Learning Atari through Dueling Double Deep Q-Learning

William Fraher

Abstract

Deep Q-Learning has recently become a very popular algorithm for solving problems in reinforcement learning. Due to its generality and superhuman performance in the Atari domain, Deep Q-Learning has witnessed many improvements over the past few years. Two of these improvements, Dueling Deep Q-Learning and Double Deep Q-Learning, can be combined to achieve vastly superior results to those of the original algorithm. In this paper, we demonstrate the effectiveness of combining Dueling Deep Q-Learning and Double Deep Q-Learning in the Atari domain.

Introduction

Many recent developments in reinforcement learning started with the development of Deep Q-Learning (Mnih et al. 2015). The Deep Q-Learning algorithm is able to learn complex decision-making policies with a multi-layer neural network. These decision-making tasks involve choosing an action given the state of some environment. The action taken causes the environment to generate a reward, which advances the state of the environment. Instead of learning to approximate the gradient of expected rewards, as has been done with techniques like Trust Region Policy Optimization (Schulman et al. 2015), Deep Q-Learning approximates the value of state-action pairs. These state-action values are called Q-values. It does this by using a neural network which takes the state of an environment as input and uses it to approximate the value of every action that can be taken from that state. The action values are stored in a vector, and the agent selects the best action, according to the neural network, by taking the argmax of the action values. These

action values are estimations of the future rewards the agent could obtain from that state.

The Deep Q-Learning algorithm was able to achieve superhuman performance on Atari 2600 games (Mnih et al. 2015). This was accomplished by using experience replay, which samples (S, a, r, S') transitions from a memory buffer. Each sample is then used to train the network. This has been shown to improve the stability of learning. The DQN algorithm also uses a target network, which has similarly been shown to stabilize learning. A convolutional neural network was also employed to generalize spatial information.

While this method was very effective in the Atari domain, several improvements to it have been introduced. The Double DQN update has been shown to further increase stability of learning by decoupling the training step (van Hasselt et al. 2015). This works by having an online network choose an action, but the target network evaluate that action. This value is then used in the training step. This is due to how the original Deep Q-Learning algorithm overestimates state-action values, which could cause the agent to act poorly in accordance to such estimations.

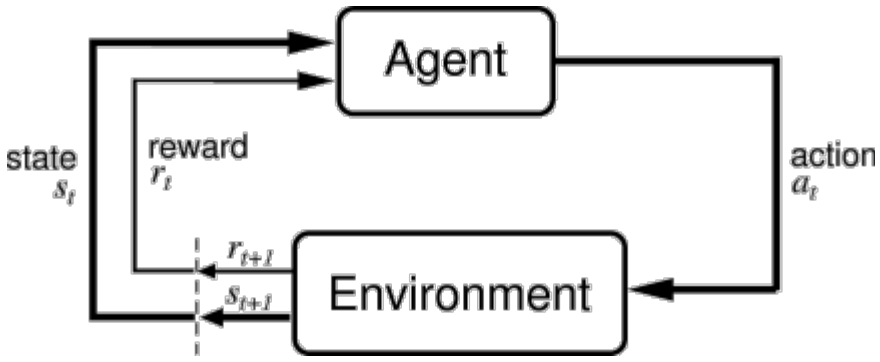
Another improvement to DQN known as Dueling DQN has been shown to result in faster convergence in the Atari domain (Wang et al. 2015). This works by having the network separately evaluate state and action values, and then combine them to evaluate state-action pairs.

By combining the stability of Double DQN with the convergence of Dueling DQN, we are able to solve Atari games in a matter of hours, in fewer than a million frames in some cases. This is considerably more efficient than the original Deep Q-Learning algorithm, which trained on each game for millions of frames (Mnih et al. 2015).

Background

Reinforcement learning. Reinforcement learning attempts to solve the problem of controlling an agent. Given an environment, reinforcement learning seeks to learn how to act in the environment, given environmental states, in a way that maximizes a scalar reward. The agent is never told what the best action is, it must use learning to decide optimal actions on its own.

Reinforcement learning can be viewed in the following abstraction:



(image from Sutton and Barto, Reinforcement Learning: An Introduction, 1998)

Where S_t is the state of the environment at time step t, a_t is the action given by the agent at time step t, and r_t is the scalar reward given by the environment for executing a_t. Each a_t causes the environment to return r_t and advance the state of the environment. This process continues until some kind of termination occurs, for example, a boolean signal from the environment could determine termination. The sequence of states, actions, and rewards is called a rollout.

Reinforcement learning seeks to find a policy π that determines actions, given S_t, that maximize the expected sum of rewards throughout the rollout. We say the expected sum of rewards as many environments are stochastic. As such, we cannot always predict the rewards we will achieve, so we maximize the overall expected rewards so we are not biased towards being lucky in a particular rollout.

This typically works by having a set of parameters, θ, which a reinforcement learning agent uses to make decisions. This could be the weights of a neural network, for instance, or weights of a linear combination, to name some examples. As more rollouts are collected, the agent learns which actions warrant higher rewards and tunes θ accordingly.

A reinforcement learning agent often is engineered to prefer the rewards we achieve sooner over the rewards we could achieve later. This is done by weighting future rewards with a hyperparameter γ. This is because, if we could achieve a reward very far in the future of a rollout, we might fail to achieve it due to chance. As such, it's better to be biased to more accessible rewards as we're more likely to achieve them. We accomplish discounting by multiplying the value of a reward k steps in the future with γ^k. A discounted sum of rewards, which is used to train agents, is determined with

$$\sum_{t'=t}^{n} (r'_t * \gamma^{t'})$$

where n is the length of the rollout. The agent's goal is to maximize this discounted sum of rewards.

Supervised learning and neural networks. Unlike reinforcement learning, supervised learning is concerned with approximating a predetermined output given a corresponding input. This is done by having the agent learn from pairs of inputs and outputs. The set containing these pairs is called training data. By training on the items in a set of training data, the agent learns to approximate the value of future inputs, which are not given an explicit value.

Supervised learning is extremely effective in tasks such as regression and classification. A common example of supervised learning is the classification of handwritten digits. Given training data consisting of images of handwritten digits and their values, we can use supervised learning to generalize a function mapping from these images to their values. This function can then be tested on new images of handwritten digits, where the value of these digits is not given to the agent. If the training is successful, it maps these new images to their actual values. In reinforcement learning, we are given a similar task. Given states of an environment, we estimate the expected rewards that state can yield. With this in mind, we can apply many techniques from supervised learning in reinforcement learning.

Artificial neural networks are a popular supervised learning technique. These work by implementing computational units that act as abstractions of neurons in the brain. Each artificial neuron stores its own weights, and takes a linear combination of these weights with the network's inputs.

These artificial neural networks are combined in layers, each layer taking the outputs of the previous layer as input. In between the layers, functions called nonlinearities are applied, such as the sigmoid function, which maps the outputs of a layer to values between 0 and 1. Introducing such nonlinearities allows the network to predict non-linear functions, unlike linear or logistic regression (Ng, Coursera). This flexibility makes neural networks very powerful, and has led to

many developments in computer vision, natural language processing, and other related fields.

Many recent successes in reinforcement learning have stemmed from applying neural networks to reinforcement learning tasks. Modern reinforcement learning algorithms that employ multi-layer neural networks are called deep reinforcement learning algorithms.

Neural networks and reinforcement learning. When applying a supervised learning technique to a given problem, we are given pairs of inputs and outputs from which we derive a prediction function. However, in reinforcement learning, we aren't given explicit training examples. We have to create it from the environment.

