Article title: .

Authors: Not Named[1]

Affiliations: 102 Christopher Columbus Dr, Jersey City, NJ 07302[1]

Orcid ids: 0000-0002-2464-7138[1]

Contact e-mail: 346815@bsd48.org

License information: This work has been published open access under Creative Commons Attribution License <http://creativecommons.org/licenses/by/4.0/>, which permits unrestricted use, distribution, and reproduction in any medium, provided the original work is properly cited. Conditions, terms of use and publishing policy can be found at <https://www.scienceopen.com/>.

Preprint statement: This article is a preprint and has not been peer-reviewed, under consideration and submitted to ScienceOpen Preprints for open peer review.

Funder: None

DOI: 10.14293/S2199-1006.1.SOR-.PPD7FVS.v1

Preprint first posted online: 27 October 2020

Keywords: Algorithms

Applying a Deep Q Network for OpenAI's Car Racing Game

Ali Fakhry: September 12, 2020

Horizon Labs

Abstract. The applications of Deep Q-Networks are seen throughout the field of reinforcement learning, a large subset of machine learning. Using a classic environment from OpenAI, CarRacing-v0, a 2D car racing environment, alongside a custom based modification of the environment, a DQN, Deep Q-Network, was created to solve both the classic and custom environments. The environments are tested using custom made CNN architectures and applying transfer learning from Resnet18. While DQNs were state of the art years ago, using it for CarRacing-v0 appears somewhat unappealing and not as effective as other reinforcement learning techniques. Overall, while the model did train and the agent learned various parts of the environment, attempting to reach the reward threshold for the environment with this reinforcement learning technique seems problematic and difficult as other techniques would be more useful.

Background

Machine learning resides as a subset of artificial intelligence (AI) that is formulated in a way to automatically learn and improve from experience without needing to be programmed manually and completely [13].

There are a variety of different machine learning strategies to use for different tasks; these strategies range in difficulty, from more basic forms of algorithms to more complex ones. Using the optimal strategy to produce the best results is a key strategy revolving upon the field of machine learning. Generally, machine learning requires large quantities of data, which have to be analyzed and evaluated. While machine learning typically delivers more accurate results in order to identify profitable opportunities or possible risks, it requires many resources and training time to produce these answers [13].

Machine learning has become increasingly popular in many industries and fields, such as biotechnology and business. The algorithms used in machine learning produce great results for a low cost, as the only cost necessary is training the data to produce a solid model. Whether it is for recognizing images at a large scale, predicting future trends, or using the computational power available to solve tough problems humans can not solve, machine learning is revolutionizing the way many businesses operate.

1. Reinforcement Learning

Reinforcement learning is when an algorithm gives feedback about specific actions, where there is a desired end state, yet typically there is no clear best path. In reinforcement learning, a model is asked to make a series of decisions. Each decision that is acted upon will provide a reward, which will either encourage the model to keep performing said actions or the opposite. Reinforcement machine learning algorithms are a form of learning methods that are applied by interacting with the environment, and the agent, what the model is targetting, will attempt to maximize its reward through trial and error.

2. Q-Learning

Q-Learning is one of the more basic reinforcement learning algorithms; that is due to its "model-free reinforcement learning" nature. A model-free algorithm, as opposed to a model based algorithm, has the agent learn policies directly. Like many of the other algorithms, Q-Learning has both positives and negatives [7]. As discussed upon, Q-Learning does not require a model, nor does it require a complicated system of operation. Instead, Q-Learning uses previously learned "states" which have been explored to consider future moves and stores this information in a "Q-Table." For every action taken from a state, the policy table, Q table, has to include a positive or negative reward. The model starts with a fixed epsilon value, which represents the randomization of movements [7]. Over

time, the randomization decreased based upon the epsilon decay value. Furthermore, if the current state of the agent is considered to be new or unexplored, the agent will just produce a randomly generated move, in an attempt to better learn the environment.

