted q-values. In training, these predicted values will be compared to the 'true' values, which are calculated using the aforementioned q-value formula.

2.2.5 Supervised learning

In supervised learning the true outputs are known and the NN will behave exactly like the data after sufficient training. It essentially learns to map an input to an output. The flaw of SL is that it requires labeled data, and usually also a large amount of data.

3 Methods

Everything was programmed in the Python language, using the TensorFlow library with Keras as the interface for Python. The full code will be in the appendix.

3.1 Simplified Battleship

After some experimentation it was decided that we would use a even more simplified version of Battleship. It turned out that the original field of 10 by 10 blocks was too big to converge to a solution quickly. Our simplified version of Battleship works as follows: the field is 5 blocks wide and there is one target which is 2 blocks wide. This means there are 4 possible setups of the field, as you can see in figure 8. The player can shoot three times, and its aim is to

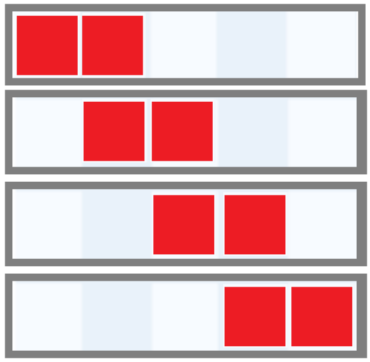


Figure 8: The 4 possible target positions.

score as many points as possible. Every turn the player can earn -100, 0 or 100 points. If the target is hit, the player earns 100 points for that turn. If a block is hit two or more times, the player earns -100 points: it is against the rules to shoot a block multiple times. If a block is hit for the first time but it is not one of the target blocks, 0 points are earned.

We decided to simplify the problem further for two reasons. The first is the reduced computation time. The second one is the fact that we can compare the strategy and results of the DRL algorithm to both our own strategy and the random stategy. Our strategy is the so-called 'hardwired strategy' and it is designed in such a way that is maximizes the reward: there is no solution better on average than this strategy. The most left box is box number 1 and the most right one is box number 5. This is how this strategy works: first, shoot box number 3 (the middle box). Next, shoot box number 4. If the last shot was a hit, shoot 5. Else, shoot 2. Using this strategy 3 out of 4 times the maximum score is reached, and the remaining time a score of 100 is reached, which comes down to an average score of 175. The other strategies will be compared with this one to see how well they perform.

3.2 Supervised approach

In testing, it turned out that using only q-learning yielded disappointing results: the algorithm did not even seem to understand that shooting the same box multiple times yields a penalty. This is a limitation on reinforcement learning: in some context it performs badly. However, by first using a form of supervised learning followed by reinforcement learning overcomes this problem. This is for example what the researchers of DeepMind did when they developed a chess, shogi and go playing algorithm named AlphaZero [8]. We will focus on their chess playing algorithm and its training. First, the neural net was trained on many games by highly rated players using supervised learning. After this, the neural net was improved further using reinforcement learning. This method led to this algorithm dominating the other chess playing algorithms. Inspired by DeepMind, we attempted to do the same: first supervised learning, then reinforcement learning. However, for the supervised learning much labeled data is necessary. This is why we generated this data separately. We also wanted to test the algorithm in between training. That is why the code is written in the following way:

1. training data is generated
2. the agent is trained using supervised learning
3. the agent is tested
4. the agent is trained using reinforcement learning
5. the agent is tested again

To generate the training data, the agent plays the game between 100 thousand and 1 million times. The states, actions and rewards are all saved to an .csv file.

The agent is programmed as a neural net. As input it receives the state of the game, which contains the 5 boxes (a 1 means the box has been shot, and a 0 means it has not been shot), it contains the current reward (-100, 0 or 100) and it contains the total reward until that point in the game. Based on the state the agent can make a calculated decision what box to shoot next. The output

of the agent are the q-values for the boxes. It will be a 5x1 array with all values summing up to 1.

The NN is fully connected with a hidden layer of 50 units. The loss function is the mean squared error function, used to calculate the error between the prediction of the NN given an input and the label corresponding with this input. All network weights are initialized with zeros for q-learning, and for supervised learning they are initialized using a random normal distribution.