Supervised learning, in the context of neural networks, applies a prediction function to training examples. Based off of the accuracy of the prediction, it tunes itself using a procedure called backpropagation. Backpropagation perturbs the weights of the neural networks using gradients, and brings them closer to predicting the training example given a loss (Ng, Coursera). This loss is calculated by seeing how much the prediction and training example differ. This is often employed using a function such as mean squared error or cross entropy loss.

Deep Q-Learning. When using neural networks, we need some kind of direction for how to tune the weights of the network, given by the loss function. This will give us relevant state-action values which our agent can select the argmax of. The problem in reinforcement learning is to decide what we use as training data for our neural network. Different algorithms use different ways of generating training data, for instance, policy gradient methods map from states to a sum of discounted future rewards (Abbeel, Deep RL bootcamp). Deep Q-learning, however, maps from states to the following value:

$$r + \gamma * max_{a_{t+1}} Q(S_{t+1}, a_{t+1})$$

where $Q(s, a)$ is the Q-value of a state-action pair, determined by the neural network, and a_{t+1} is an action that can be taken from S_{t+1}. So $max_{a'}$ iterates over each action that can be taken from that state. This means that the neural network attempts to map from states to this target value. The target value is determined by adding the reward with the product of the discount and the maximal Q-value for

the next state. As the deep Q-learning agent acquires new rollouts, it pushes its decision-making weights towards this target value.

To act in the environment, a deep Q-network, or DQN, takes a state as input, and approximates the value of each action that can be taken from that state. It then decides which action would be the most valuable, and chooses that action. The agent repeats this process, collecting the rest of the rollout. Once the rollout is collected, it tunes it with respect to the following loss:

$$[(r + \gamma * max_{a_{t+1}} Q(S_{t+1}, a_{t+1})) - Q(S_t, a_t)]^2$$

using backpropagation, a procedure for optimizing neural networks. We use mean squared error here as opposed to the difference between the two terms to prevent a negative loss from occurring. If a negative loss occurred in this context, it would tell the neural network that it is doing well, even though it would actually represent an error. We also use the mean squared error as opposed to categorical crossentropy loss as deep Q-learning is essentially a regression problem.

To replicate the deep Q-learning algorithm used to play Atari games, we need to introduce several augmentations.

Improvements to DQN

Exploitation vs exploration. A problem in Q-learning is that the agent can easily get stuck in a local optimum (Silver, 2015). To alleviate this, we can have the agent sometimes act randomly. We maintain this with a hyperparameter, epsilon, which is always between 0 and 1. When it is time for our agent to act, we generate a random number between 0 and 1. If that number is greater than epsilon, we have the agent compute an action. Otherwise, we have the agent act randomly. By forcing the agent to make random moves, we can have it explore other rollouts, which may be better, that the agent currently has not explored. This is typically done by starting epsilon at 1.0 and annealing it over an amount of frames that the programmer decides. This will prevent the agent from getting stuck in a local optimum at the beginning of training, when the weights of the DQN are mostly random. There is still opportunity for the agent to make random moves later. In this paper, we explore epsilon decay from 1 to 0.1.

Target network. If we have the agent always perform the mean squared error update previously described, the updates of the network can spiral out of control

from small influences (Juliani). Q-learning becomes more stable when we introduce a second set of parameters, θ^-, that we update periodically. Notationally, we call this the target weights. We call the weights we use to evaluate actions the online weights. The target weights are used to value actions of our training target, giving us the following error function:

$$[(r + \gamma * max_{a_{t+1}} Q(S_{t+1}, a_{t+1}; \theta^-)) - Q(S_t, a_t; \theta)]^2$$

where $Q(S, a, \theta)$ is the Q-value of action a at state S, estimated by weights θ. This means that we create the Q-values of our target with the target weights. We determine the mean squared error between the corresponding target and what our online network computes to get the agent's error. To update the target weights, instead of using backpropagation as we would for the online weights, we copy the online weights over to the target weights. We do this every τ time steps. We decide the frequency at which the target weights are updated.

Experience replay. We store each S_t, a_t, r_t, S_{t+1} transition in a memory buffer. When training our agent, we sample transitions randomly from the buffer. We use these transitions to update the DQN, using the update rule described above. By sampling the transitions randomly, the DQN avoids a bias that can come from sampling frames in succession. For example, imagine training off of every frame in Pong. Each frame is fairly similar to the previous frame when sampling them one after the other, in terms of the layout and the reward. Consequently the DQN would be tuned more towards the result of the previous frame, and could have a difficult time producing relevant results for a future rollout. Its generalizations would be biased as they would be tuned more towards the previous frame as opposed to evaluating any random frame from a rollout. As such, by sampling transitions randomly, the DQN's training is more robust.

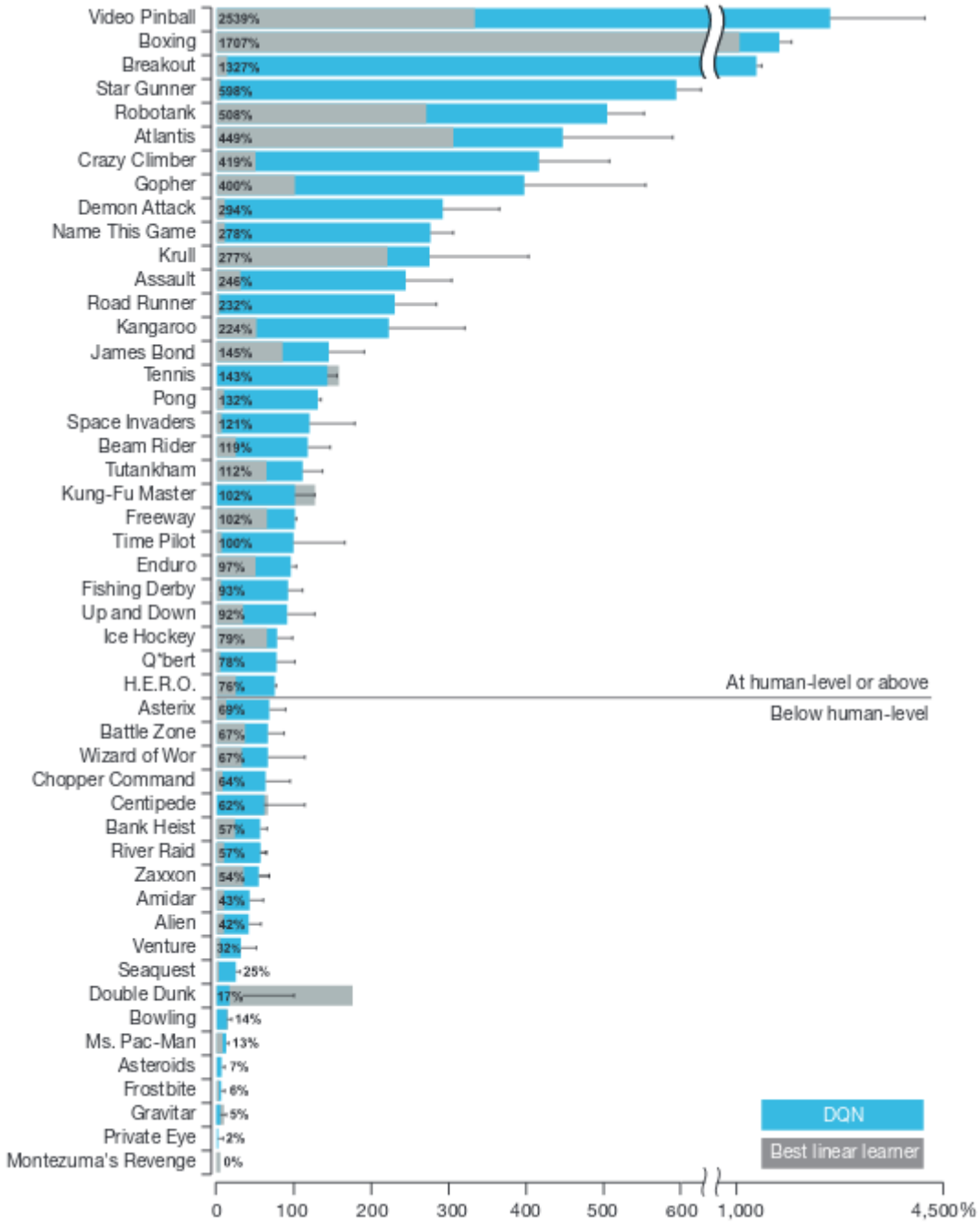
Frame stacking. When training on an Atari game, a single frame may not have enough information needed to intelligently compute an output. In Pong, for example, a single frame tells us the position of the paddles as well as the ball. However, we may need to take the velocity and direction of the ball into account in order to figure out how to deflect it. Other such calculations could be important in different Atari games, such as Breakout and Space Invaders. As humans, we can see how the ball moves throughout many frames. To achieve human-level performance in one of these games, we need a workaround that will let us process this kind of temporal information. We could use recurrent neural nets to do this (Hausknecht and Stone, 2015). In the original DQN paper, the authors instead