This form of learning is great when there are a limited number of moves or the environment is not complicated, as the agent remembers past moves and repeats them with ease.

However, for more complex environments with a significantly larger number of states, the Q-Table will fill up rapidly, causing large training times. The issue is, Q-Learning does not predict but rather deals in absolutes the majority of the time: the state has already been considered and the action to produce is already known, or the state is unknown, and a random action has to be applied upon.

The equation for this algorithm is depicted below and can be explained simply going from left to right. Note: the “ $\leftarrow$ ” represents what is generally denoted as an equal symbol. This formula is applied to determine the best action to take best on the current state.

$$Q^{new}(s_t, a_t) \leftarrow \underbrace{Q(s_t, a_t)}_{\text{old value}} + \underbrace{\alpha}_{\text{learning rate}} \cdot \left(\underbrace{r_t}_{\text{reward}} + \underbrace{\gamma}_{\text{discount factor}} \cdot \underbrace{\max_a Q(s_{t+1}, a)}_{\text{estimate of optimal future value}} - \underbrace{Q(s_t, a_t)}_{\text{old value}} \right)$$

temporal difference
new value (temporal difference target)

Equation 1: Q-Learning from [6].

The “Q” value represents the quality of a value, or how well the action is perceived in the algorithm. The higher the quality value is, the more likely the same action will be performed again. The quality of the action is represented with $Q^{new}(s_t, a_t)$, where the s_t represents state and a_t represents action. The model will discount new values using gamma and adjust the action process, step, based around the learning rate [3].

3. Neural Networks

Neural networks are considered to be “function approximators,” which are useful for various applications. The functions that can be approximated ranges from linear functions to more complex equations, unlike other basic models of machine learning. Due to Neural Networks being function approximators, they are great for tasks such as computer vision and image classification and can be applied to all sorts of machine learning tasks. Because of this, the input for convolutional neural networks, CNNs, tend to be images turned into pixels, at classic red-green-blue pixels or they might be gray-scaled which can sometimes be used to boost performance. After the input layer, there are many layers before the final output layer; these layers are called, “hidden layers,” which are represented with a number of parameters, tuned to fit a certain task ahead. In CNNs, a common type of neural networks, the mathematical operations applied to the hidden layers are called convolutions. A convolution is considered to be an example of a linear operation, which without modification would only be able to approximate linear functions as a result, this is where activation functions come in, many are simple - such as ReLU activations - and others are more complicated. ReLUs are commonly used due to their simplicity and ability to convert linear approximations easily into non-linear conversions. At the end of the feature extractions, the result is a number of features that have been shown to describe the qualities of the data; these features are then mapped as possible outputs.

Over the course of the last 20 years, the computational power accessible has increased, allowing for larger neural networks to be constructed; by the sizing of the neural network, what is being referenced is the amount of layers. Thus, more neural networks tend to be applied upon as deep neural networks, DNN, as they would have multiple layers between the input and the output layers [9]. This allows for better approximations, thereby creating a better model.

4. Deep Q-Networks

Essentially, Deep Q-Networks (DQNs) are using Neural Networks in a combination with Q-Learning to produce accurate predictions on future decisions without the need for Q-Tables [8]. The Q table is a lookup table, where every entry is a combination of states and actions, which as explained prior, are great for more basic environments. However, for large environments, the Q table will grow substantially and the computational power will not have the memory or time to search environments. For example, if there were 10

different variables, each having 10 possible values that are able to contribute to the state, while having 2 possible actions, the Q table would have 20 billion entries. That is too much.

A downside to using DQNs, as with any neural networks is that training them is a slow process. For more complex environments, using DQNs would be slower than using classic Q-Learning algorithms. The equation for DQNs is the Bellman equation.

$$Q(s, a) = r(s, a) + \gamma \max_a Q(s', a)$$

Equation 2: DQN Equation from [3].