The agent can be trained using supervised learning, q-learning, or both. For supervised learning the input data is the state, and the labels are the actions. The action of choosing the middle box would for example look like [0,0,1,0,0]. For reinforcement learning it is a bit more complicated to obtain the labels. Below is the python code that calculates the labels required for deep reinforcement learning. The input-output pairs are (X,y).

```
for index, state in enumerate(states):
        action = np.argmax(actions[index])
        # The reward is the reward the agent got this iteration
        reward = rewards[index]
        # future_qs_list contains the q-values of the next state
        max_future_q = np.max(future_qs_list[index])
        # Calculate Q-value using the formula
        new_q = reward + DISCOUNT * max_future_q
        # Update Q value for given state
        current_qs = current_qs_list[index]
        current_qs[action] = new_q
        # Normalise Q
        current_qs = np.array(current_qs)/sum(current_qs)
        # And append to our training data
        X.append(state)
        y.append(current_qs)
```

Figure 9: In this figure the labels required for reinforcement learning are calculated using the q-learning formula from section 2.2.3. The output of the neural net after given input are q-values. In training these are compared with the labels. The labels are the original values with a small difference: the q-value of the chosen action is updated based on the reward the agent received after performing this action

4 Results

In testing it turned out that only using QL to train worked very poorly: the average scores were negative, which means that shooting completely random was still much better. This is why the final results are of models which are first trained using SL, tested, and then improved further using QL. In total 5 different strategies were tested.

- random: pick one of the 5 boxes at random
- pseudo-random: pick one of the boxes that have not been shot
- SL model 1: model trained on pseudo-random data
- SL model 2: model trained on filtered pseudo-random data
- SL model 3: model trained on hardwired strategy

The first one was completely random: regardless of the past turn, every time the agent chooses one of the five boxes at random. This strategy earns on average a score of 42.8, which is quite low, due to the fact that it often earns -100 points.
The second is called 'pseudo-random': it is still random, but now the agent knows the rules of the game, which means that it never shoots the same box twice. This strategy earns on average a score of 124. The model which was trained using SL on this pseudo-random data is called 'SL model 1' and earned a score of 125 after SL. Next, after further improvement using RL the score reached 150.
The third strategy is the model trained on the pseudo-random data and further improved by reinforcement learning. After supervised learning it earned a similar score to the pseudo-random strategy, which was 125 points. The reinforcement learning made the score improve until 150 points.
The fourth strategy worked in the following way: use the pseudo-random data but filter it; only keep in the trials that reached the maximum score of 200. The idea is that a model trained using supervised learning on this data would mimic the good behaviour of the strategy. For example, the middle three blocks have a higher probability of being a target, therefore it is better to shoot one of these blocks the first time you shoot. The prediction is then that this model behaves this way, and this is exactly what we see. This model, named 'SL model 2', earned on average a score of 150, and after RL a optimal score of 175.
The final strategy is the so-called 'hardwired strategy'. The model which was trained on this training data is called SL model 3 and earned on average a score of 175. In every way it behaved like the hardwired strategy, essentially it was a copy. Since the optimal score was already reached, this model was not trained using RL.

