9 more stable value-based methods

In this chapter

- You improve on the methods you learned in the previous chapter by making them more stable and therefore less prone to divergence.
- You explore advanced value-based deep reinforcement learning methods, and the many components that make value-based methods better.
- You solve the cart-pole environment in a fewer number of samples, and with more reliable and consistent results.

> "Let thy step be slow and steady, that thou stumble not."
>
> — Tokugawa Ieyasu
> Founder and first shōgun of the Tokugawa shogunate of Japan and one of the three unifiers of Japan.

©Manning Publications Co. To comment go to liveBook
https://forums.manning.com/forums/grokking-deep-reinforcement-learning

WOW! eBook
www.wowebook.org

In the last chapter, you learned about value-based deep reinforcement learning. NFQ, the algorithm we developed, is a simple solution to the two most common issues value-based methods face: first, the issue that data in RL is not independent and identically distributed. It is probably the exact opposite. The experiences are dependent on the policy that generates them. And, they are not identically distributed since the policy changes throughout the training process. Second, the targets we use are not stationary, either. Optimization methods require fixed targets for robust performance. In supervised learning, this is easy to see. We have a dataset with pre-made labels as constants, and our optimization method uses these fixed targets for stochastically approximating the underlying data-generating function. In RL, on the other hand, targets such as the TD target, use the reward, and the discounted predicted return from the landing state as a target. But this predicted return comes from the network we are optimizing, which changes every time we execute the optimization steps. This issue creates a moving target that creates instabilities in the training process.

The way NFQ addresses these issues is through the use of batch. By growing a batch, we have the opportunity of optimizing several samples at the same time. The larger the batch, the more the opportunity for collecting a diverse set of experience samples. This somewhat addresses the IID assumption. NFQ addresses the stationarity of target requirements by using the same mini-batch in multiple sequential optimization steps. Remember that in NFQ, every E episodes, we "fit" the neural network to the same mini-batch K times. That K in there allows the optimization method to move toward the target more stably. Gathering a batch, and fitting the model for multiple iterations is similar to the way we train supervised learning methods, in which we gather a dataset and train for multiple epochs.

NFQ does OK job, but we can do better. Now that we know the issues, we can address them using better techniques. In this chapter, we explore algorithms that address not only these issues, but other issues that you learn about making value-based methods more stable.

DQN: Making reinforcement learning more like supervised learning

The first algorithm that we discuss in this chapter is called **Deep Q-Network** (DQN). DQN is one of the most popular DRL algorithms because it started a series of research innovations that mark the history of RL. DQN claimed for the first time super-human level performance on an ATARI benchmark in which agents learned from raw pixel data, from mere images.

Throughout the year, there have been many improvements proposed to DQN. And while these days, DQN in its original form is not a go-to algorithm, with the improvements, many of which you learn about in this book, the algorithm still has a spot among the best performing DRL agents.

©Manning Publications Co. To comment go to liveBook
https://forums.manning.com/forums/grokking-deep-reinforcement-learning

WOW! eBook
www.wowebook.org

Common problems in value-based deep reinforcement learning

We must be clear and understand the two most common problems that consistently show up in value-based deep reinforcement learning: the violations of the IID assumption, and the stationary of targets.

In supervised learning, we obtain a full dataset in advance. We pre-process it, shuffle it, and then split it into sets for training. One crucial step in this process is the shuffling of the dataset. By doing so, we allow our optimization method to avoid developing overfitting biases, to reduce the variance of the training process and speed up convergence, and overall learn a more general representation of the underlying data-generating process. In reinforcement learning, unfortunately, data is often gathered online, which as a result, the experience sample generated at time step t+1 correlates with the experience sample generated at time step t. Moreover, as the policy is to improve, and it changes the underlying data-generating process changes, too, which means that new data is locally correlated and not evenly distributed.

Boil It Down

Data is not Independent and Identically distributed (IID)

The second problem is the non-compliance with the IID assumption of the data. Optimization methods have been developed with the assumption that samples in the dataset we train with are independent and identically distributed.

We know, however, our samples are not independent, but instead, they come from a sequence, a time series, a trajectory. The sample at time step *t+1*, is dependent on sample at time step *t*. Samples are correlated and we can't prevent that from happening, it is a natural consequence of online learning.

But samples are also not identically distributed as they depend on the policy that generates the actions. We know the policy is changing through time, and for us that's a good thing. We want policies to improve. But that also means the distribution of samples (state-action pairs visited) will change as we keep improving.

Also, in supervised learning, the targets used for training are fixed values on your dataset; they are fixed throughout the training process. In reinforcement learning in general, and even more so in the extreme case of online learning, targets move with every training step of the network. At every training update step, we optimize the approximate value function and therefore change the shape of the function, that is, of possibly the entire value function. Changing the value function means that the target values change as well. Which in turn

©Manning Publications Co. To comment go to liveBook
https://forums.manning.com/forums/grokking-deep-reinforcement-learning