The equation has similar attributes to that of Q-Learning, such as similar notations. Yet, the way the formula is arranged is different. The equation yields that the Q Value, quality, is from the state s' and that of the selected action a with the immediate reward received, plus the highest possible Q-value from all the states. The algorithm will produce the highest Q Value from the s' by deciding the action that maximizes the Q Value [8]. Yet, the discount factor, similar to that of the Q-Learning algorithm still applies, which relates to future reward compared to the current.

Structures

1. Pytorch Framework

Pytorch is a great introductory deep learning research platform, with easy access to GPUs and is compatible with NumPy. Pytorch, no matter the use of it, is flexible, easy to learn, and fast [14].

Through the many parts of Pytorch, there are some key features. The first part of Pytorch is the use of tensors, which are very similar to what is considered to be NumPy's "ndarrays," yet they have the addition of being used through a GPU; thereby, using this hastens the speed of training. Furthermore, there are many operations that can be applied upon, varying with the uniqueness of the syntax. One main difference that is seen between TensorFlow, a direct competitor of Pytorch, and Pytorch is the fact that TensorFlow uses static computational graphs while PyTorch relies more on dynamic computational graphs. Using Dynamic Computational Graphs is rather rare in comparison to other frameworks; Chainer and Dynet are among the few frameworks that use Dynamic Computational Graphs, as opposed to other methods. This capability is referenced as a "Define By Run" rather than a more classic "Define and Run."

2. Pretrained Models

Pretrained models are models created in the past to solve other, but similar problems. These models' architecture is given for free within the model and does not require much extra training. The packages, particularly for Pytorch, contain definitions of models for addressing different tasks. These include image classification, pixelwise semantic segmentation, instance segmentation, object detection, person keypoint detection, and video classification [12].

The use of pretrained models - by definition - is considered to be transfer learning. While the majority of the layers are already trained, the final layers must be manipulated and reshaped in order to have the same input as the output of the pretrained models. Furthermore, while training, the user has to optimize the pretrained model by choosing which layers do not get retrained, as it makes the use of transfer learning somewhat useless if all the layers are retrained [10]. Thus, it is typical that the entirety of the pretrained model is frozen.

Environments

1. Classic Environment

The classic CarRacing-v0 environment is both simple and straightforward. Without any external modifications, the state consists of 96x96 pixels, starting off with a classical RGB environment as well. The reward is equal to -0.1 for each frame and $+1000/N$ for every track tile visited, where N is represented by the total number of tiles throughout the entirety of the track. To be considered a successful run, the agent must achieve a reward of 900 consistently, thus meaning that the maximum time the agent has to be on the track is 1000 frames. Furthermore, there is a barrier outside of the track, which results in a -100 penalty and an immediate finish of the episode if crossed over. Outside the track consists of grass, which does not give rewards but due to the friction of the environment, results in a struggle for the vehicle to move back onto the track. Overall, this environment is a classic 2D environment which is significantly simpler than that of 3D environments, making OpenAI's CarRacing-v0 much simpler.

Figure 1: A screenshot of the classic CarRacing-v0 environment.

2. Custom Environment

The borders of the classic environment force the agent inside the restrictions of the border. Thus, a theory was created that replacing the grass with the borders will force the vehicle within the track, allowing for quicker learning time. Removing the grass leaves only two possible locations, track or border, yet being on the border would end the episode immediately, meaning that all the states had to be on the track, compared to the classic environment where the vehicle would spend extensive time learning the mechanics of the grass and the friction applied upon. Going over the more prominent barrier still gives the agent a -100 reward and still causes the agent to be considered “done” for that episode.

Figure 2: A screenshot of the custom CarRacing-v0 environment.

CNN Methods

While there are differences between using a custom architecture and pretrained models, both shared some similarities. The learning rate for both was $\alpha = 0.001$ and a discount value equal to $\gamma = 0.999$.

1. Custom Architecture

Convolutional layers in a neural network are great for image recognition. Due to this environment's approach on image recognition, it seemed that conv2d layers were the best fit. Using PyTorch's Conv2d, a network was designed.

