Proceedings of the CTD 2022
PROC-CTD2022-37
October 31, 2022

Reinforcement learning for charged-particle tracking

Liv Helen Våge

Department of Physics
Imperial College London, UK

ABSTRACT

The LHC is undergoing upgrades that will increase the nominal luminosity by at least a factor of five. This creates a computational challenge for the experiments' real time filter systems, particularly for their particle tracking algorithms. Reinforcement learning is introduced as a potential way to address this. It is shown that several reinforcement learning algorithms are able to learn to approximate hit positions given the previous hit associated with a candidate track. The reinforcement learning does not yet perform well enough to be a standalone solution, but it is shown that it may aid other methods such as graph neural networks.

PRESENTED AT

Connecting the Dots Workshop (CTD 2022)
May 31 - June 2, 2022

1 Introduction

The Large Hadron Collider [1] at CERN has brought several scientific discoveries since its inception in 2010, including the discovery of the Higgs boson. It is now being upgraded, and will continue its search for new physics, with a particular emphasis on rare processes and Higgs boson properties. The rate of collisions will increase by at least a factor of five compared to the original design value, and the average number of interactions per bunch crossing will increase from 50 to 200 [2]. Having these interactions simultaneous to the interaction of interest (pileup) poses a great computational challenge, particularly for the real time filter systems. At the CMS [3] experiment, a two tiered trigger system reduce the amount of data read to storage by around two orders of magnitude in an order of 100ms [4]. Even with a planned latency increase, current algorithms will fall short of the upgrade requirements. Particle tracking takes up the largest part of the triggering time budget, and scales particularly badly for high pileup scenarios [5].

1.1 Particle tracking

As charged particles traverse the detector, they leave energy deposits in silicon detector layers. These hits are connected to reconstruct the particle track, known as tracking. The track gives information about the momentum and charge of the particle, the proton-proton interaction point, and potential particle decay points. All of this contributes to the decision on whether to write the event to storage. The high level trigger at CMS uses a Kalman filter [6] method to reconstruct tracks. Given a hit in the detector, a prediction is made of where the track will go next. If a hit is found near this point, the track is updated with this hit. If several compatible hits are found, the track splits, so the different possibilities are explored. This has proven a very fast and accurate method for reconstructing tracks. With the updates of CMS, this is not the case anymore. At the current pileup, there are O(10 000) hits in each event, which will scale to O(100 000) with 200 pileup. There are many more compatible hits in each iteration of the Kalman filter [6], causing a combinatoric explosion [5], and sending tracking far over the trigger time budget. This could be addressed by e.g. accelerating the Kalman filter method, or by machine learning.

1.2 Accelerating particle tracking

Many projects have been launched to accelerate particle tracking. Some of the main analytical methods that have been explored are mkFit and Patatrack. The mkFit [5] project has parerallised the traditional tracking by paralellising over each event, sections of the detector, and seed tracks. There are plans to port this to GPUs. The Patatrack [7] collaboration swapped the Kalman filter for a line fit method, parellalised it and ported it to GPUs. Several machine learning approaches have also been investigated, but graph neural networks have shown the highest track reconstruction accuracy [8]. Hits act as nodes in a graph, and nearby hits are connected with edges. One can then use a graph neural network to predict whether the edges are false or true edges. This has shown a high accuracy, though building the graphs is time consuming [9].

All these projects have shown promising acceleration, but have not met timing requirements yet. One challenge is that they still retain the combinatiorial nature of the problem. Reinforcement learning [10] could provide an approach that directly addresses the combinatorics. It offers a way to take advantage of previous experience and is easily parallelisable. If the accuracy is not adequate, it could still help build smaller graphs for the graph neural networks; potentially improving both latency and accuracy.

2 Reinforcement learning

Reinforcement learning (RL) relies on the reward hypothesis; any goal can be represented as a maximisation of the expected value of the cumulative sum of a reward signal. In other words, a problem is phrased as an agent acting in an environment that receives rewards based on its actions. This has had large successes in recent years, including AlphaGo, AlphaZero, and fusion plasma control [11, 12] . RL can be appropriate when a problem is intractable and the environment changes based on the action. Most often, RL also requires Markovian state; where the agent will go next depends only on the current state. The set of rules

that determine which actions an RL agent takes is called a policy. This can be deterministic, like a neural network, or stochastic, like sampling from a distribution. One can learn a policy by learning the parameters of e.g. a neural network or a Gaussian distribution. Most algorithms also have a value function that estimates the expected return given a state-action pair.