consider multiple frames. Every time the environment goes through four frames, the agent only keeps one. The agent stacks up four of these frames, skipping 12 of every 16 frames between each decision it makes. It stacks each of these frames on top of one another and then passes all four of the frames into the neural network at once. The original paper uses an 84 by 84 by 4 dimensional tensor. However, wrappers in OpenAI Baselines use a 334 by 84 by 1 dimensional tensor, so the dimensions of the stacked tensor are not particularly sensitive.

Convolutional Neural Networks. Recent successes in computer vision such as One-Shot Learning and Neural Style Transfer have employed convolutional neural networks, which allow a neural network to generalize spatial information (Ng, Coursera). These networks compute outputs by convolving matrices instead of multiplying them. This works by optimizing filters, which are convolved against images and other layers, as well as weights, which are used conventionally. This allows the neural network to figure out how configurations of spatial data, such as pixels, should be weighted. In the context of Atari games, this could be used to weight the pixels around important objects more than blank spaces. In Pong, this can cause the network to give more consideration to the area around the ball, for instance.

The original algorithm, proposed by Mnih et al., uses a convolutional neural network with frame stacking and experience replay to achieve superhuman performance in Atari games. In many games, such as Pong and Q*Bert, the agent was able to achieve comparable scores to a human. The agent was also able to achieve considerably higher scores, such as in Video Pinball and Breakout. In spite of this, it was not able to learn every game effectively, as it had a lot of trouble with Private Eye and Montezuma’s Revenge.

The following picture depicts the scores achieved by the agent, normalized against human performance.



(image from Mnih et al. 2015)

This implementation is powerful, however, improvements can be made to it to make Deep Q-Learning even more effective.

Double DQN

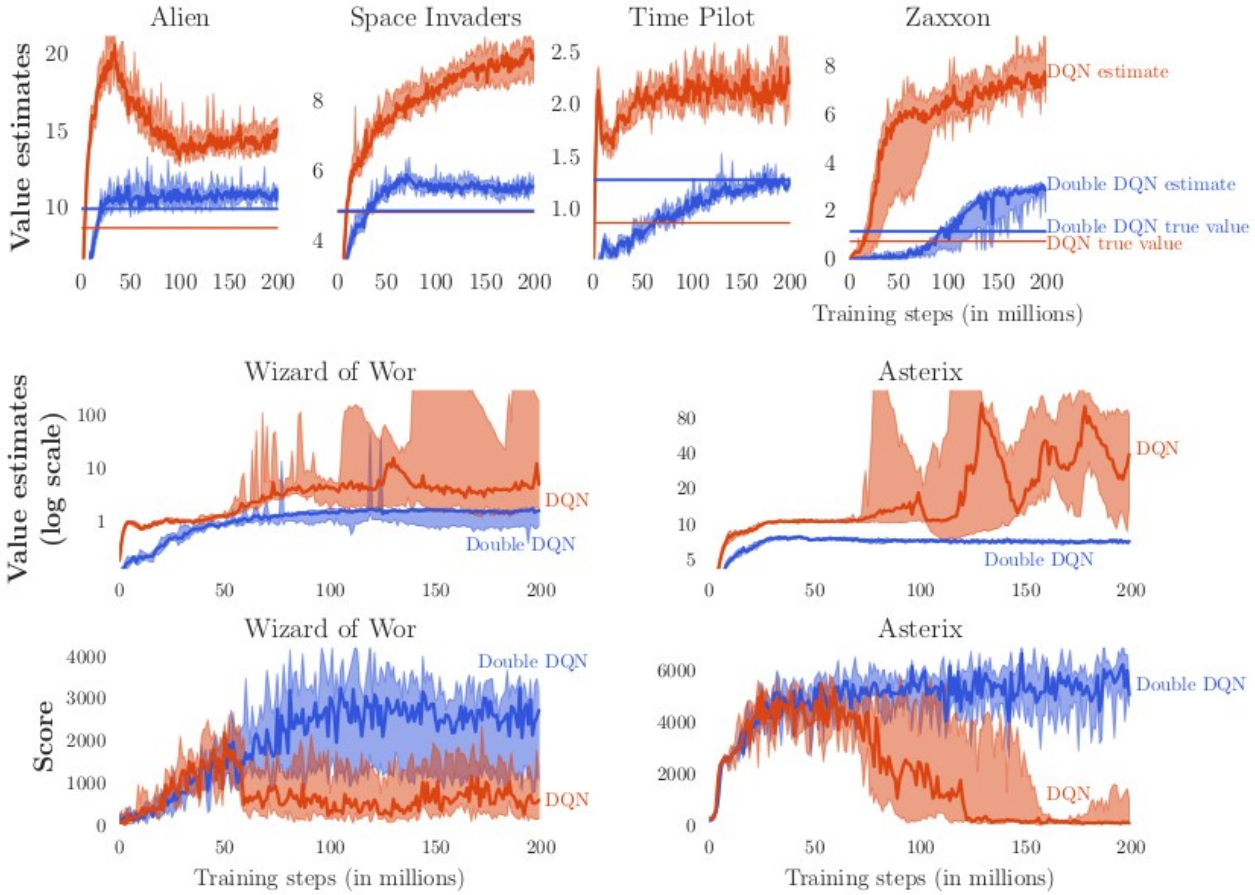
Q-learning, as was implemented by Mnih et al., suffers from an overestimation bias (van Hasselt et al, 2015). If the Deep Q-learning agent continually gives some

state-action pairs an unnecessarily high value, it could learn to retain such an overestimation in the future. This is because the DQN differentiates towards its own output, even if we use a target network. As such, any overestimation bias it retains might persist throughout the training process. We could get around this by replacing our former update, which selects $max_{a_{t+1}} Q(S_{t+1}, a_{t+1})$, with a more sophisticated update rule.