1. Conv2d(1, 6) (Kernel Size 4, Stride 4)
2. ReLU Activation (True)
3. Conv2d(6, 24) (Kernel Size 4, Stride 1)
4. ReLU Activation (True)
5. MaxPool2d (Kernel_Size 2)
6. Flatten Layer

While the image initially was recognized in a RGB state, each frame was directly translated upon using a grayscale filter. Thus, the input channel had to start at 1. The linear layers had to be implemented as well, following the Conv2d layers.

1. Linear (((9 x 9)x 24), 1000)
2. ReLU Activation (True)
3. Linear (1000, 256)
4. ReLU Activation (True)
5. Linear (256, 4)

Due to the nature of the input layer of linear layers, the calculation has to reflect the Conv2d layers. The image that would be sent through the linear activation layer would be 84 x 84 post-cropping. The first conv2d layer has a convolution of 4 x 4 without any padding with a stride of 4, dropping the size to 21 x 21. The second layer would then have an input of 21 x 21, and would apply a 4 x 4 convolution again, this time with a stride of 1, dropping the size to 18 x 18. Lastly, there is a MaxPool2d layer, with a kernel size of

2, thus making the overall size equal to 9×9 . This number, multiplied by 24, the output of the conv2d layer, produces the input of the linear activations.

Figure 3: A diagram of the custom built CNN architecture.

Note: the diagram above is intended to represent the custom CNN. However, ReLU and Flatten activations have not been included, as they do not alter the linear input.

2. Resnet18 Pretrained Model

Since release, Resnet has become one of the most common pretrained models in the space of transfer learning due to its accurate results and representations, especially for computer vision based tasks. Resnet18, from the family of Resnet pretrained models, is the variant with 18 layers, outputting 512 channels. Using untrained CNN's, typically slows down the training process significantly. The model can be simplified as the following:

1. Resnet18 (18 layers) (Frozen)
2. Linear (512, 256)
3. ReLU (True)
4. Linear (256, 4)

The diagram below shows the model explained. The input layer of 512 is received from the Resnet18 pretrained model, the added hidden layer has a size of 256, and the final output layer is 4 in length.

3. Customization

The approach to many DQN's is the same, especially for the network. However, the approach to the agent and the state varies. Many strategies employed effectively improved the training time, but the impact ranged by the method. Some approaches have been explained before and some have not.

- Cropping: 84 x 84 frame shape

There are many parts of the image that are needed. However, the bottom part is not needed. In fact, it can mess up the image recognition process, as the color black especially could be recognized as a border of the map. Thus, cropping the bottom part is necessary to optimize the model. Keeping the image in a fixed size also helps the model stabilize. Thereby, the edges were also cropped out of the PIL image, from 6 pixels from the left and 6 pixels to the right, which would not have a large effect on the training process.

- Gray Scaling: 3 channels to 1 channel

Using color - RGB - for computer vision tasks typically complicates the model and also introduces more channels necessary. Using 1 channel rather than 3, thus makes the model approximately 3 times faster gray scaling in comparison to normal RGB

images. Many times, using color adds no benefit, especially for a model as simple as CarRacing-v0, where the image recognition part is not as heavy as the actual learning side, especially due to the reason that the environment is not 3D, but rather maintains a 2D approach.

- Image Equalize: equalize image

For PIL images, there is a function that applies a non-linear mapping to an input image, in order to create a uniform distribution of grayscale values in the output image. Used to equalize the "image histogram." This method is used to increase contrast in the images, and due to the use of grayscale images, this is an important step.

- Epsilon Fluctuation: adjusting epsilon

Epsilon is needed for training, as it allows the model the necessary exploration needed. However, many times it appears that the epsilon applied upon is not enough. Thus, more epsilon and randomization are necessary. To rectify this, adjusting the epsilon automatically rather than manually appears to be the best course of action. The process was simple; if the last 50 episodes had better improvement than the 50 prior, then the model should decrease the epsilon by 0.025. If not, the model should add 0.05 instead, as it seems that more training is needed to perfect the model.