RL is divided into two main categories; model based and non-model based. Model based learning has a model of the environment, which is sample efficient, but relies on a good model of the environment. The second dividing element of RL methods is what they learn. One can learn a policy, a value function, or a model of the environment. When choosing a reinforcement learning method, it can be worth keeping in mind that non-model based methods have been explored more, and are often easier to implement [10]. Some of the methods that were used to explore RL for particle tracking are introduced below.

2.1 Policy gradient

Policy gradient algorithms [13] are some of the simplest RL algorithms and are often a logical starting point for a new reinforcement learning application. They aim to learn the parameters of a policy through gradient descent. For continuous action spaces, Gaussian policies are often used. Each agent state is associated with a Gaussian distribution, and actions will be sampled from this distribution. As the agent learns, the mean might shift, and the distribution will narrow as the policy learns the best action for that state. Policy gradients are considered to have good convergence properties, which is one reason they are often used as a first test for new reinforcement learning applications. A downside is that they often converge to local instead of global optima. Policy gradients are suitable for both continuous and discrete action spaces.

2.2 Deep Q-Network

Another relatively simple algorithm is deep Q-learning [14]. Q-learning tries to learn the quality of an action given a certain state. The simplest version of Q-learning simply tabulates the quality of all possible actions given all possible states. In many problems, the dimensionalty is too big for all combinations to be explored. Instead, one can learn a function that calculates the quality of the possible actions given a certain state. This is most often done with deep neural networks, called deep Q-Networks (DQN) [15]. The neural network predicts the quality of each possible action given an input state. To avoid situations where a lot of similar states dominate the learning, previous experiences are stored in a buffer and replayed through the network. Methods that use neural networks to estimate qualities of actions usually always use another neural network called a target network. The Q-learning neural network uses the reward signal to calculate the estimated quality of an action, and performs supervised learning with this as a target. This quality can change depending on the samples, so the target changes. To avoid this, the weights of this network is copied to a target network. This only updates every N iterations to keep the learning stable. The action space for DQN must be discrete since one chooses actions from a discrete set of output nodes.

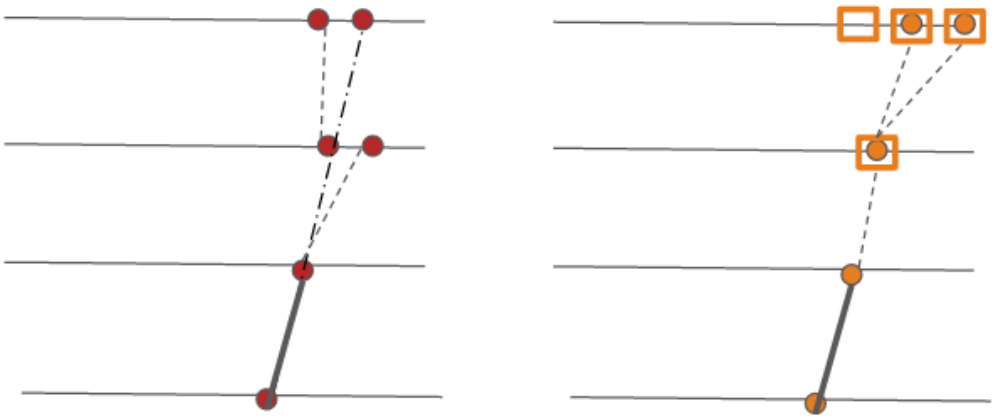
2.3 Deep deterministic policy gradient

Deep deterministic policy gradient (DDPG) [16] has been developed specifically to act as a RL method similar to DQN that can be used in continuous action spaces. It combines ideas from the two previous methods, and learns both a policy and a Q-function. This idea is called actor-critic since there is a policy (actor) and a critic (Q-function) that critisises the moves made by the policy. The policy can predict continuous states, and the critic will estimate the quality of the action. The policy will be guided by the critic as it updates. This can create a good, advanced model for continuous spaces, but has been known not to converge, even in simple environments [17].