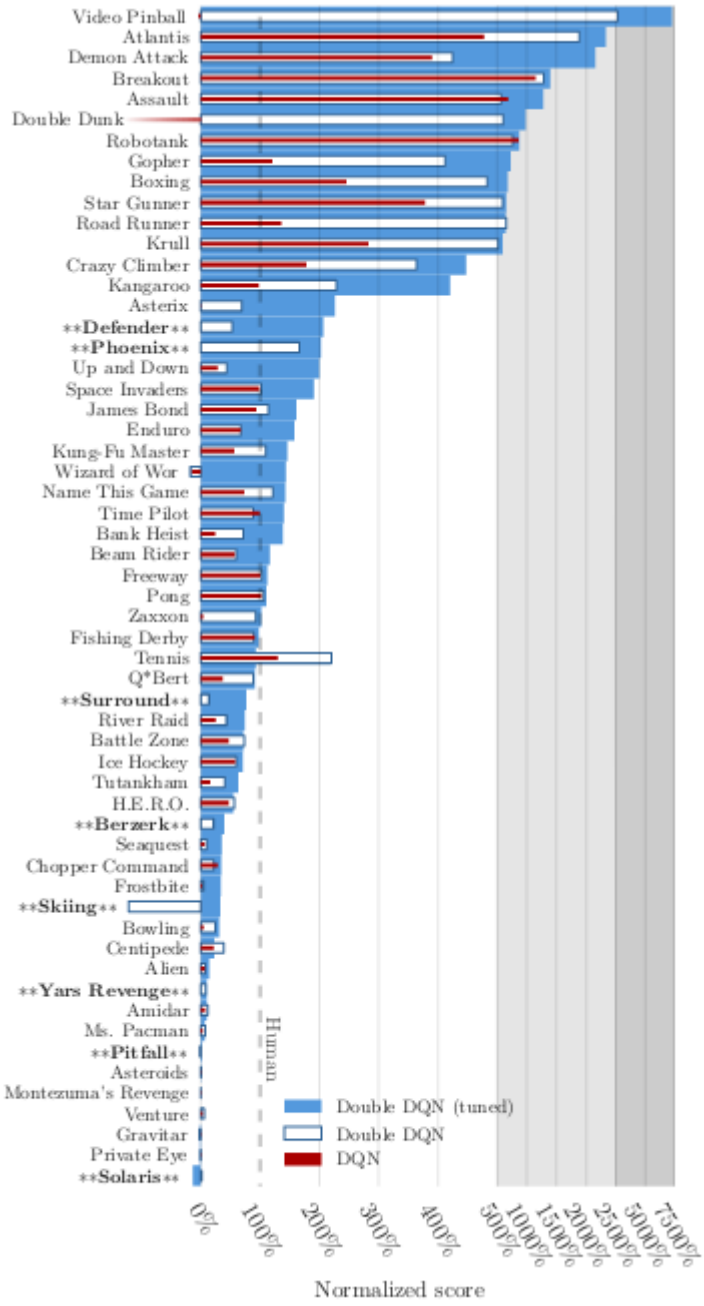
$$(r + Q(S_{t+1}, argmax_{a_{t+1}} Q(S_{t+1}, a_{t+1}; \theta^-)) - Q(S_t, a_t, \theta))^2$$

We use target weights like we did earlier. Here, we decouple the $max_{a_{t+1}} Q(S_{t+1}, a_{t+1})$ selection by evaluating the actions of the next state with the target weights and selecting them with the online weights. This prevents any discontinuity from one set of weights preferring an action to another. For instance, one set of weights might think moving up is the superior action in a game of Pong, whereas a different set of weights could prefer moving down. As such, when computing the error, it wouldn't make sense to compare the values of two different actions. By selecting actions with the online network and evaluating them with the target network, we're able to have a more stable error function.

Van Hasselt et al. actually created some graphs showing how the value estimations and scores change between the initial DQN update and the Double DQN update.



These graphs show that the lower value estimates can result in higher scores overall on certain games. Van Hasselt et al. also demonstrated this on other Atari games.



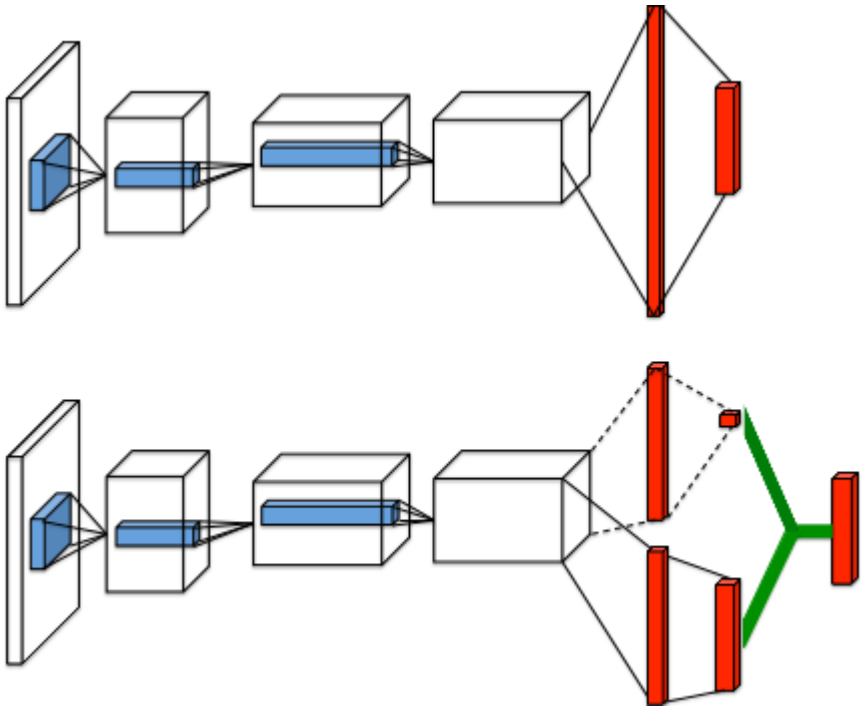
As such, the Double DQN update is important to consider when impelmenting a DQN.