Epsilon, at its max value, is set to 1.0; at that point, all actions are random. Gradually, based on the epsilon decay, it will decrease.

- Reward Modifications: adjusting rewards

It is apparent that the rewards allocated on this environment are improperly aligned for various reasons. The penalty for going past the border gives a larger penalty than needed, causing the agent to limit its movements due to the large reward penalty, preventing exploration and attempting the track. Thus, the reward had to be modified from -100 to 0, to allow for better training, along with punishing the agent for staying on the grass at -0.05 for each step. Reward modifications were tested for the custom environment. However, it had no effect as catastrophic forgetting had a large effect on the training.

Results and Evaluation

1. Theory: Catastrophic Forgetting

Catastrophic forgetting is one of the dreadful cycles that a model could go through, particularly when using neural networks. Catastrophic forgetting, or sometimes called catastrophic interference, occurs when training a new task, or categories of tasks, a neural network may forget past information to replace present information [2]. For the custom environment, the death rate was incredibly high, meaning the agent would die almost immediately when training. While, for other environments, such as CartPole, it is not such a big deal, but for a somewhat complicated environment such as CarRacing, it produces a larger problem. While, for the majority of the classic CarRacing-v0 environment this issue was not prominent, catastrophic forgetting appeared to be occurring at a large scale throughout the custom CarRacing-v0 environment.

The car, the agent, appears to be dying too quickly to learn new information and gradually forgets everything that it learns and instead learns to remain still in an attempt to prevent surpassing the border, as the repercussion for such action produces a heavy reward penalty which the model attempts to prevent.

Going to the classic model, the same result initially occurred. The agent would learn that the barrier would produce a "die" state, causing the agent to "learn" to spin around in a circle in an attempt to prevent any deaths and the large penalty. The agent would also forget its past experiences on the track as well to do this sequence. This seems to be a common issue with others attempting to solve the environment, yet allowing more episodes to train and explore, alongside adjusting reward penalties, allowed the model to realize that the track is the targeted route.

Altering the rewards for the custom environment, however, added no change and neither did more training and exploration. For the custom environment, the agent would suffer the tragedy of catastrophic forgetting regardless.

2. Comparing the Models

The first two graphs, before displaying the diagrams shown, were trained at a maximum epsilon of 1 without decaying for 500 episodes. Then, these graphs were produced based on the results displayed.

While using a pretrained model, it appears that it did not train as quickly as a fully custom architecture. Attempting to solve this, the frozen layers were unfrozen to be retrained, which in a way removed the purpose of having a pretrained model at all. The results of using transfer learning over a course of many episodes is displayed below, with the title “Transfer Learning Exploration.” It appears that there was a significant amount of fluctuation in the graph and its learning curve. The rewards over each episode appears to go down, then up, then back up. However, there is still an ever so slight upwards trend, if the outliers are not considered.

Graph 1: Using Pretrained Model from Resnet18.

The other method, using a custom architecture produced similar results, yet had a more gradual change in results, as depicted in the graph below titled “CNN- Stable Exploration” which again, compares the episodes to the rewards. There is a gradual increase in rewards per episode, yet, not as quick as it should.

Graph 2: Custom CNN with a Stable Epsilon.

To speed up the training process, more episodes were allocated for the initial training process. Instead of 500 training episodes, there were now 1000. The epsilon decay was changed also, increased to allow for due to the fact that more learning has occurred. The

results of this are displayed in the graph below, with the classic title of “CNN.” The change seen is more immediate than the other graphs, showing the speed of learning was faster.

Graph 3: Custom CNN with less epsilon decay.

The last graph, Graph 4: CNN - More Epsilon had the best results. The initial training episodes were five times larger than the initial graphs with only 500 training episodes. Not only was there 2500 training episodes, the initial epsilon was allocated and was hiked up to 1.5 from the original 1.0. This allowed for more training and learning and can be seen with the graph below, as the values at the end surpass a reward of 300 multiple times.