3 Reinforcement learning for particle tracking

The tracking problem was recast as a reinforcement learning problem where an agent is given a state and predicts where the next hit will be. The state contains the current hit position, and to be Markovian, it also

needs track parameter estimates or information about the previous hits for the track candidate. The simplest Markovian state can for instance contain the current and previous hit positions. The two inner hits of each track were used as a seed to fill the track parameter or previous hit information. An illustration in shown in Fig. 1. Although this uses truth-level information, it does not make the scenario entirely unrealistic, since seeds are created at the low level trigger [4]. The hit positions were described by two dimensional coordinates; z and r, where r is $\sqrt{x^2 + y^2}$. The tracks are straight in this space when ignoring energy loss, making it much easier to learn than a three dimensional helix. The reward was defined by the negative distance between the predicted and true hit position. Each episode contained five hit predictions, so only tracks with a total of seven hits were used. To simplify the problem, only one hit per layer were considered. The TrackML dataset [18] was used. This is a dataset created to benchmark machine learning approaches for tracking. It uses a generic detector layout and an average pileup of 200, making it a reasonable reflection of the upgrade environment.



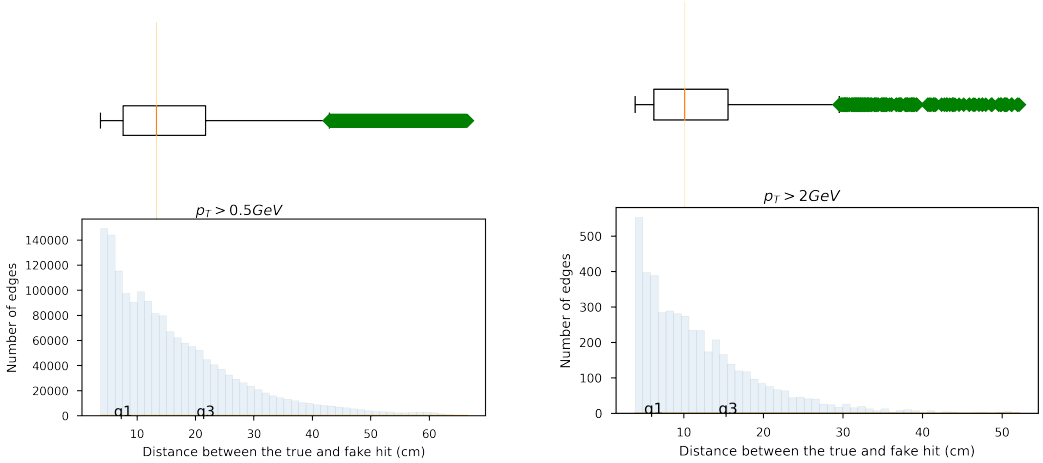
(a) (b)

Figure 1: Visualisation of the tracking problem. A seed is created from the two inner hits. The next hit position is predicted using help from the previous hit, a seeded linear fit (a) (Section 3.1.2) or a module mapping (b) (Section 3.1.3). The reward is given after each hit prediction based on the distance between the actual hit and the predicted hit.

Ideally, this approach could be used standalone for tracking. If it does not achieve sufficient performance, it could instead be used to cut down the number of fake edges built in a graph neural network. To know how accurate the RL algorithm needs to be to do this, the distance between true hits and hits for built fake edges in a graph neural network was studied. This was done using the graph neural network implementation given in [9]. The results are shown in Fig. 2. It shows that if one has a RL agent that consistently predicts hit positions 10cm within the true hit, one would cut around half of the fake edges built for the graph. Within 20cm one would eliminate 25% of the fake edges. The resolution of the detector is on the order of micrometres [3], so a method that predicts within 10-20cm of the true hits would be very far away from performing tracking well on its own. Fig. 2 shows that it could still be used to build smaller and more accurate graphs for graph neural networks. It does not outperform a straight line fit, as is discussed in Section 3.1.2.