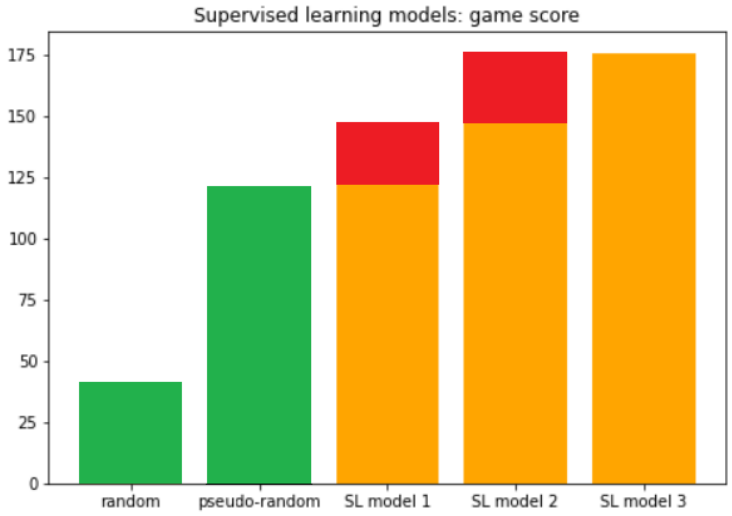
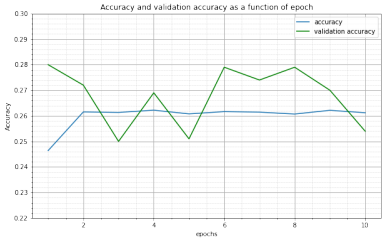
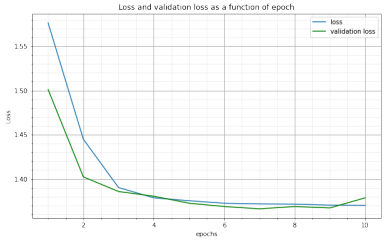


Figure 10: Summary of results. The score after SL is depicted in orange and the increase in score due to RL is depicted in red. The score of the random and pseudo-random strategies are depicted in green.

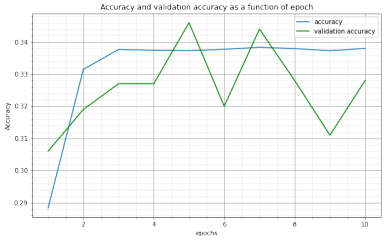
In these next plots you can see the loss and accuracy of the neural net after supervised learning. As expected, in all three cases the accuracy increases and the loss decreases as a function of epochs, since these two are inversely related. Furthermore, the validation accuracy and losses are distributed around the calculated value. In figure 11 a) you can see that the accuracy converges to a value of 0.26. This is quite low, due to the fact that it is impossible to predict a random move. In the first state of the game, no shots have been fired, and the chance of guessing the right box is 1/5. The next turn, it is 1/4 and the last turn it is 1/3. Thus, the average accuracy is $\frac{1/5+1/4+1/3}{3} = 0.26$. In figure 11 c) the accuracy is a bit higher, because there is more actual strategy behind the filtered data and therefore is less random. Finally, in figure 11 e) the accuracy is 100 percent due to the fact that the hardwired strategy is fully determined. Also due to this, the loss is at a value of zero. After 2 epochs the NN is optimized and fully captures the behaviour of the hardwired strategy.



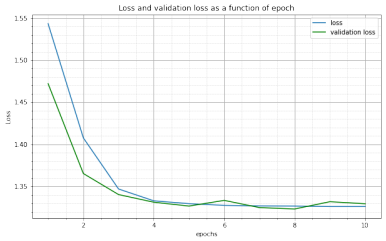
(a) pseudo-random data accuracy plot



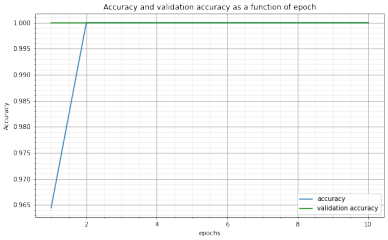
(b) pseudo-random data loss plot



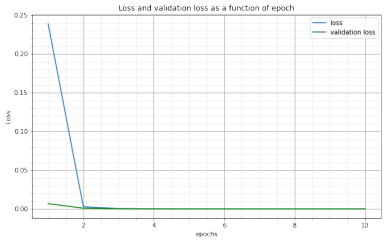
(c) filtered data accuracy plot



(d) filtered data loss plot



(e) hardwired data accuracy plot



(f) hardwired data loss plot

Figure 11: Supervised Learning loss and accuracy plots

5 Summary and overlook

The models that are dealt with in particle physics have very large parametric spaces. These spaces are in fact so large that it is infeasible to try out all points. The objective in particle physics is to find all and the complete islands in this parameter space that yield correct solutions. In this thesis we attempted to do exactly that using different techniques of machine learning, including deep reinforcement learning. Hence the research question:

Can an algorithm trained by deep reinforcement learning find the global maxima in fewer iterations than a random strategy in a non-differentiable problem?