Graph 4: Custom CNN with more training and more epsilon.

As it is being demonstrated, the pretrained model seems to have a more random flow of rewards per episode, while also appearing to train faster at the start than the custom built model. However, as can be seen, the models that apply a custom CNN gradually increases the rewards per episode, and results in a higher reward peak compared to the model using a pre trained model..

These assumptions have remained consistent throughout many tests, and the graphs presented above do not appear to be an outlier to this assumption. Yet, no matter the model used, none of them have achieved enough reward to be considered successful, as for no episodes did the car have a total reward of 900, much less for 100 episodes consecutively.

Discussion

1. Double DQNs (Double Learning)

While DQN are useful for many environments, a modification of DQNs, called Double DQNs - double learning - are typically better than DQN, as DQN tend to be more optimistic compared to Double DQNs, which tend to be more skeptical with the actions chosen, calculating the target Q value for taking that action [11]. Overall, Double DQNs help reduce the overestimation of q values, allowing training to be faster and have more stable learning [11]. Due to this information it could be understood that using double learning would have been more effective and beneficial for this environment, compared to the classic DQN that was applied.

2. Google Colab GPU(Training Time)

Training time is an essential part of deep learning, due to the fact that learning takes lots of time to be able to learn whatever process the model is being applied upon. For many situations, GPUs are a must as they exponentially increase the speed of training, thus training the model quicker, allocating the additional training time for perfecting the model and tweaking the extra changes.

Using Google Colab for the accessible cloud GPU allows for quicker training, yet, the cloud program is not great for environments that require rendering. For the CarRacing-v0 environment, using the classic “env.render()” after an episode was not possible. After spending a decent time trying to configure the environment on Google Colab, the best that could be achieved was rendering the last episode after running the code by creating a function that allows for “show_video()”. Yet, this was not practical as it did not allow me to check each iteration and episode and be able to analyze what was happening closely.

3. Environmental Flaws

While OpenAI has plenty, open sourced environments that are easily accessible by the general public, such as CarRacing-v0, are not entirely great in the development of the environment. The rather substandard placement of the bottom information could hinder the performance of the agent. The poorly written physics applied upon the track forces the vehicle in circles, and the reward scores collectively, along with other flaws, ruin the overall appeasement of the environment, making it difficult to work with.

4. More Exploration (Epsilon)

For any variation of a Q-Learning algorithm, exploration is how learning occurs. Allowing more epsilon allows for the model to learn more about the environment and how to solve through it. Truly, theoretically, more training and more exploration should be able to push any model to learn an environment enough to solve it.

5. Referencing the Work of Others

This project applies to a niche part of the “OpenAI gym environments” community. As explained, using a DQN for a project such as this is not considered to be the best course of action by any metrics. Yet, there has been a similar paper that used some similar applications as I did. The paper, published in December 2018, by two Stanford researchers attempted the same environment Aldape and Sowell, the writers of said paper “Reinforcement Learning for a Simple Racing Game,” could have adjusted some parts of their model to make it train better, which they did not try.

In their project, Aldape and Sowell did not crop the image, but rather left it at the classic 96 x 96, with a node size of 9216. However, they then mentioned that they did not grayscale, or even keep the RGB coloring, but rather each node took in the green channel of its respective pixel [1]. This seems like a complicated manner of gray scaling, when grayscaling would have been easier, and possibly more effective. It seems ridiculous in that manner. Furthermore, cropping should have been employed better. The coloring on the meter bar could potentially pollute the environment’s greenscreen and an 84 x 84 crop could have removed it.

Besides that, there were many similarities between their project and mine. For example, their project and mine both used a pretrained model as a form of training and both reached a similar ending point, as neither had the computational power or time to fully solve the environment.