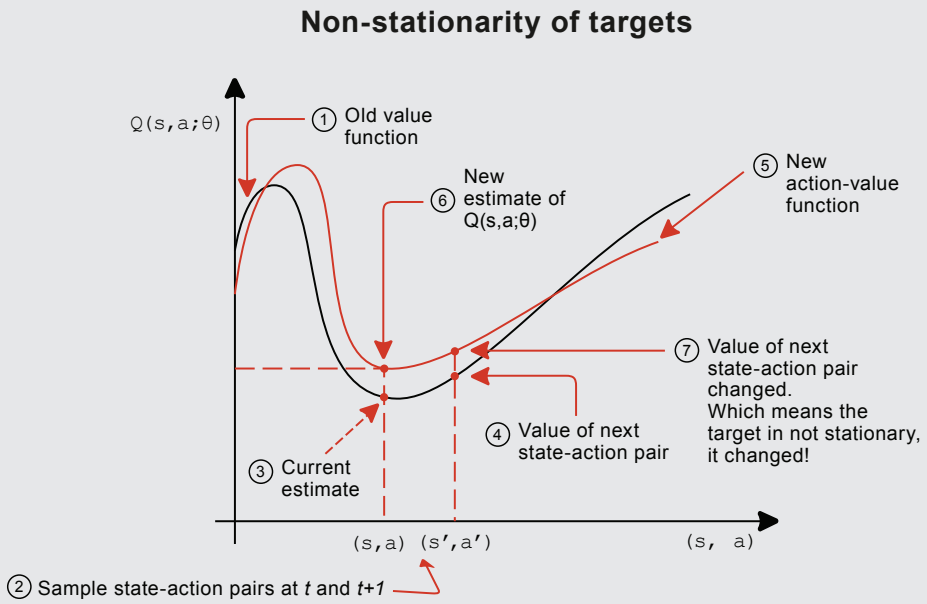
WOW! eBook
www.wowebook.org

means, the targets used are no longer valid. Even more, because the target come from the network, even before we use them, we can assume targets are invalid or biased at a minimum.

Boil It Down

Non-stationarity of targets

The problem of the non-stationarity of the targets is depicted. These are the targets we use to train our network, but these targets are calculated using the network itself.



As a result, the function changes with every update, changing in turn the targets.

In NFQ, we lessen this problem by using a batch and fitting the network to a small fixed dataset for multiple iterations. In NFQ, we collect a small dataset, calculating targets, optimize the network several times before going out to collect more samples. By doing this on a large batch of samples, the updates to the neural network are composed of many points across the function, additionally making changes even more stable.

DQN is an algorithm that addresses the question: How do we make reinforcement learning look more like supervised learning? Consider this question for a second, and think about the tweaks you would make to make the data look IID and the targets fixed.

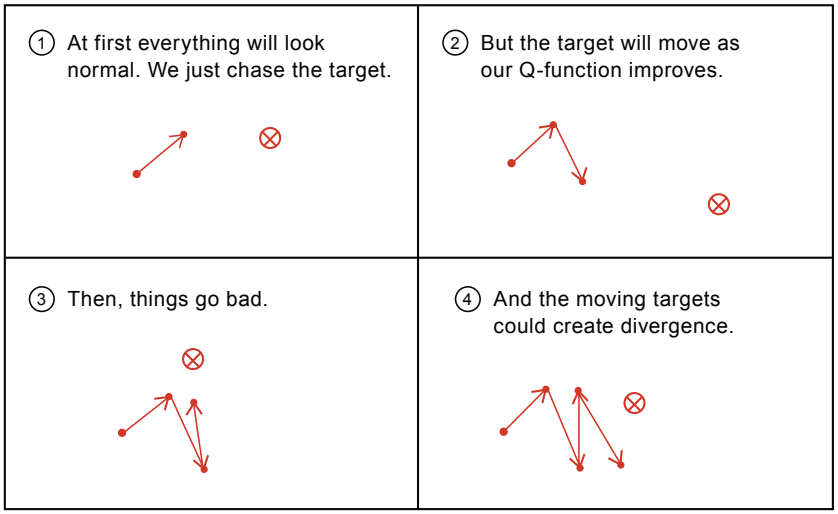
©Manning Publications Co. To comment go to liveBook
https://forums.manning.com/forums/grokking-deep-reinforcement-learning

WOW! eBook
www.wowebook.org

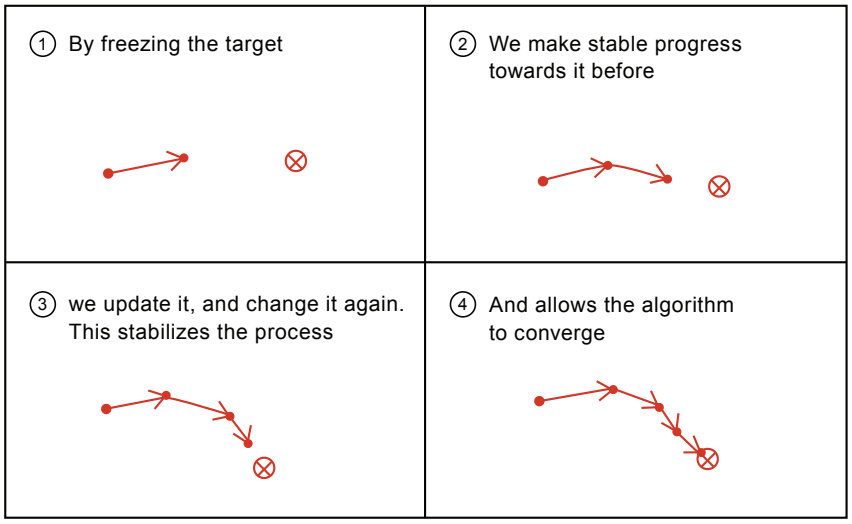
Using target networks

A very straightforward way to make target values more stationary is to have a separate network that we can fix for multiple steps and reserve it for calculating more stationary targets. The network with this purpose in DQN is called the target network.