In our version of the problem, where the field consisted of 5 boxes and the target consisted of 2 boxes, the algorithm was able to perform better than random. We introduced 2 random strategies, and after training our algorithm performed better than both. We obtained the best performing algorithm by doing the following:

1. generate training data using a random strategy
2. separate training data on score; save the trials where the maximum score was reached as 'filtered data'
3. create the agent and its neural network, train using supervised learning on the generated data
4. train this neural network further using reinforcement learning on all training data

Further research is needed to find out if reinforcement learning can be used in particle physics. Our problem could be expanded to make it closer to the actual problem in particle physics. First, the battleship field could be made bigger, e.g. 10 or 100 blocks per dimension. Second, multiple dimension could be used, for example 12-dimensional. Third, multiple targets or ships could be introduced. Finally, the ships could be made in slightly different shapes and sizes. If all of this is done, the Battleship-problem would be very close to the actual problem and its solution could be used. The challenge with expanding the problem lies in computation time. As the field becomes bigger and thus the problem more complex, the training time will increase.

References

[1] Csaba Balázs et al. A comparison of optimisation algorithms for high-dimensional particle and astrophysics applications. *JHEP*, 05:108, 2021.

[2] Eva-Maria Hainzl, Maarten Löffler, Daniel Perz, Josef Tkadlec, and Markus Wallinger. Finding a battleship of uncertain shape, 2022.

[3] Ronald A Howard. *Dynamic Programming and Markov Processes*. "[Cambridge] : Technology Press of Massachusetts Institute of Technology", 1960.

[4] Sumeet Khatri. On the design and analysis of near-term quantum network protocols using Markov decision processes. July 2022.

[5] Stephen P. Martin. A Supersymmetry primer. *Adv. Ser. Direct. High Energy Phys.*, 18:1–98, 1998.

[6] Shigeki Matsumoto, Satyanarayan Mukhopadhyay, and Yue-Lin Sming Tsai. Singlet majorana fermion dark matter: a comprehensive analysis in effective field theory. *Journal of High Energy Physics*, 2014(10), oct 2014.

[7] Francisco S. Melo. Convergence of Q-learning with linear function approximation. 2007.

[8] David Silver, Thomas Hubert, Julian Schrittwieser, Ioannis Antonoglou, Matthew Lai, Arthur Guez, Marc Lanctot, Laurent Sifre, Dharshan Kumaran, Thore Graepel, Timothy Lillicrap, Karen Simonyan, and Demis Hassabis. A general reinforcement learning algorithm that masters chess, shogi, and go through self-play. *Science*, 362(6419):1140–1144, 2018.

[9] Bonifatius Paulus Josephus Stienen. *Working in high-dimensional parameter spaces - Applications of machine learning in particle physics phenomenology*. PhD thesis, Nijmegen U., 2021.

[10] Dušan Teodorović. *Bee colony optimization (BCO)*, volume 248, pages 39–60. 10 2009.

[11] Mark Thompson. *Modern particle physics*. University of Cambridge, 2013.

[12] Tingting Zhang, Yongjie Sun, Pengpeng Wang, and Cunguang Zhu. Concentration retrieval in a calibration-free wavelength modulation spectroscopy system using particle swarm optimization algorithm, 2022.