It can be inferred that there are no great examples of solving the CarRacing-v0 environment using a DQN, thus, the next closest paper relating to the project was one created by a French Pre-PhD student who solved the problem at hand, yet just used a convolutional neural network. In the paper, Dancette, the author, states how the model was trained in the conclusion. Dancette claims that the network recognizes shapes to keep the car on the desired path [4], which would be more useful for this environment than attempting a DQN, as the most important part of this environment is recognizing the track and staying within its boundaries.

6. Conclusion

While I have had experience solving other OpenAI environments, this one seems to be the most challenging one that I have attempted. Since so much time was spent trying to train the custom and classic environment using a DQN, there was no time to apply different reinforcement learning techniques. While the results did not live up to the high standards that I expected, nor did the environment reach the 900 reward threshold, a lot was learned, as modifying the environment and the model to test changes was both interesting and gave me insightful knowledge. Overall, throughout all the trails and testing, the outlook on this project remains positive.

References

- [1] Aldape, P., & Sowell, S. (2018, December 18). Reinforcement Learning for a Simple Racing Game. Retrieved September 14, 2020, from <https://web.stanford.edu/class/aa228/reports/2018/final150.pdf/>
- [2] Chen, Z. (., & Liu, B. (n.d.). *Lifelong machine learning*.
- [3] Choudhary, A. (n.d.). [DQN Formula]. Retrieved September 15, 2020, from <https://www.analyticsvidhya.com/blog/2019/04/introduction-deep-q-learning-python/>
- [4] Dancette, C. (2018, April 09). [Tutoriel] Conduite autonome par imitation grâce à un réseau de con... Retrieved September 10, 2020, from <https://cdancette.fr/2018/04/09/self-driving-CNN/>
- [5] FreeCodeCamp.org. (2018, April 16). Improvements in Deep Q Learning: Dueling Double DQN, Prioritized Experience Replay, and fixed... Retrieved September 14, 2020, from <https://www.freecodecamp.org/news/improvements-in-deep-q-learning-dueling-double-dqn-prioritized-experience-replay-and-fixed-58b130cc5682/>
- [6] Sentdex, H. (n.d.). [Q-Learning Formula]. Retrieved September 15, 2020, from <https://pythonprogramming.net/q-learning-reinforcement-learning-python-tutorial>
- [7] Shyalika, C. (2019, November 16). A Beginners Guide to Q-Learning. Retrieved September 14, 2020, from <https://towardsdatascience.com/a-beginners-guide-to-q-learning-c3e2a30a653c>
- [8] Zychlinski, S. (2020, May 26). Qrash Course: Reinforcement Learning 101 & Deep Q Networks in 10 Minutes. Retrieved September 14, 2020, from <https://towardsdatascience.com/qrash-course-deep-q-networks-from-the-ground-up-1bbda41d3677>
- [9] A Beginner's Guide to Deep Reinforcement Learning. (n.d.). Retrieved September 14, 2020, from <https://wiki.pathmind.com/deep-reinforcement-learning>
- [10] Fine Tuning Torchvision Models¶. (n.d.). Retrieved September 14, 2020, from https://pytorch.org/tutorials/beginner/finetuning_torchvision_models_tutorial.html
- [11] FreeCodeCamp.org. (2018, April 16). Improvements in Deep Q Learning: Dueling Double DQN, Prioritized Experience Replay, and fixed... Retrieved September 14, 2020, from <https://www.freecodecamp.org/news/improvements-in-deep-q-learning-dueling-double-dqn-prioritized-experience-replay-and-fixed-58b130cc5682/>
- [12] Torchvision.models¶. (n.d.). Retrieved September 14, 2020, from <https://pytorch.org/docs/stable/torchvision/models.html>
- [13] What is Machine Learning? A definition. (2020, May 29). Retrieved September 14, 2020, from <https://expertsystem.com/machine-learning-definition/>
- [14] What is PyTorch?¶. (n.d.). Retrieved September 14, 2020, from https://pytorch.org/tutorials/beginner/blitz/tensor_tutorial.html