Q-function optimization without a target network



Q-function approximation with a target network



©Manning Publications Co. To comment go to liveBook
https://forums.manning.com/forums/grokking-deep-reinforcement-learning

WOW! eBook
www.wowebook.org

By using a target network to fix targets, we mitigate the issue of "chasing your own tail" by artificially creating several small supervised learning problems presented sequentially to the agent. Our targets are fixed for as many steps as we fix our target network. This improves our chances of convergence, not to the optimal values because such things don't exist with non-linear function approximation, but convergence in general. But, more importantly, it substantially reduces the chances of divergence, which are not uncommon in value-based deep reinforcement learning methods.

Show Me the Math

Target network gradient update

$$\nabla_{\theta_i} L_i(\theta_i) = \mathbb{E}_{s,a,r,s'}\left[\left(r + \gamma \max_{a'} Q(s', a'; \theta_i) - Q(s, a; \theta_i)\right)\nabla_{\theta_i} Q(s, a; \theta_i)\right]$$

(1) The only difference between these two equations is the age of the neural network weights.

$$\nabla_{\theta_i} L_i(\theta_i) = \mathbb{E}_{s,a,r,s'}\left[\left(r + \gamma \max_{a'} Q(s', a'; \theta^-) - Q(s, a; \theta_i)\right)\nabla_{\theta_i} Q(s, a; \theta_i)\right]$$

(2) A target network is an previous instance of the neural network that we freeze for a number of steps. The gradient update has now time to catch up to the target, which is much more stable when froze. This adds stability to the updates.

It is important to note that in practice, we don't have two "networks," but instead, we have two instances of the neural network weights. We use the same model architecture and frequently update the weights of the target network to match the weights of the online network, which is the network we optimize on every step. "Frequently" here means something different depending on the problem, unfortunately. It is common to freeze these target network weights for 10 to 10,000 steps at a time, again depending on the problem (that's time steps, not episodes. Be careful there!). If you are using a convolutional neural network, such as what you'd use for learning in ATARI games, then a 10,000-step frequency is the norm. But for more straightforward problems such as the cart-pole environment, 10-20 steps is more appropriate.

By using target networks, we prevent the training process from spiraling around because we are fixing the targets for multiple time steps, thus allowing the online network weights to move consistently towards the targets before an update changes the optimization problem, and a new one is set. By using target networks, we stabilize training, but we also slow down learning because you are no longer training on up-to-date values; the frozen weights of the target network can be lagging for up-to 10,000 steps at a time. It's is essential to balance stability and speed and tune this hyperparameter.

©Manning Publications Co. To comment go to liveBook
https://forums.manning.com/forums/grokking-deep-reinforcement-learning

WOW! eBook
www.wowebook.org

I Speak Python

Use of the target and online networks in DQN

```
def optimize_model(self, experiences):
    states, actions, rewards, \
        next_states, is_terminals = experiences
    batch_size = len(is_terminals)
```

(1) Notice how we now query a target network to get the estimate of the next state.

```
    q_sp = self.target_model(next_states).detach()
```

(2) We grab the maximum of those values, and make sure to treat terminal states appropriately.

```
    max_a_q_sp = q_sp.max(1)[0].unsqueeze(1)
    max_a_q_sp *= (1 - is_terminals)
```

(3) Finally, we create the TD targets.

```
    target_q_sa = rewards + self.gamma * max_a_q_sp
```

(4) Query the current "online" estimate.

```
    q_sa = self.online_model(states).gather(1, actions)
```

(5) Use those values to create the errors.

```
    td_error = q_sa - target_q_sa
    value_loss = td_error.pow(2).mul(0.5).mean()
    self.value_optimizer.zero_grad()
    value_loss.backward()
    self.value_optimizer.step()
```

(6) Calculate the loss, and optimize the online network.

```
def interaction_step(self, state, env):
    action = self.training_strategy.select_action(
                        self.online_model, state)
```

(7) Notice how we use the online model for selecting actions.

```
    new_state, reward, is_terminal, _ = env.step(action)
    <...>
    return new_state, is_terminal
```

(8) This is how the target network (lagging network) gets updated with the online network (up-to-date network).

```
def update_network(self):
    for target, online in zip(
                        self.target_model.parameters(),
                        self.online_model.parameters()):
        target.data.copy_(online.data)
```

©Manning Publications Co. To comment go to liveBook
https://forums.manning.com/forums/grokking-deep-reinforcement-learning

WOW! eBook
www.wowebook.org

Using larger networks

Another way you can lessen the non-stationarity issue, to some degree, is to use larger networks. With more powerful networks, subtle differences between states are more likely detected. Larger networks reduce the aliasing of state-action pairs; the more powerful the network, the lower the aliasing, the lower the aliasing, the less apparent correlation between consecutive samples. And all of this can make target values and current estimates look more independent of each other.

By "aliasing" here I refer to the fact that two states can look like the same (or very similar) state to the neural network, but still possibly require different actions. State aliasing can occur when networks lack representational power. After all, neural networks are trying to find similarities to generalize; their job is to find these similarities. But, too small of a network and the generalization can go wrong. The network could get fixated with simple, easy to find patterns.

One of the motivations for using a target network is that they allow you to differentiate between correlated states more easily. Using a more capable network helps your network learn subtle differences, too.

But, a more powerful neural network takes longer to train. It needs not only more data (interaction time) but also more compute (processing time). Using a target network is a more robust approach to mitigating the non-stationary problem, but I want you to know all the tricks. It is favorable for you to know how these two properties of your agent (the size of your networks, and the use of target networks, along with the update frequency), interact and affect final performance in similar ways.