Dueling DQN

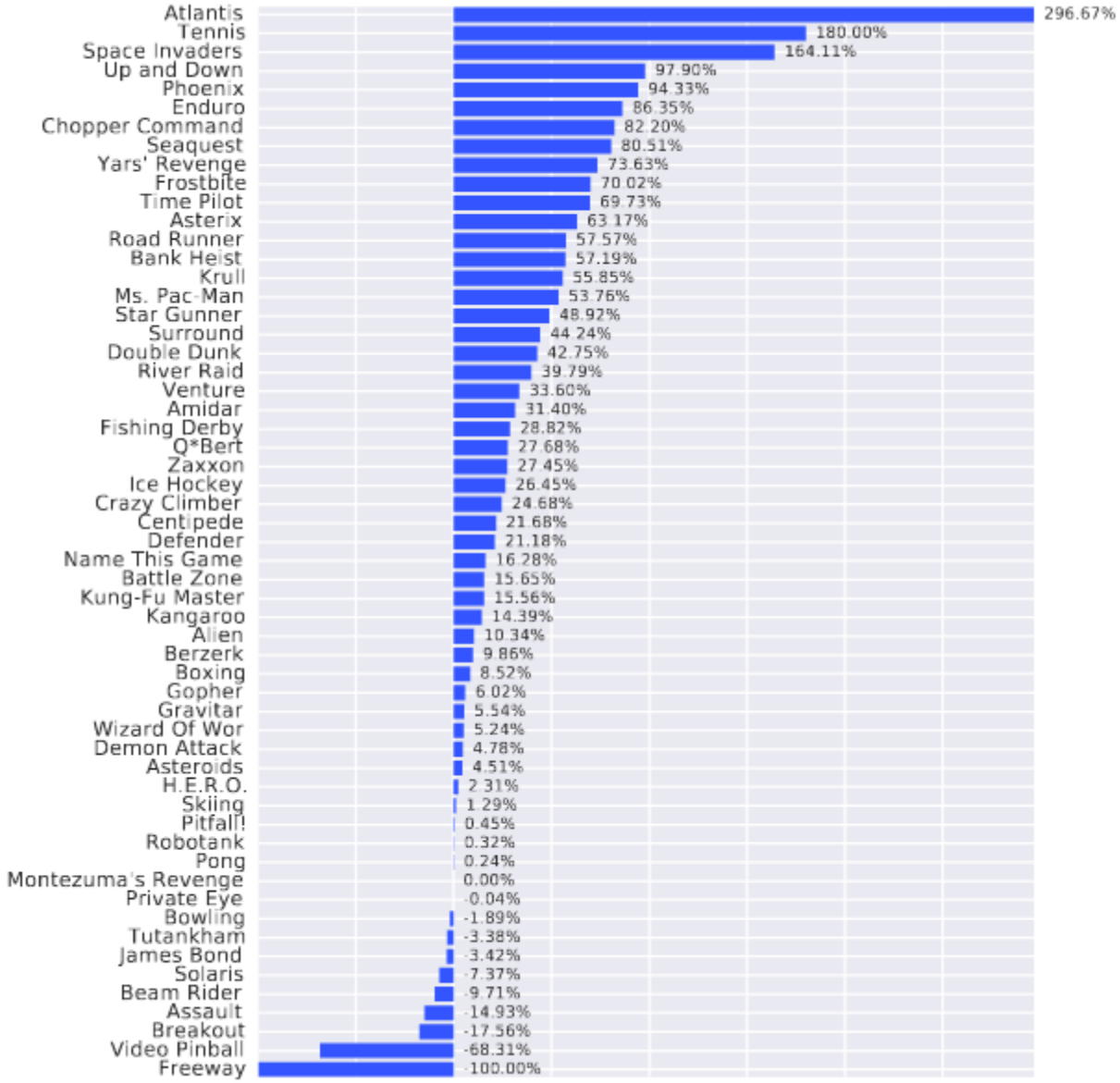
Another important augmentation of Deep Q-Learning is to use the Dueling DQN architecture. This architecture works by evaluating the state and the actions separately. Q-values are computed by adding the estimated value of the state with the estimated action values.

$$Q^{\pi}(S, a) = V^{\pi}(S) + A^{\pi}(S, a)$$

The neural network does this by splitting its final layer into two streams. One stream computes the estimated state value, and the other computes the advantage of taking an action on that state. Each stream has its own weights, so the neural network can distinguish between these two objectives. Wang et al. illustrate this architecture as



where the first network represents a conventional DQN and the second network represents the Dueling DQN architecture. Wang et al. also compared the Dueling DQN architecture to the Double DQN architecture for different Atari games.



(image from Wang. et al. 2015. Scores are normalized against the network used by van Hasselt. et al.)

The Dueling DQN performs much more effectively on most Atari games. However, the Double DQN is still more effective on a few games. Video Pinball, which DQN has had the most success with throughout all Atari games, has had better results with the Double DQN than with the Dueling DQN.

Current Implementation

The attached code is able to consistently achieve a score of 20.0 in Pong, which is superior to the scores in Wang et al.'s implementation. This may be due to different hyperparameters or a different architecture of the neural network. Wang et al. train each stream with 512 hidden neurons. In the included implementation, each stream has 8192 hidden neurons. The included implementation also has four convolutional layers, the last of which has a kernel size of 7 by 7 and a stride of 1. This may be a deciding factor in the higher score. It is possible that the extra layer and the larger amount of hidden neurons resulted in a higher score.

The included implementation uses the DeepMind wrapper from OpenAI Baselines to facilitate frame stacking. This causes the network to take a 334 by 84 by 1 tensor of resized, grayscale pixels. The original agent uses an 84 by 84 by 4 tensor of grayscale pixels. Grayscale pixels are taken as opposed to colorized pixels as the color of most objects in an Atari game is typically not very important. By taking grayscale frames, we are able to reduce the dimensionality of an Atari frame from 84 by 84 by 3, as we need three color channels, to 84 by 84 by 1. This can significantly increase how many frames we can store in memory. This should also make training more efficient as the agent does not need to distinguish objects by color. The grayscale images do still retain some information from the colorized versions, as the tones of the grayscale pixels are determined from their original colorizations.

Unfortunately there wasn't enough time to test out this implementation against other Atari games. Partially trained weights were obtained for Space Invaders and Breakout, however, these weights aren't competitive with those of the original papers. If there were more time to train this implementation on different games, a careful analysis could be conducted comparing this architecture against those of Wang et al. and van Hasselt et al. It is possible that the increase in hidden layer sizes could lead to significantly superior results, as the Pong scores represent that this implementation may be superior.

This implementation was mostly coded by me though parts of it were coded by Arthur Juliani. It also uses TensorFlow and the wrapper from OpenAI Baselines. The neural network structure is based off of that of Juliani and Wang et al. I coded an implementation of the vanilla DQN, without the double DQN update or the Dueling DQN architecture, using none of Juliani's code, for comparison purposes. However I didn't have enough time to compare it against the included

implementation or against the original DQN algorithm, though it seems to find a good policy very quickly for Pong.

While these results are pretty impressive, as we're able to learn many Atari games without changing our neural network, using only screen output, scores, and no prior knowledge of any of the games, there is still more we can do to create a more sophisticated agent. Recall that some games, such as Private Eye, did not come close to human performance. As such, there are some techniques and architectures we could use to further increase our agent's performance.