3.1 Policy gradient for tracking

A simple policy gradient algorithm was applied to the environment outlined above. A neural network using just two hidden layers with 64 neurons each was used. Since the aim is to accelerate tracking, it is important



(a) (b)

Figure 2: Distance between the true track hit and fake edge hit for two minimum p_T points. The orange line illustrates the median and the box shows the interquartile range. The green points mark points that are outliers, i.e. a value higher than 1.5 times the interquartile range plus the third quartile.

to keep the neural network lightweight, so it does take longer than traditional tracking. A lightweight approach also makes any potential GPU or FPGA acceleration easier. A Gaussian policy parameterised by a neural network was chosen. Many variations of state descriptions were explored, with some highlighted in the section below.

3.1.1 Inductive bias

Every machine learning model has inductive bias - something that makes it favour one model over another. This becomes especially important in reinforcement learning. If a human were to connect the dots in Fig. 1, it is likely they would immediately assume that hits would only be possible on detector layers, and given the physics of the problem, there would be a straight line from the seed. A RL agent does not have this intuition. Infusing the model with this inductive bias is therefore important to avoid extremely long training times or a lack of convergence.

Fig. 3 shows three different RL scenarios. If one predicts the position of the next hit, the algorithm does not appear to learn in a reasonable time frame. There is too much action space to explore to find good solutions. Since we know the next hit in the track will be close to the previous, predicting a change in position eliminates a lot of the action space, giving a jump in performance. One can go even further and assume that the change in coordinates is going to be similar to that of the last hit, and learn a correction to this assumption. This has a slight advantage early on, but it converges on the same solution as predicting a change in position. The black dashed line shows the accuracy when always assuming the same change in position as the seed without any RL. This illustrates that the RL learns valuable information in addition to the inductive bias it is given. A small increase in performance not shown in the graph was also seen by taking advantage of the symmetry in the z-axis. Projecting all tracks to the positive z-plane removes half the phase space, so the agent learns more easily. Tracks that cross the r-axis and are in both positive and negative z-space would be displaced tracks and are not considered in this study. The average return is the total reward for a track averaged over the past forty iterations. This converges around -100. Since five hits were predicted, this means that the hit predicted is consistently 20cm within the actual position. Consulting Fig. 2, this would cut down about 25% of the fake edges of a graph neural network. While it is an improvement, it is likely not enough to compensate for the extra time taken for the RL evaluation. To improve, more prior knowledge was considered.

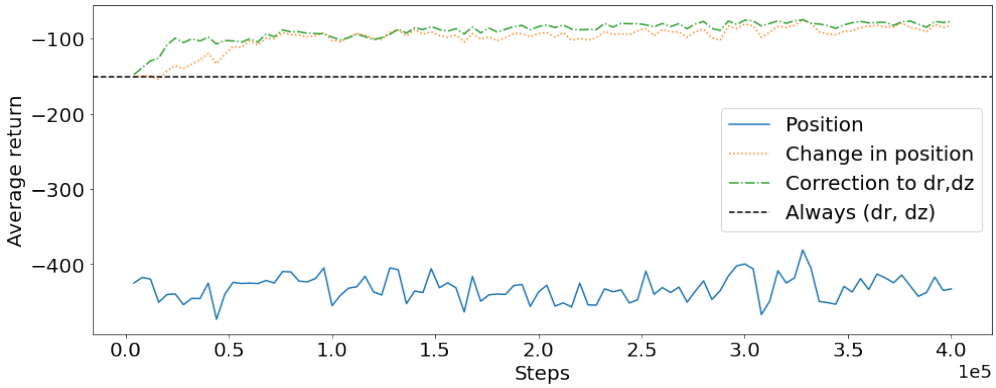


Figure 3: Reward for tracks averaged over the past forty training iterations using a policy gradient environment with different inductive biases. The tracks have two seed hits and five hit predictions. The various lines shows what the policy was trying to predict. Always (dr, dz) corresponds to a non-RL scenario where one assumes the change in r and z are the same as in the seed for the entire track.