Boil It Down

Ways to mitigate the fact that targets in reinforcement learning are non-stationary

Allow me to restate that to mitigate the non-stationarity issue we can:

1. Create a target network that provides us with a temporarily stationary target value.
2. Create large-enough networks so that they can "see" the small differences between similar states (like those temporally correlated).

Now, target networks work and work well, have been proven to work multiple times. The technique of "Larger networks" is more of a hand-wavy solution than something scientifically proven to work every time. Though, feel free to experiment with this chapter's Notebook. You'll find it very easy to change values and test hypotheses.

©Manning Publications Co. To comment go to liveBook
https://forums.manning.com/forums/grokking-deep-reinforcement-learning

WOW! eBook
www.wowebook.org

Using experience replay

In our NFQ experiments, we use a mini-batch of 1,024 samples, and train with it for 40 iterations, alternating between calculating new targets and optimizing the network. These 1,024 samples are temporally correlated since most of them belong to the same trajectory since the maximum number of steps in a cart-pole episode is 500. One way to improve on this is to use a technique called experience replay. Experience replay consists of a data structure, often referred to as a replay buffer or a replay memory, that holds experience samples for several steps (much more than 1,024 steps), allowing the sampling of mini-batches from a broad set of past experiences. Having a replay buffer allows the agent two critical things. First, the training process can use a more diverse mini-batch for performing updates. Second, the agent no longer has to fit the model to the same small mini-batch for multiple iterations. Adequately sampling a sufficiently large replay buffer yields a slow-moving target, so the agent can now sample and train on every time step with a lower risk of divergence.

0001 A Bit Of History

Introduction of experience replay

Experience replay was introduced by Long-Ji Lin on a paper titled "Self-Improving Reactive Agents Based On Reinforcement Learning, Planning and Teaching", believe it or not, published in *1992*!

That's right, 1992! Again, that's when neural networks were referred to as "connectionism"... Sad times!

After getting his Ph.D. from CMU, Dr. Lin has moved through several technical roles in many different companies. Currently, he's the Chief Scientist at Signifyd, leading a team that works on a system to predict and prevent online fraud.

There are multiple benefits to using experience replay. By sampling at random, we increase the probability that our updates to the neural network have low variance. When we use the batch in NFQ, most of the samples in that batch were correlated and similar. Updating with similar samples concentrates the changes on a limited area of the function, and that potentially over-emphasizes the magnitude of the updates. If we sample uniformly at random from a substantial buffer, on the other hand, chances are, our updates to the network are better distributed all across, and therefore more representative of the true value function.

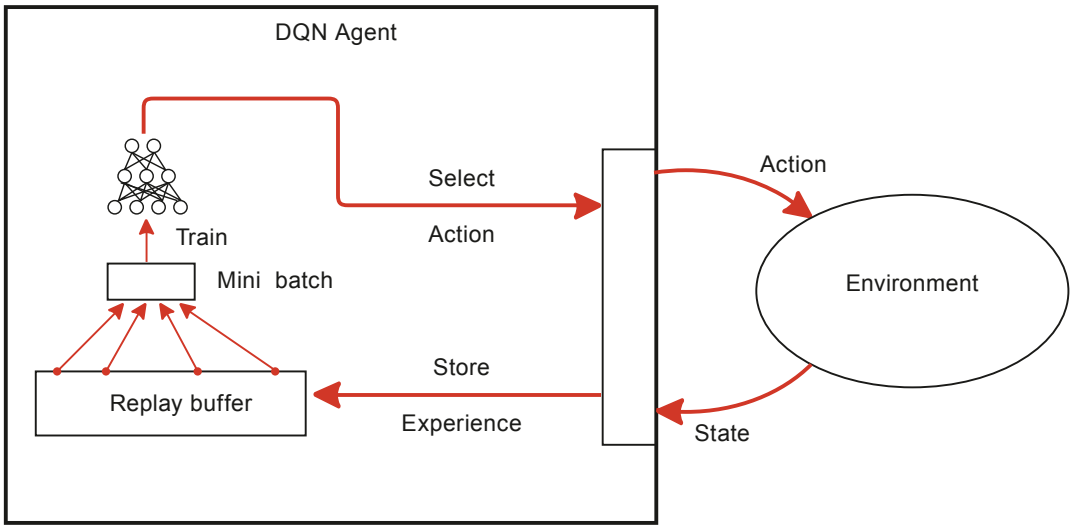
©Manning Publications Co. To comment go to liveBook
https://forums.manning.com/forums/grokking-deep-reinforcement-learning

WOW! eBook
www.wowebook.org

Using a replay buffer also gives the impression our data is IID so that the optimization method is stable. Samples appear independent and identically distributed because of the sampling from multiple trajectories and even policies at once.

By storing experiences and later sampling them uniformly, we make the data entering the optimization method look independent and identically distributed. In practice, the replay buffer needs to have a considerable capacity to perform optimally, from 10,000 to 1,000,000 experiences depending on the problem. Once you hit the maximum size, you evict the oldest experience before inserting the new one.

DQN with Replay Buffer



Unfortunately, the implementation becomes a little bit of a challenge when working with high-dimensional observations, because poorly implemented replay buffers hit a hardware memory limit quickly in high-dimensional environments. In image-based environments, for instance, where each state representation is a stack of the 4 latest image frames, as it is common for ATARI games, you probably don't have enough memory on your personal computer to naively store 1,000,000 experience samples. For the cart-pole environment, this is not much of a problem. First, we don't need 1,000,000 samples, and we use a buffer of size 50,000 instead. But also, states are represented by 4-element vectors, so there is not much of an implementation performance challenge.