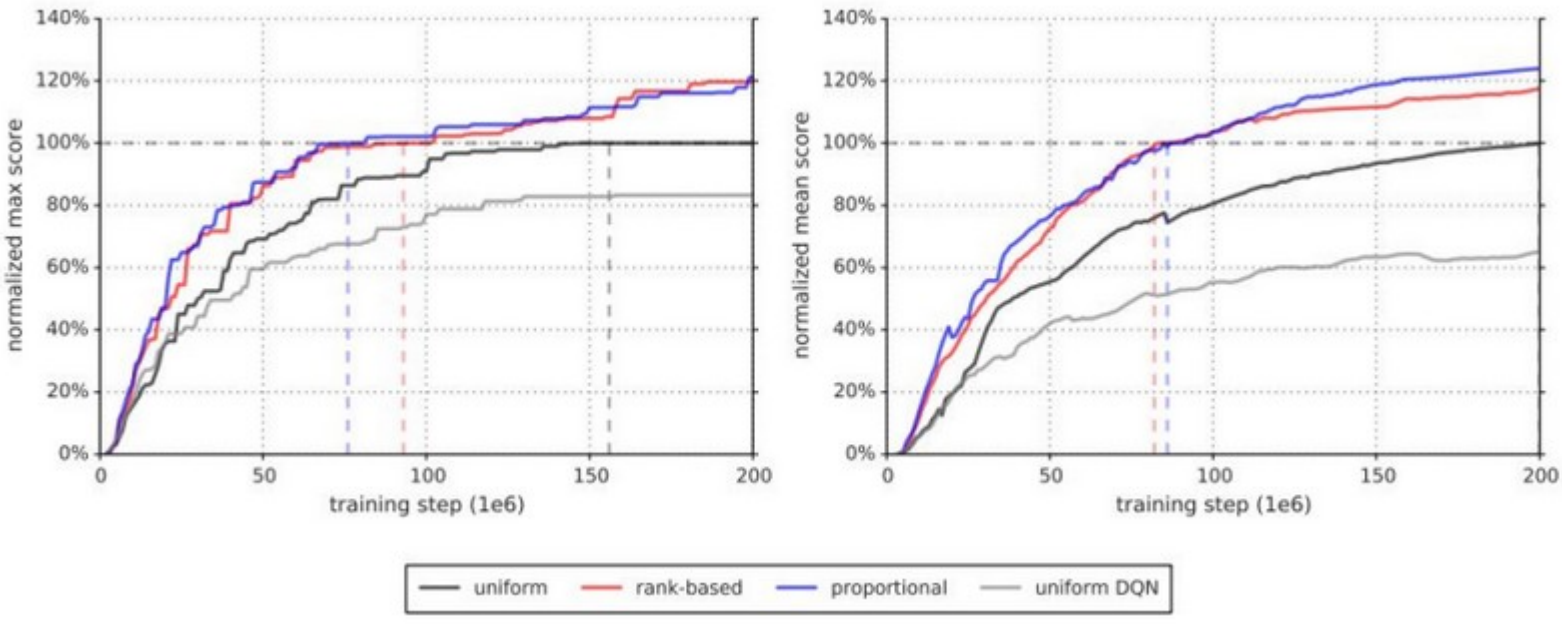
Prioritized Experience Replay

We could, instead of sampling transitions from the buffer at random, sample them based off of TD error. In other words, we sample transitions based off of how poorly our neural network predicts the target value. We use our error, such as

$[(r + \gamma * max_{a_{t+1}} Q(S_{t+1}, a_{t+1}; \theta^-)) - Q(S_t, a_t; \theta)]^2$ for the DQN architecture without the double DQN update and

$(r + Q(S_{t+1}, argmax_{a_{t+1}} Q(S_{t+1}, a_{t+1}; \theta^-)) - Q(S_t, a_t, \theta))^2$ if we use use the double DQN update.

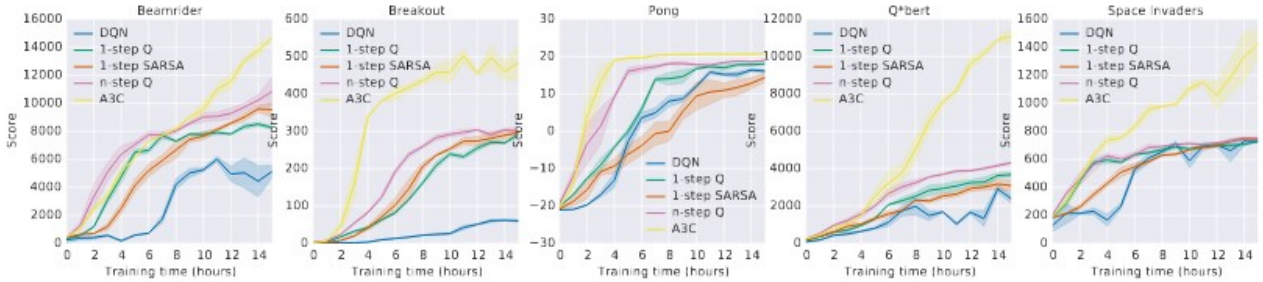
We store all of these errors in our experience buffer and train off of those with the highest error. This has been shown to lead to much faster results (Schaul et al. 2016).



In the previous graph, different adaptations of experienced replay are given. These graphs show the median normalized scores over 57 different Atari games. Methods that sample based off of error, being the rank-based and proportional methods, perform more effectively than the uniform sampling methods.

Asynchronous Advantage Actor-Critic

A policy gradient method known as Asynchronous Advantage Actor-Critic uses a Deep Q-learning agent to determine how a policy gradient network should be optimized. It was shown by Mnih et al. to be considerably more effective than previous Deep Q-learning methods in the Atari domain.

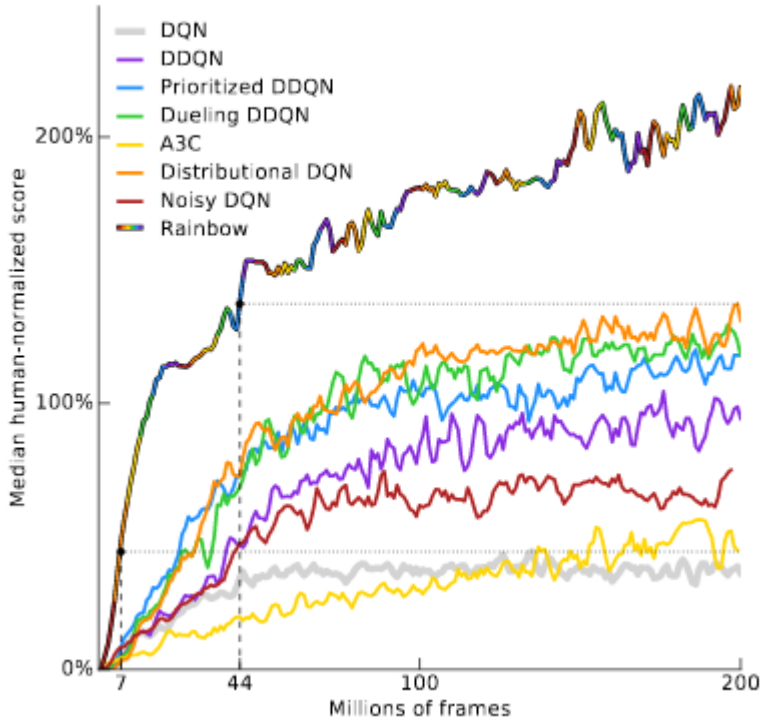


(image from Mnih et al. 2016)

These graphs show that Asynchronous Advantage Actor-Critic (A3C) is able to achieve considerably greater performance in some Atari games in less time. This algorithm has also been proven to be effective in 3D environments, such as the TORCS driving simulator and MuJoCo robotics simluations (Mnih et al. 2016).

Rainbow DQN

More improvements to DQN have since been released, such as Rainbow DQN, which combines more improvements from DQN to achieve considerably high scores, even surpassing those of A3C (Hessel et al. 2017).