A Appendix: Python code

```
import tensorflow as tf
import math
import numpy as np
import matplotlib.pyplot as plt
import keras
import os
import random

from tensorflow.keras.datasets import mnist
from tensorflow.keras.optimizers import Adam
from tensorflow.keras.initializers import RandomNormal
from tensorflow.keras.layers import Conv2D, Flatten, Dense,
↪  Dropout, MaxPooling2D, Input
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Reshape
from tensorflow.keras.losses import MeanAbsoluteError

import tensorflow.keras.backend as kb

#MACROS
FIELD_SIZE = 5
GAMMA = 0.8
NUMBER_OF_EPISODES = 100
DIMENSION = 1
STATE_VECTOR_SIZE = 1
SIGMA = 10
DISCOUNT = 0.8
EPOCHS = 10

epsilon = 1
EPSILON_DECAY = 0.999975
MIN_EPSILON = 0.001
RESET_ENV_EVERY = 3
UPDATE_TARGET_EVERY = 5
TRIALS_PER_BATCH = 1000
MIN_AVERAGE_SCORE = 170
MIN_TEST_SCORE = 100
TAKE_AVERAGE_EVERY = 100
AMOUNT_TRIALS_TESTED = 100

# Definitions
class Environment:

    def __init__(self):
        self.target = np.random.randint(0,FIELD_SIZE**DIMENSION-1)
        self.current_state = np.array([0,0,0,0,0])
```

```
        self.memory = []
        self.final_reward = []
        self.episode_number = 0
        self.reward = 0
        self.episode_reward = 0
        self.full_state = np.array([[0,0,0,0,0,0,0]])

    def reset(self):
        self.target = np.random.randint(0,FIELD_SIZE**DIMENSION-1)
        self.current_state = np.array([0,0,0,0,0])
        self.reward = 0
        self.episode_reward = 0
        self.episode_number += 1
        self.full_state = np.array([[0,0,0,0,0,0,0]])

    def moving_battleship_reward_function(self, action):
        action = np.argmax(action)
        if self.current_state[action]==1:
            return -100
        if action == self.target or action == self.target + 1 :
            return 100
        else:
            return 0

    def update_env(self, action):
        self.reward =
        ↪  self.moving_battleship_reward_function(action)
        self.episode_reward += self.reward
        if self.reward >= 0:
            self.current_state = action + self.current_state
        self.full_state =
        ↪  np.array([np.append(self.current_state,[self.reward,
        self.episode_reward])])
        self.memory.append(self.full_state)

    def extract_memory(self):
        return np.array(self.memory)

class Agent:
    epsilon = 0
    EPSILON_DECAY = 1
    MIN_EPSILON = 0.01
    triple_same_state_counter = 0
    optimizer = Adam(learning_rate = 0.001, epsilon = 0.1)

    def __init__(self):
        self.model = self.create_model()
        self.target_model = self.create_model()
        self.target_model.set_weights(self.model.get_weights())
```

```
        self.target_update_counter = 0
        self.epsilon = 1

    def create_model(self):
        model = Sequential()
        model.add(Input(shape = (7,)))
        model.add(Dense(10, kernel_initializer = 'RandomNormal'))
        model.add(Dense(FIELD_SIZE**DIMENSION,kernel_initializer =
        ↪  'RandomNormal',activation='softmax' ))
        model.compile( optimizer= Adam(learning_rate = 0.0001,
        ↪  epsilon = 0.1),  loss ='categorical_crossentropy' ,
        ↪  metrics= ['Accuracy'])
        return model

    def predict_action(self, state):
        output = np.array([0,0,0,0,0])
        action = self.model.predict(state)
        action = np.argmax(action)
        output[action] = 1.0
        return output

    def choose_action(self,input):
        qs = self.model.predict(input)
        action = np.argmax(qs)
        while 1.0 == input[0][action]:
            qs[0][action] = -1
            action = np.argmax(qs)
        output = np.array([0,0,0,0,0])
        output[action] = 1.0
        return output

    def supervised_training(self, X, y):
        number = int(len(X)*.98)
        number2 = int(len(X)*.02)
        X_train = X[:number]
        X_val = X[-number2:]
        y_train = y[:number]
        y_val = y[-number2:]
        history = self.model.fit(X_train, y_train, verbose =
        ↪  1,epochs=EPOCHS,  batch_size = TRIALS_PER_BATCH,
        ↪  validation_data = [X_val,y_val],validation_steps=1)
        return history

class QAgent:
    epsilon = 0
    EPSILON_DECAY = 1
    MIN_EPSILON = 0.01
    triple_same_state_counter = 0