©Manning Publications Co. To comment go to liveBook
https://forums.manning.com/forums/grokking-deep-reinforcement-learning

WOW! eBook
www.wowebook.org

Show Me the Math

Replay buffer gradient update

$$\nabla_{\theta_i} L_i(\theta_i) = \mathbb{E}_{s,a,r,s'}\left[\left(r + \gamma \max_{a'} Q(s', a'; \theta^-) - Q(s, a; \theta_i)\right)\nabla_{\theta_i} Q(s, a; \theta_i)\right]$$

(1) The only difference between these two equations is that we are now obtaining the experiences we use for training by sampling uniformly at random the replay buffer D, instead of using the online experiences as before.

$$\nabla_{\theta_i} L_i(\theta_i) = \mathbb{E}_{(s,a,r,s')\sim\mathcal{U}(\mathcal{D})}\left[\left(r + \gamma \max_{a'} Q(s', a'; \theta^-) - Q(s, a; \theta_i)\right)\nabla_{\theta_i} Q(s, a; \theta_i)\right]$$

(2) This is the full gradient update for DQN. More precisely the one referred to as Nature DQN, which is DQN with a target network and a replay buffer.

Nevertheless, by using a replay buffer, your data looks more IID and targets stationary than in reality. By training from uniformly sampled mini-batches, you make the RL experiences gathered online look more like a traditional supervised learning dataset with IID data and fixed targets. Sure, data is still changing as you add new and discard old samples, but these changes are happening slowly, and so they go somewhat unnoticed by the neural network and optimizer.

Boil It Down

Experience replay makes the data look IID, and targets somewhat stationary

The best solution to the problem of data not being IID is called experience replay.

The technique is very simple and it's been around for decades: As your agent collects experiences tuples $e_t=(S_t,A_t,R_{t+1},S_{t+1})$ online, we insert them into a data structure, commonly referred to as the *replay buffer D*, such that $D=\{e_1, e_2, \ldots, e_M\}$. M, the size of the replay buffer, is a value often between 10,000 to 1,000,000, depending on the problem.

We then train the agent on mini-batches sampled, usually uniformly at random, from the buffer, so that each sample has equal probability of being selected. Though, as you learn on the next chapter, you could possibly sample with some other distribution. Just beware, it is not that straightforward, we'll discuss details in the next chapter.

©Manning Publications Co. To comment go to liveBook
https://forums.manning.com/forums/grokking-deep-reinforcement-learning

WOW! eBook
www.wowebook.org

I Speak Python

A simple replay buffer

```
class ReplayBuffer():
    def __init__(self,
                 m_size=50000,
                 batch_size=64):
        self.ss_mem = np.empty(shape=(m_size), dtype=np.ndarray)
        self.as_mem = np.empty(shape=(m_size), dtype=np.ndarray)
        <...>
        self.m_size, self.batch_size = m_size, batch_size
        self._idx, self.size = 0, 0

    def store(self, sample):
        s, a, r, p, d = sample
        self.ss_mem[self._idx] = s
        self.as_mem[self._idx] = a
        <...>
        self._idx += 1
        self._idx = self._idx % self.m_size

        self.size += 1
        self.size = min(self.size, self.m_size)

    def sample(self, batch_size=None):
        if batch_size == None:
            batch_size = self.batch_size
        idxs = np.random.choice(
            self.size, batch_size, replace=False)
        experiences = np.vstack(self.ss_mem[idxs]), \
                      np.vstack(self.as_mem[idxs]), \
                      np.vstack(self.rs_mem[idxs]), \
                      np.vstack(self.ps_mem[idxs]), \
                      np.vstack(self.ds_mem[idxs])
        return experiences

    def __len__(self):
        return self.size
```

(1) This is a simple replay buffer with a default maximum size of 50,000, and a default batch size of 64 samples.

(2) We initialize 5 arrays to hold states, actions, reward, next states and done flags. Shorten for brevity.

(3) We initialize several variables to do storage and sampling.

(4) When we store a new sample, we begin by unwrapping the sample variable, and then setting each array's element to its corresponding value.

(5) Again removed for brevity.

(6) _idx points to the next index to modify, so we increase it, and also make sure it loops back after reaching the maximum size (the end of the buffer).

(7) Size also increases with every new sample stored, but it doesn't loop back to 0, it stops growing instead.

(8) In the sample function, we begin by determining the batch size. We use the default of 64 if nothing else was passed.

(9) Sample batch_size ids from 0 to size.

(10) Then, extract the experiences from the buffer using the sampled ids.

(11) And return those experiences.

(12) This is a handy function to return the correct size of the buffer when 'len(buffer)' is called.

©Manning Publications Co. To comment go to liveBook
https://forums.manning.com/forums/grokking-deep-reinforcement-learning

WOW! eBook
www.wowebook.org

Using other exploration strategies

Exploration is a vital component of reinforcement learning. In the NFQ algorithm, we use an epsilon-greedy exploration strategy, which consists of acting randomly with epsilon probability. We sample a number from a uniform distribution [0, 1). If the number is less than the hyperparameter constant, called epsilon, your agent selects an action uniformly at random (that's including the greedy action), otherwise, it acts greedily.

For the DQN experiments, I added to chapter 9's Notebook some of the other exploration strategies introduced in chapter 4. I adapted them to use them with neural networks, and the are re-introduced next. Make sure to checkout all Notebooks and play around.