3.1.2 Linear fit

A disadvantage of the approach in Section 3.1.1 is that it allows hit predictions outside the detector layers. To address this and take advantage of how tracks form a straight line in the r-z space, a line fit was applied. A straight line fit was created from the seed, and intersection points with the detector layers were calculated. To see if this would be a viable method, it was first tried without any reinforcement learning. Since a potential goal was to reduce the number of fake edges in a graph neural network, the graph building outlined in [9] was used as a baseline. Implementing it with the linear fit, only hits that fell within $\pm$ 1.5 cm of the detector intersection points were connected. The graph neural network was then trained and evaluated. The results are shown in Fig. 4.

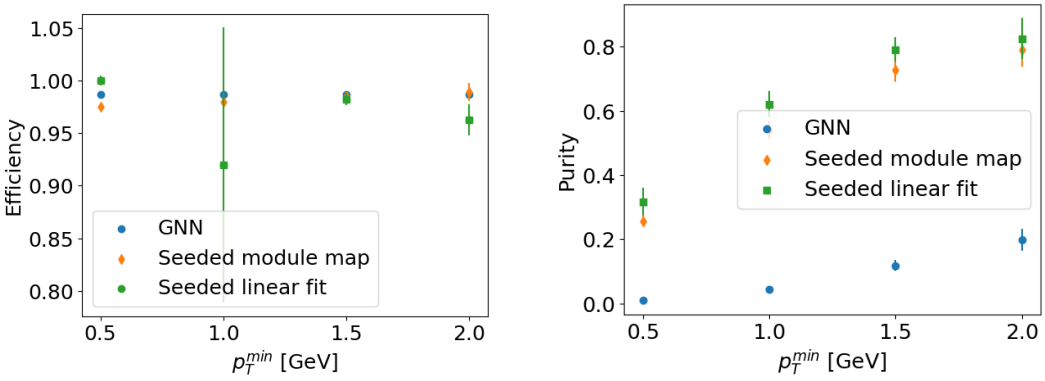
The linear fit shows a similar efficiency to the baseline graph neural network and a much higher purity. Since it showed promise, it was then combined with reinforcement learning. The baseline assumption was that the next hit would be where the next detector intersection is, and the reinforcement learning agent learnt a correction to this. This reduced the average episode return from -100 to around -60, so resulted in much more accurate hit predictions. Perhaps surprisingly, the reward when using just the linear approach and no machine learning was -30. This discrepancy is partially due to missing hits.

The corrections to the linear fit are generally quite small. When there are missing hits, all the predictions are offset by one, causing very low rewards for the remains of the episode. This hides what the reinforcement learning agent needs to learn. When only considering the inner tracker, and removing any tracks with missing hits, the average episode reward is around -1.5 for the linear fit, and -5 for the linear fit reinforcement learning. This would remove more than 75% of the fake edges in the graph building, and approaches the limit where RL might be considered as a standalone method. The reason the reward is lower when applying RL is that the corrections to the linear assumption are likely to be close to stochastic. Any pattern probably does not emerge until one also considers where the observed detector hits are.

Ideally, the reinforcement learning agent would know roughly where the next hit is and sample its next state from the compatible detector hits. Learning how to choose the best hit given a set of available hits is difficult because every event will look different. A simple way to account for hit information is to simply choose the hit that is closest to the hit prediction. Without using reinforcement learning, that brings the reward up to around -0.7 for the linear fit without machine learning. With RL, the reward is still around -5, since the agent is not aware of the hits before making its prediction. The linear fit approach therefore looks good as an analytical solution, but to benefit from reinforcement learning, one would need the RL agent to know the position of available detector hits before making its prediction. This could be done by for instance behaviour priors [19] or action masking [20].