```

```
    def __init__(self):
        self.model = self.create_model()
        self.target_model = self.create_model()
        self.target_model.set_weights(self.model.get_weights())
        self.target_update_counter = 0
        self.epsilon = 1

    def create_model(self):
        model = Sequential()
        model.add(Input(shape = (7,)))
        model.add(Dense(10, kernel_initializer = 'Zeros'))
        model.add(Dense(FIELD_SIZE**DIMENSION,kernel_initializer =
        ↪  'Zeros',activation='softmax' ))
        model.compile( optimizer= Adam(learning_rate = 0.0001,
        ↪  epsilon = 0.1),  loss = 'mse',  metrics= ['Accuracy'])
        return model

    def predict_action(self, state):
        output = np.array([0,0,0,0,0])
        action = self.model.predict(state)
        action = np.argmax(action)
        output[action] = 1.0
        return output

    def choose_action(self,input):
        qs = self.model.predict(input)
        action = np.argmax(qs)
        while 1.0 == input[0][action]:
            qs[0][action] = -1
            action = np.argmax(qs)
        output = np.array([0,0,0,0,0])
        output[action] = 1.0
        return output

    def q_training(self, data):
        X = []
        y = []
        states, actions, new_states  = data
        rewards = np.delete(new_states,[0,1,2,3,4,6],1)
        rewards = rewards.flatten()
        current_qs_list = self.model.predict(np.array(states))
        future_qs_list =
        ↪  self.target_model.predict(np.array(new_states))
        for index, state in enumerate(states):
            action = np.argmax(actions[index])
            reward = rewards[index]
            scaled_reward = reward / 200 +0.5
            max_future_q = np.max(future_qs_list[index])
            new_q = scaled_reward + DISCOUNT * max_future_q
```

```
            # Update Q value for given state
            current_qs = current_qs_list[index]
            current_qs[action] = new_q
            current_qs = np.array(current_qs)/sum(current_qs)
            # And append to our training data
            X.append(state)
            y.append(current_qs)
        X = np.array(X)
        y = np.array(y)
        history = self.model.fit(X, y, verbose = 1, shuffle =
        ↪  True,batch_size = TRIALS_PER_BATCH, epochs=EPOCHS  )#
        return history

class RandomAgent:
    def hardwired_stategy(self, current_full_state, shots_fired):
        output = np.array([0,0,0,0,0])
        if shots_fired == 0:
            output[2] = 1.0
            return output
        if shots_fired == 1:
            output[3] = 1.0
            return output
        if shots_fired == 2:
            if current_full_state[0][5] == 0:
                output[1] = 1.0
                return output
            else:
                output[4] = 1.0
                return output

    def fully_random(self, current_state):
        output = np.array([0,0,0,0,0])
        rand_action = np.random.randint(FIELD_SIZE**DIMENSION)
        output[rand_action] = 1.0
        return output

    def pseudo_random(self, current_state):
        output = np.array([0,0,0,0,0])
        rand_action = np.random.randint(FIELD_SIZE**DIMENSION)
        while current_state[0][rand_action] == 1:
            rand_action = np.random.randint(FIELD_SIZE**DIMENSION)

        output[rand_action] = 1.0
        return output

    def choose_action(self, current_state):
        if all(current_state == np.array([0,0,0,0,0])):
            rand_action = np.random.randint(3) + 1
            output = np.array([0,0,0,0,0])
```

```
            output[rand_action] = 1.0
            return output
        rand_action = np.random.randint(FIELD_SIZE**DIMENSION)
        while 1.0 == current_state[rand_action]:
            rand_action = np.random.randint(FIELD_SIZE**DIMENSION)
        output = np.array([0,0,0,0,0])
        output[rand_action] = 1.0
        return output

def data_refiner(data, episode):
    states = data.astype(int)
    only_states = np.delete(states, 5, 1)
    only_states = np.delete(only_states, 5, 1)
    new_states = states
    X = states
    y = []
    for index in range(int(len(only_states)/4)):
        new_states = np.delete(new_states, 3*index, 0)
        X = np.delete(X, 3*index + 3, 0)
        ac1 = only_states[4*index+1]
        ac2 = only_states[4*index+2] - ac1
        ac3 = only_states[4*index+3] - ac1 - ac2
        y.append([ac1,ac2,ac3])
    y = np.array(y)
    y = y.reshape(-1,5)
    os.chdir(r'C:\...') # fill in folder where you want to saved
    ↪  the supervised learning data.
    np.savetxt(f'X_{episode}.csv', X, delimiter = ",")
    np.savetxt(f'y_{episode}.csv', y, delimiter = ",")
    np.savetxt(f'new_state_{episode}.csv', new_states, delimiter =
    ↪  ",")
    print('saved to file')

def average_tester(env,agent):
    total_reward = 0
    for i in range(AMOUNT_TRIALS_TESTED):
        env.reset()
        for i in range(RESET_ENV_EVERY):
            output = np.array([0,0,0,0,0])
            action = agent.model.predict(env.full_state)
            action = np.argmax(action)
            output[action] = 1.0
            new_action = output
            env.update_env(new_action)
        total_reward += env.episode_reward
    print(f'the average score
    ↪  is:{total_reward/AMOUNT_TRIALS_TESTED} ')
    return (total_reward/AMOUNT_TRIALS_TESTED)