I Speak Python

Linearly decaying epsilon-greedy exploration strategy

```
class EGreedyLinearStrategy():
    <...>
    def _epsilon_update(self):
        self.epsilon = 1 - self.t / self.max_steps
        self.epsilon = (self.init_epsilon - self.min_epsilon) * \
                        self.epsilon + self.min_epsilon
        self.epsilon = np.clip(self.epsilon,
                               self.min_epsilon,
                               self.init_epsilon)
        self.t += 1
        return self.epsilon

    def select_action(self, model, state):
        self.exploratory_action = False
        with torch.no_grad():
            q_values = model(state).cpu().detach()
            q_values = q_values.data.numpy().squeeze()
        if np.random.rand() > self.epsilon:
            action = np.argmax(q_values)
        else:
            action = np.random.randint(len(q_values))
        self._epsilon_update()
        self.exploratory_action = action != np.argmax(q_values)
        return action
```

(1) In an linearly decaying epsilon-greedy strategy we start with a high epsilon value and decay its value in a linear fashion.

(2) We clip epsilon to be between the initial and the minimum value.

(3) This is a variable holding the number of times epsilon has been updated.

(4) In the 'select_action' method, we use a model and a state.

(5) For logging purposes, I always extract the q_values.

(6) We draw the random number from a uniform distribution and compare it to epsilon.

(7) If higher, we use the argmax of the q_values, otherwise a random action.

(8) Finally, we update epsilon, set a variable for logging purposes, and return the action selected.

©Manning Publications Co. To comment go to liveBook
https://forums.manning.com/forums/grokking-deep-reinforcement-learning

WOW! eBook
www.wowebook.org

I Speak Python

Exponentially decaying epsilon-greedy exploration strategy

```
class EGreedyExpStrategy():
    <...>

    def _epsilon_update(self):
        self.epsilon = max(self.min_epsilon,
                           self.decay_rate * self.epsilon)
        return self.epsilon

    # def _epsilon_update(self):
    #     self.decay_rate = 0.0001
    #     epsilon = self.init_epsilon * np.exp( \
    #                                 -self.decay_rate * self.t)
    #     epsilon = max(epsilon, self.min_epsilon)
    #     self.t += 1
    #     return epsilon

    def select_action(self, model, state):
        self.exploratory_action = False
        with torch.no_grad():
            q_values = model(state).cpu().detach()
            q_values = q_values.data.numpy().squeeze()
        if np.random.rand() > self.epsilon:
            action = np.argmax(q_values)
        else:
            action = np.random.randint(len(q_values))
        self._epsilon_update()
        self.exploratory_action = action != np.argmax(q_values)
        return action
```

(1) In the exponentially decaying strategy, the only difference is now epsilon is decaying in an exponential curve.

(2) This is yet another way to exponentially decay epsilon, this one actually uses the exponential function. The epsilon values will be pretty much the same, but the decay rate will have to be a different scale.

(3) This 'select_action' function is identical to the previous strategy. One thing I want to highlight is, I'm querying the q_values every time only because I'm collecting information to show to you. But if you care about performance, this is a bad idea. A faster implementation would only query the network after determining a greedy action is being called for.

(4) 'exploratory_action' here is a variable used to calculate the percentage of exploratory actions taken per episode. Only used for logging information.

©Manning Publications Co. To comment go to liveBook
https://forums.manning.com/forums/grokking-deep-reinforcement-learning

WOW! eBook
www.wowebook.org

I Speak Python

SoftMax exploration strategy

```
class SoftMaxStrategy():
    <...>
    def _update_temp(self):
        temp = 1 - self.t / (self.max_steps * self.explore_ratio)
        temp = (self.init_temp - self.min_temp) * \
               temp + self.min_temp
        temp = np.clip(temp, self.min_temp, self.init_temp)
        self.t += 1
        return temp
    def select_action(self, model, state):
        self.exploratory_action = False
        temp = self._update_temp()
        with torch.no_grad():
            q_values = model(state).cpu().detach()
            q_values = q_values.data.numpy().squeeze()
            scaled_qs = q_values/temp
            norm_qs = scaled_qs - scaled_qs.max()
            e = np.exp(norm_qs)
            probs = e / np.sum(e)
            assert np.isclose(probs.sum(), 1.0)
        action = np.random.choice(np.arange(len(probs)),
                                  size=1, p=probs)[0]
        self.exploratory_action = action != np.argmax(q_values)
        return action
```

(1) In the SoftMax strategy, we use a "temperature" parameter which, the closer the value to 0, the more pronounced the differences in the values will become, making action selection more "greedy". The temperature is decayed linearly.

(2) Here, after decaying the temperature linearly we clip its value to make sure it is in an acceptable range.

(3) Notice that in the SoftMax strategy we really have no chance of going without extracting the q_values from the model. After all, actions depend directly on the values.

(4) After extracting the values, we want to accentuate their differences (unless temp equals 1).

(5) We normalize them to avoid an overflow in the 'exp' operation below.

(6) Calculate the exponential.

(7) Finally, convert to probabilities.

(8) Finally, we use the probabilities to select an action. Notice how we pass the probs variable to the p function argument.

(9) And just as before: Was the action the greedy or exploratory?