3.1.3 Module mapping

The linear fit has the disadvantage of doing analytical calculations for every track. An alternative could be module mapping, which has been extensively explored as a way to reduce the number of edges in graph neural networks for particle tracking [21]. By iterating over tracks and storing which modules are adjacent, one can create a mapping of allowed module connections. This has shown very good results for reducing the number of fake edges, but could perhaps be put to even better use when combined with reinforcement learning. Other module mapping methods usually consider a sequence of two or three modules that are connected. These are stored in a mapping and any of the connections are allowed in the graph building. Inspired by the linear fitting in the approach above, one might consider that each module connection is likely to only occur for a certain line slope. By iterating over 10 000 tracks, a mapping was created with this in mind. The end result is a dictionary where one can look up a module, the current line parameters, and find all the connected modules that have been seen. Similarly to the linear fit, it was first tried with the graph building outlined in [9]. Only hits that conformed with the module mapping were connected. The results are shown in Fig. 4.



(a) (b)

Figure 4: Efficiency (a) and purity (b) comparisons for graph neural networks using two analytical methods and a baseline. The efficiency is the ratio of edges that were correctly classified as true. The purity is the ratio of real to total edges in the graph. The linear fit and module map methods are described in Section 3.1.2 and Section 3.1.3 and show similar efficiency and consistently higher purity than the baseline graph neural network method.

The module mapping results are quite similar to the linear approach, with a slightly lower purity. The module mapping has not been combined with reinforcement learning yet. This is because it is a similar situation to the one outlined in Section 3.1.2. One could e.g. use the module mapping to learn a correction to assuming that the track would go to the average allowed module position. To benefit from RL, however, one would need to learn which hit to choose given a set of available hits.

3.2 DDPG and Q-learning

Since the policy gradient is a very simple reinforcement learning method, other approaches were also explored. DDPG is a more advanced method well suited for continuous spaces. It did not improve the accuracy compared to the policy gradient method. Some environments achieved similar performance, but many had lower performance and a much more unstable learning process. As described in [17], DDPG uses several neural networks, so these behaviours are known to occur. Learning in continuous environments can also be difficult, so a discrete approach was tried. Deep-Q learning was implemented to predict which module the track would propagate to given previous modules. This had very bad performance and did not seem to learn any valuable information. This is likely due to the somewhat arbitrary numbering of modules, making it

difficult to create a meaningful mathematical relation between them. There are many ways to deal with this in Deep-Q learning, but many of them require quite complex representation of the state. There might be some value to these discrete approaches, but the combinatorics of such problems make them a less attractive alternative.

4 Conclusion

Reinforcement learning for particle tracking has been explored. Using a simple policy gradient method, it was shown that an RL agent could learn to predict where the next hit would be with an accuracy of about $\pm$ 20cm. This is quite a wide range, and would not be sufficient for RL to be a standalone method for tracking. If used in conjunction with graph neural networks, it could cut down about 25% of the fake edges built, but would likely not make up for the additional evaluation time and complexity. Linear fitting and module mapping were explored for use in reinforcement learning. Both showed promise, with the linear fit predicting consistently within 5cm of the true hit for a slightly simplified scenario. The linear fit performed better without any reinforcement learning than with it. To benefit further from reinforcement learning, the agent needs to know about the available detector hits before making a prediction. The linear fit without any machine learning was consistently accurate within less than a centimetre in a slightly simplified scenario, so it can be used as a benchmark for accuracy and latency for future work.

ACKNOWLEDGEMENTS

I am very grateful to Prof. Alex Tapper and Dr. Lucas Borgna for their valuable contributions to this work.

References

[1] Lyndon Evans. "The Large Hadron Collider". In: *New Journal of Physics* 9 (9 Sept. 2007), pp. 335–335. ISSN: 1367-2630. DOI: 10.1088/1367-2630/9/9/335.

[2] Burkhard Schmidt. "The High-Luminosity upgrade of the LHC: Physics and Technology Challenges for the Accelerator and the Experiments". In: *Journal of Physics: Conference Series* 706 (Apr. 2016), p. 022002. ISSN: 1742-6588. DOI: 10.1088/1742-6596/706/2/022002.

[3] The CMS Collaboration. "The CMS experiment at the CERN LHC". In: *Journal of Instrumentation* 3 (08 Aug. 2008), S08004–S08004. ISSN: 1748-0221. DOI: 10.1088/1748-0221/3/08/S08004.

[4] The CMS Collaboration. "The CMS trigger system". In: (Sept. 2016). DOI: 10.1088/1748-0221/12/01/P01020.