```

```
# Generate training data
agent = RandomAgent()
env = Environment()
TOTAL_TRAINING_POINTS = 1000000
SAVE_TRAINING_POINTS_EVERY = 100000
for i in range(TOTAL_TRAINING_POINTS/SAVE_TRAINING_POINTS_EVERY):
    for episode in range(SAVE_TRAINING_POINTS_EVERY):
        env.reset()
        shots_fired = 0
        for i in range(RESET_ENV_EVERY):
            new_action = agent.pseudo_random(env.full_state)
            ↪  #choose between pseudo_random, fully_random, or
            ↪  hardwired_strategy
            shots_fired += 1
            env.update_env(new_action)
    data = env.extract_memory()
    env.erase_memory()
    data = data.reshape(-1, 7)
    data_refiner(data,env.episode_number)

# Supervised Learning
agent = Agent()
env = Environment()
training_files = 1
training_count = 100000
X= []
y= []
for i in range(1,training_files+1):
    os.chdir(r'C:\...') # fill in folder where you saved the
    ↪  supervised learning data inputs (X).
    a = np.loadtxt(f'X_{i*training_count}.csv', delimiter=",")
    X.extend(a)
    os.chdir(r'C:\...') # fill in folder where you saved the
    ↪  supervised learning data labels (y).
    b = np.loadtxt(f'y_{i*training_count}.csv', delimiter=",")
    y.extend(b)
X = np.array(X)
y = np.array(y)
number = int(len(X)/1000)
X = X[:1000*number]
y = y[:1000*number]
X = X.reshape(-1,7)
y = y.reshape(-1,5)
history = agent.supervised_training(X,y)
average_tester(env,agent)

# Transer weights to q-learning agent
DeepQagent = QAgent()
env = Environment()
```

```
weights = agent.model.get_weights()
DeepQagent.model.set_weights(weights)
DeepQagent.target_model.set_weights(weights)

# Load data for reinforcement learning
training_files = 7
training_count = 100000
A,B,C = [],[],[]
for i in range(1,training_files+1):
    os.chdir(r'C:\...') # fill in folder where you saved the
    ↪  training data.
    a = np.loadtxt(f'X_{i*training_count}.csv', delimiter=",")
    b = np.loadtxt(f'y_{i*training_count}.csv', delimiter=",")
    c = np.loadtxt(f'new_state_{i*training_count}.csv',
    ↪  delimiter=",")
    A.extend(a)
    B.extend(b)
    C.extend(c)
number = int(len(A)/1000)
A = A[:1000*number]
B = B[:1000*number]
C = C[:1000*number]

# Reinforcement learning
history = DeepQagent.q_training([A,B,C])
average_tester(DeepQagent)
```