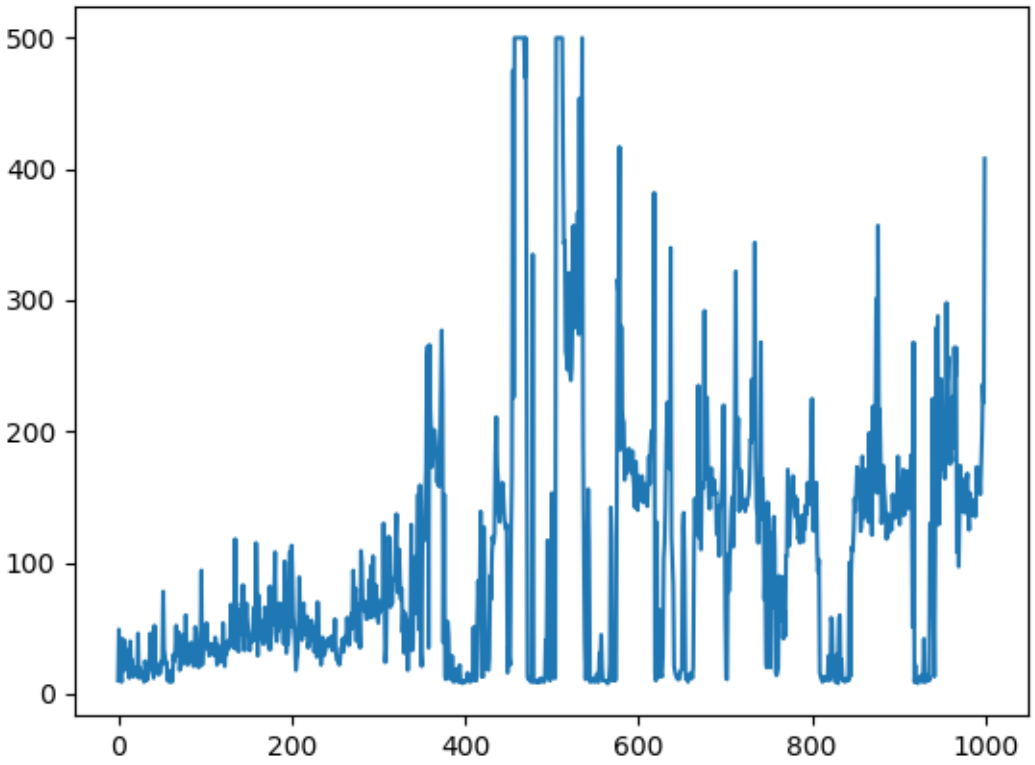
(image from Hessel et al. 2017)

This new DQN algorithm used a detailed ablation study to determine which improvements to DQN are the most effective (Hessel et al. 2017). In doing so, the authors were able to engineer an extremely effective DQN, which achieved state-of-the-art results at the time.

Conclusion

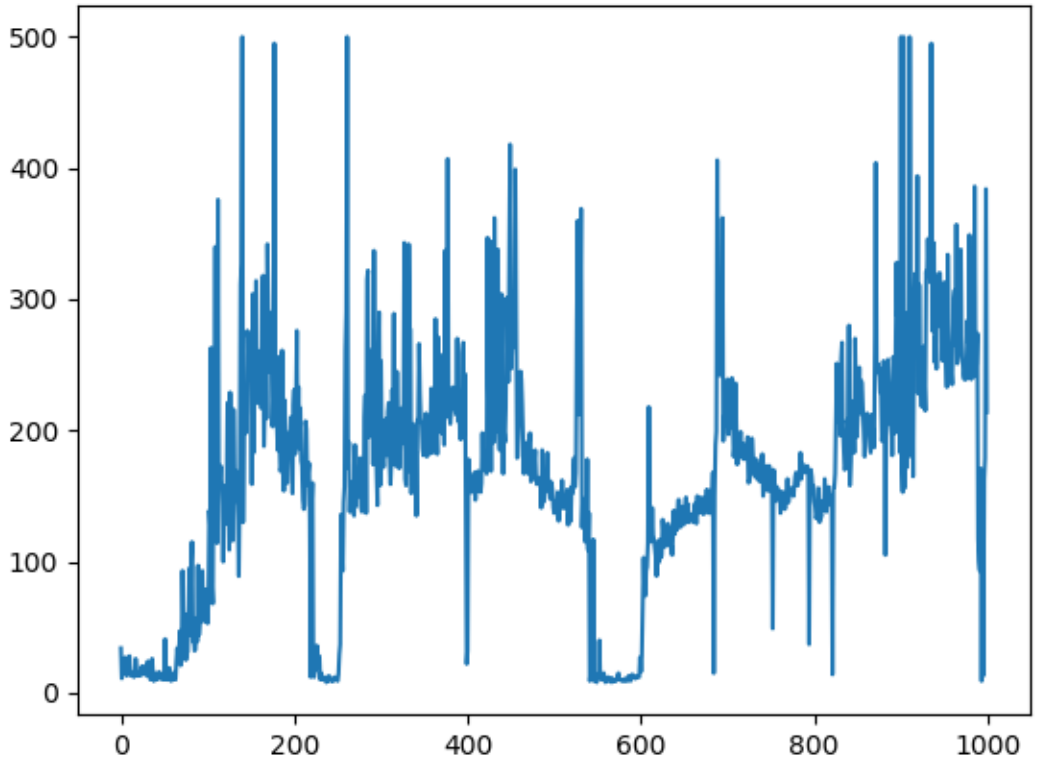
Deep Q-learning has proven to be an effective method for approximating value functions. By taking only reward signals, screen images, and a list of legal actions, we are able to create agents that can learn Atari games without any prior knowledge of those games, by approximating the value of state-action pairs. However, while this is effective, much of modern reinforcement learning research focuses on policy gradient methods. These methods approximate a gradient of rewards as opposed to state-action pairs. State-of-the-art techniques, such as proximal policy optimization, have this goal in mind (Schulman et al. 2017). Other techniques work without any kind of differentiation, such as Neuroevolution of Augmenting Topologies (Stanley and Miikkulainen, 2002). These methods often

use a genetic algorithm to find powerful policies. Some methods, such as Augmented Random Search and the Cross Entropy Method, can be surprisingly effective. I tested CEM against a Dueling Double DQN for the pole balancing problem, as well as some DQN variants. The DQN variants did not perform nearly as well as CEM in terms of score and stability.

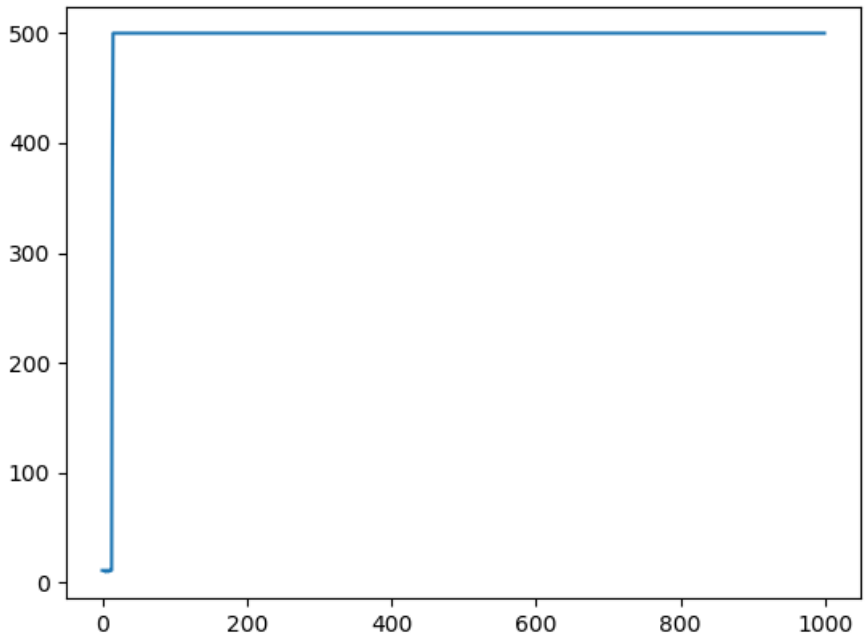


This graph shows the scores of a DQN with experience replay on the pole balancing problem. The vertical axis represents the score achieved, and the horizontal axis represents the number of episodes throughout the training.

Dueling DQN performs more effectively than a regular DQN, as shown by the following graph.