[5] Giuseppe Cerati et al. "Speeding up Particle Track Reconstruction in the CMS Detector using a Vectorized and Parallelized Kalman Filter Algorithm". In: (June 2019). DOI: 10.48550/arxiv.1906.11744.

[6] R. E. Kalman. "A New Approach to Linear Filtering and Prediction Problems". In: *Journal of Basic Engineering* 82 (1 Mar. 1960), pp. 35–45. ISSN: 0021-9223. DOI: 10.1115/1.3662552.

[7] A. Bocci et al. "Heterogeneous Reconstruction of Tracks and Primary Vertices With the CMS Pixel Tracker". In: *Frontiers in Big Data* 3 (2020). ISSN: 2624-909X. DOI: 10.3389/fdata.2020.601728.

[8] Steven Farrell et al. *Novel deep learning methods for track reconstruction*. 2018. DOI: https://doi.org/10.48550/arXiv.1810.06111.

[9] Gage DeZoort et al. "Charged particle tracking via edge-classifying interaction networks". In: *Computing and Software for Big Science* 5.1 (2021), pp. 1–13. DOI: https://doi.org/10.1007/s41781-021-00073-z.

[10] Richard S Sutton and Andrew G Barto. *Reinforcement learning: An introduction*. MIT press, 2018.

[11] David Silver et al. "A general reinforcement learning algorithm that masters chess, shogi, and Go through self-play". In: *Science* 362 (6419 Dec. 2018), pp. 1140–1144. ISSN: 10959203. DOI: 10.1126/SCIENCE.AAR6404/SUPPL_FILE/AAR6404_DATAS1.ZIP.

[12] Jonas Degrave et al. "Magnetic control of tokamak plasmas through deep reinforcement learning". In: *414 | Nature |* 602 (2022). DOI: 10.1038/s41586-021-04301-9.

[13] Richard S Sutton et al. "Policy gradient methods for reinforcement learning with function approximation". In: *Advances in neural information processing systems* 12 (1999). URL: https://proceedings.neurips.cc/paper/1999/file/464d828b85b0bed98e80ade0a5c43b0f-Paper.pdfAdvances%20in%20neural%20information%20processing%20systems.

[14] Christopher J C H Watkins and Peter Dayan. "Q-Learning". In: *Machine Learning* 8 (1992), pp. 279–292. DOI: https://doi.org/10.1007/BF00992698.

[15] Volodymyr Mnih et al. "Human-level control through deep reinforcement learning". In: *nature* 518.7540 (2015), pp. 529–533. DOI: https://doi.org/10.1038/nature14236.

[16] David Silver et al. "Deterministic Policy Gradient Algorithms". In: *Proceedings of the 31st International Conference on Machine Learning*. Ed. by Eric P. Xing and Tony Jebara. Vol. 32. Proceedings of Machine Learning Research 1. Bejing, China: PMLR, 22–24 Jun 2014, pp. 387–395. URL: https://proceedings.mlr.press/v32/silver14.html.

[17] Guillaume Matheron, Nicolas Perrin, and Olivier Sigaud. "The problem with DDPG: understanding failures in deterministic environments with sparse rewards". In: *arXiv preprint* (Nov. 2019). DOI: 10.1007/978-3-030-61616-8_25.

[18] David Rousseau et al. "The TrackML challenge". In: *NIPS 2018-32nd Annual Conference on Neural Information Processing Systems*. 2018, pp. 1–23. URL: https://hal.inria.fr/hal-01745714/document.

[19] Dhruva Tirumala et al. "Behavior Priors for Efficient Reinforcement Learning". In: *arXiv preprint* (Oct. 2020). URL: http://arxiv.org/abs/2010.14274.

[20] Anssi Kanervisto, Christian Scheller, and Ville Hautamäki. "Action space shaping in deep reinforcement learning". In: *2020 IEEE Conference on Games (CoG)*. IEEE. 2020, pp. 479–486. DOI: 10.1109/CoG47356.2020.9231687.

[21] Alexis Vallier. *Graph Neural Networks for track reconstruction @ HL-LHC*. Learning To Discover. Apr. 2022.