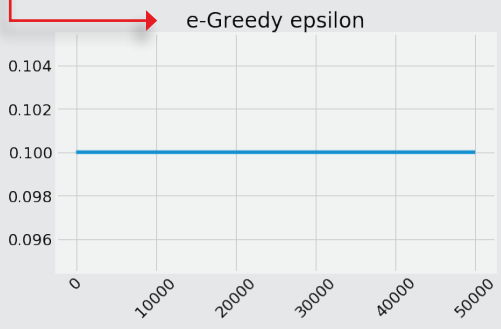
©Manning Publications Co. To comment go to liveBook
https://forums.manning.com/forums/grokking-deep-reinforcement-learning

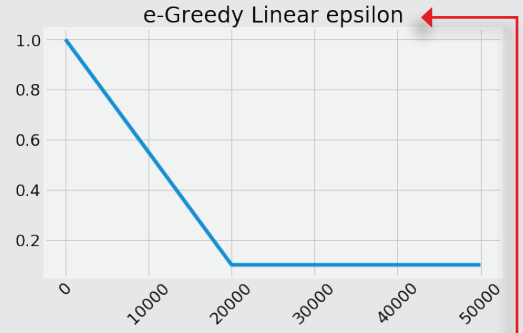
WOW! eBook
www.wowebook.org

It's In The Details

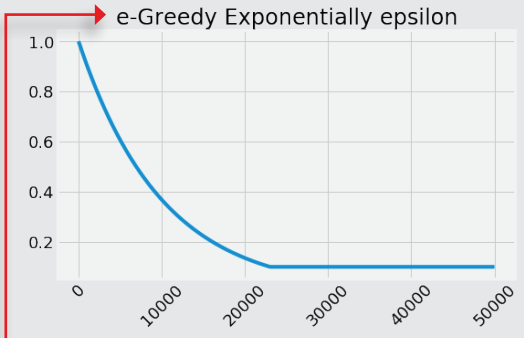
Exploration strategies have an impactful effect on performance

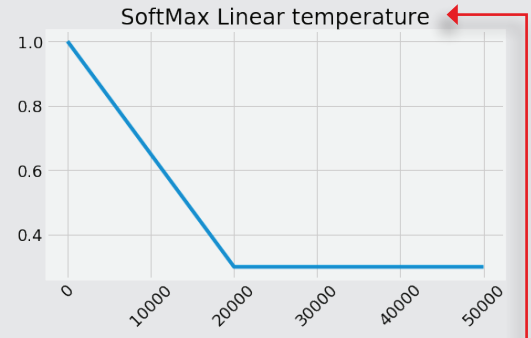
(1) In NFQ, we used epsilon greedy with a constant value of 0.5. Yes! That is 50% of the time we acted greedily, and 50% of the time, we chose uniformly at random. Given that there are only two actions in this environment, the actual probability of choosing the greedy action is 75%, and the chance of selecting the non-greedy action is 25%. Notice that in large action space, the probability of selecting the greedy action would be smaller. In the Notebook, I output this effective probability value under 'ex 100'. That means "ratio of exploratory action over the last 100 steps".





(2) In DQN and all remaining value-based algorithms in this and the following chapter, I use the exponentially decaying epsilon-greedy strategy. I prefer this one because it is simple and it works well. But other, more advanced strategies may be worth trying. I noticed even a small difference in hyperparameters makes a significant difference in performance. Make sure to test that yourself.





(3) The plots in this box are the decaying schedules of all the different exploration strategies available in chapter 9's Notebook. I highly encourage you to go through it and play with the many different hyperparameters and exploration strategies. There is a lot more to deep reinforcement learning than just the algorithms.

©Manning Publications Co. To comment go to liveBook
https://forums.manning.com/forums/grokking-deep-reinforcement-learning

WOW! eBook
www.wowebook.org

It's In The Details

The full Deep Q-Network (DQN) algorithm

Our DQN implementation has very similar components and settings to our NFQ, we:

- Approximate the action-value function $Q(s,a;\theta)$.
- Use a state-in-values-out architecture (nodes: 4, 512,128, 2).
- Optimize the action-value function to approximate the optimal action-value function $q^*(s,a)$.
- Use off-policy TD targets ($r + gamma*max\_a'Q(s',a';\theta)$) to evaluate policies.
- Use mean squared error (MSE) for our loss function.
- Use RMSprop as our optimizer with a learning rate of 0.0005.

Some of the differences are that in the DQN implementation we now:

- Use an exponentially decaying epsilon-greedy strategy to improve policies, decaying from 1.0 to 0.3 in roughly 20,000 steps.
- Use a replay buffer with 320 samples min, 50,000 max, and a mini-batches of 64.
- Use a target network that updates every 15 steps.

DQN has 3 main steps:

1. Collect experience: $(S_t, A_t, R_{t+1}, S_{t+1}, D_{t+1})$, and insert it into the replay buffer.
2. Randomly sample a mini-batch from the buffer and calculate the off-policy TD targets for the whole batch: $r + gamma*max\_a'Q(s',a';\theta)$.
3. Fit the action-value function $Q(s,a;\theta)$: Using MSE and RMSprop.

0001 A Bit Of History

Introduction of the DQN Algorithm

DQN was introduced in 2013 by Volodymyr "Vlad" Mnih in a paper called "Playing Atari with Deep Reinforcement Learning". This paper introduced DQN with experience replay. In 2015, another paper came out: "Human-level control through deep reinforcement learning". This second paper introduced DQN with the addition of target networks; the full DQN version you just learned about.

Vlad got his Ph.D. under Geoffrey Hinton (one of the fathers of deep learning), and works as a Research Scientist at Google DeepMind. He's been recognized for his DQN contributions, and has been included in the 2017 MIT Technology Review 35 Innovators under 35 list.