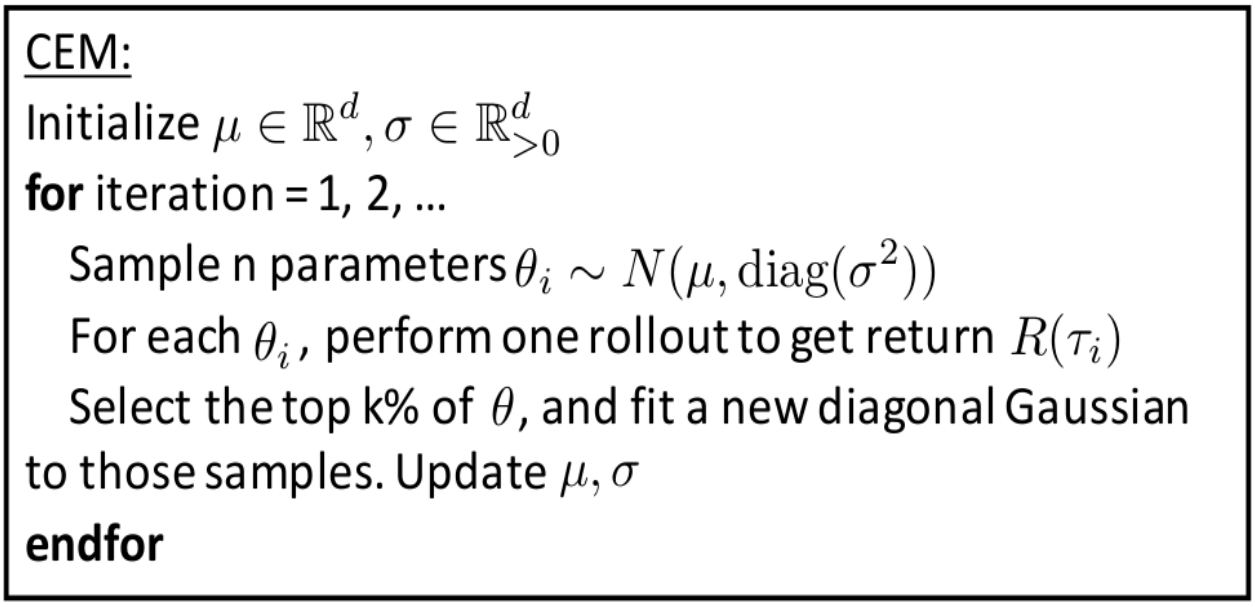
However, the Cross Entropy Method performs considerably more effectively. The following graph shows performance of CEM on the pole balancing problem.



It solves the pole balancing problem after 15 iterations. Each iteration collects 20 rollouts of the pole balancing problem, so technically CEM solves the environment after 300 episodes. However, once it reaches this solution, it is able to achieve the highest score in the pole balancing problem throughout the rest of the rollouts.

CEM is effective in this problem as there is a relatively small parameter space, however, it is extremely inefficient for tasks with large parameter spaces. I tried to use CEM to learn Atari games based off of screen input, but my computer was not able to run CEM as it has $O(n^2)$ space complexity. This is because we need to form a covariance matrix with size n by n, where n is the length of the parameter vector. CEM also samples parameters randomly, similarly to hill climbing search, so it may take a long time to converge on an effective policy if we are working with many parameters.

To be specific, I used the following algorithm.



(image from Berkeley Deep RL Bootcamp, Peter Chen, John Schulman, Peter Abbeel)

Even more embarrassingly, Augmented Random Search is very effective. Particularly, it has been proven to achieve higher performance than state-of-the-art techniques such as Proximal Policy Optimization and Trust Region Policy Optimization on simulated robotics tasks (Mania, Guy, and Recht, 2018). However, Augmented Random Search is a derivative-free method inspired by

random search. As such, it is quite surprising that this kind of method could be more effective than sophisticated policy gradient methods.

Reinforcement learning has come a long way from the original experiments with DQN (Mnih et al. 2013, 2015). Neural nets are being used to create new reinforcement learning algorithms that continue to push the boundaries of the discipline. New deep reinforcement learning algorithms often combine innovations in deep learning in creative ways. For instance, Evolved Policy Gradients trains a gradient-based reinforcement learning agent that evolves a differentiable loss function (Houthooft et al. 2018). Deep reinforcement learning is still in its infancy and will see many enhancements, which may complete unsolved problems like end-to-end learning for self driving cars. Future enhancements may combine innovations from many of these techniques to produce truly amazing technology.

References

V. Mnih *et al.*, “Human-level control through deep reinforcement learning,” *Nature*, vol. 518, no. 7540, pp. 529--533, 2015.

John Schulman, Sergey Levine, Philipp Moritz, Michael I. Jordan: “Trust Region Policy Optimization”, 2015; arXiv:1502.05477.

Hado van Hasselt, Arthur Guez: “Deep Reinforcement Learning with Double Q-learning”, 2015; arXiv:1509.06461.

Ziyu Wang, Tom Schaul, Matteo Hessel, Hado van Hasselt, Marc Lanctot: “Dueling Network Architectures for Deep Reinforcement Learning”, 2015; arXiv:1511.06581.

Andrew Ng: “Deep Learning Specialization”. Coursera.org.

Pieter Abbeel: “Lecture 4a Policy Gradients and Actor Critic”. 2017; Berkeley Deep RL Bootcamp.

David Silver: “Lecture 9: Exploration and Exploitation”, 2015; RL course by David Silver.

Arthur Juliani: “Simple Reinforcement Learning with Tensorflow Part 4: Deep Q-Networks and Beyond”, 2016; Medium.com.

Matthew Hausknecht: “Deep Recurrent Q-Learning for Partially Observable MDPs”, 2015; arXiv:1507.06527.

Tom Schaul, John Quan, Ioannis Antonoglou: “Prioritized Experience Replay”, 2015; arXiv:1511.05952.

Volodymyr Mnih, Adrià Puigdomènech Badia, Mehdi Mirza, Alex Graves, Timothy P. Lillicrap, Tim Harley, David Silver: “Asynchronous Methods for Deep Reinforcement Learning”, 2016, ICML 2016; arXiv:1602.01783.

Matteo Hessel, Joseph Modayil, Hado van Hasselt, Tom Schaul, Georg Ostrovski, Will Dabney, Dan Horgan, Bilal Piot, Mohammad Azar: “Rainbow: Combining Improvements in Deep Reinforcement Learning”, 2017; arXiv:1710.02298.

John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford: “Proximal Policy Optimization Algorithms”, 2017; arXiv:1707.06347.

Kenneth Stanley and Risto Miikkulainen: “Evolving Neural Networks Through Augmenting Topologies”, 2002; *Evolutionary Computation*, 10(2):99-127, 2002.

Horia Mania, Aurelia Guy: “Simple random search provides a competitive approach to reinforcement learning”, 2018; arXiv:1803.07055.

Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra: “Playing Atari with Deep Reinforcement Learning”, 2013; arXiv:1312.5602.

Rein Houthooft, Richard Y. Chen, Phillip Isola, Bradly C. Stadie, Filip Wolski, Jonathan Ho: “Evolved Policy Gradients”, 2018; arXiv:1802.04821.

Prafulla Dhariwal, Christopher Hesse, Oleg Klimov, Alex Nichol, Matthias Plappert, Alec Radford, John Schulman, Szymon Sidor, Yuhuai Wu: “OpenAI Baselines”, 2017; GitHub.

Greg Brockman, Vicki Cheung, Ludwig Pettersson, Jonas Schneider, John Schulman, Jie Tang: “OpenAI Gym”, 2016; arXiv:1606.01540.