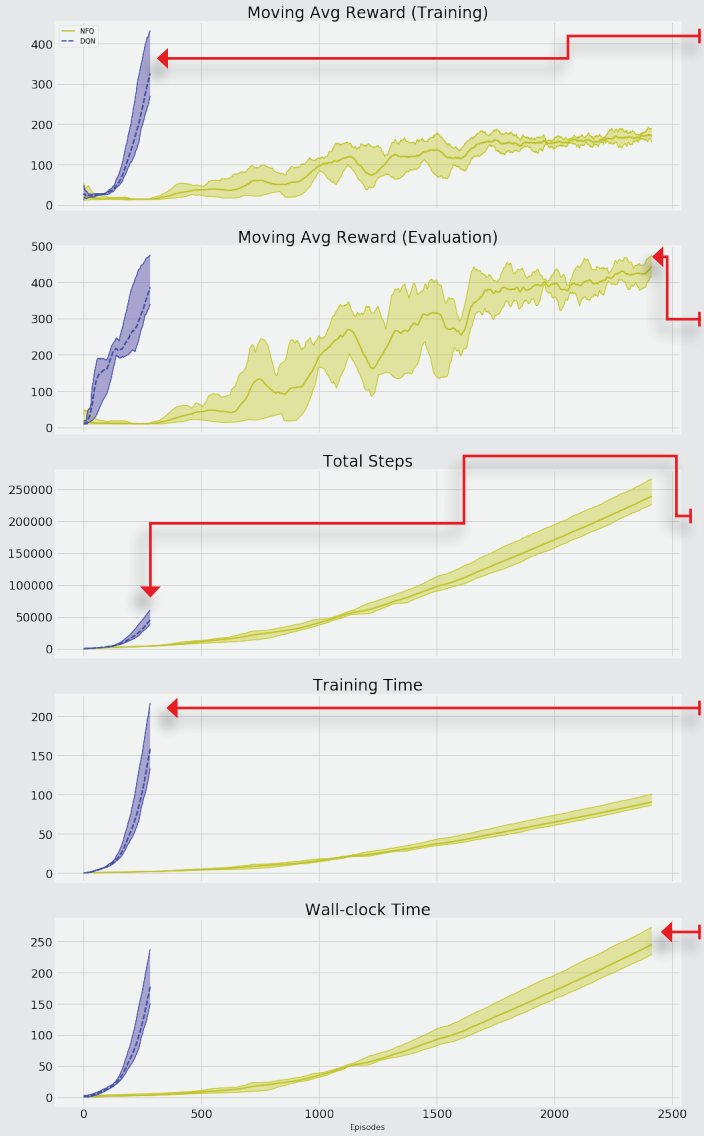
©Manning Publications Co. To comment go to liveBook
https://forums.manning.com/forums/grokking-deep-reinforcement-learning

WOW! eBook
www.wowebook.org

Tally It Up

DQN passes the cart-pole environment

The most remarkable part of the results is that NFQ needs far more samples than DQN to solve the environment; DQN is more sample efficient. However, they take about the same time, both training (compute) and wall-clock time.



(1) The most obvious conclusion we can draw from this first graph is the DQN is more sample efficient than NFQ. But, if you pay attention to the curves, you notice how NFQ is also noisier than DQN. This is one of the most important improvements we accomplished so far.

(2) As you can see, they both pass the cart-pole environment, but DQN takes about 250 episodes while NFQ takes almost 2,500 episodes. That's a tenfold reduction in samples.

(3) Here you can see the same trend in sample efficiency, but with time steps instead of episodes: DQN takes about 50,000 experience tuples while NFQ uses about 250,000.

(4) But, DQN takes more training time than NFQ to pass the environment. Now, by training time here I mean the time from the beginning to the end of all episodes, not just computation.

(5) In terms of wall-clock time (that is training time, and statistics calculation, evaluation steps, etc) they are both about 5 minutes.

©Manning Publications Co. To comment go to liveBook
https://forums.manning.com/forums/grokking-deep-reinforcement-learning

WOW! eBook
www.wowebook.org

Double DQN: Mitigating the overestimation of action-value functions

In this section, we introduce one of the main improvements that have proposed to DQN throughout the year, called Double Deep Q-Networks (Double DQN, or DDQN). This improvement consists of adding Double learning to our DQN agent. It's very straightforward to implement, and it yields agents with consistently better performance than DQN. The changes required are very similar to the changes applied to Q-learning to develop Double Q-learning; however, there are some differences that we need to discuss.

The problem of overestimation, take two

As you can probably remember from chapter 6, Q-learning tends to overestimate action-value functions. Our DQN agent is no different; we are using the same off-policy TD target after all with that max operator. The crux of the problem is very simple: We are taking the max of estimated values. Estimated values are often off-center, some higher than the true values, some lower, but the bottom line is they are off. Now, the problem is that we are always taking the max of these values. So, we have a preference for higher values, even if they are not correct. So our algorithms show a positive bias, and performance suffers.

Miguel's Analogy

The issue with over-optimistic agents, and people

I used to like super positive people until I learned about Double DQN. No, seriously, imagine you meet a very optimistic person, let's call her DQN. DQN is very optimistic. She's experienced many things in life, from the toughest defeat to the highest success. The problem with DQN, though, is she expects the sweetest possible outcome from every single thing she does, regardless of what she actually does. Is that a problem?

One day, DQN went to a local casino. It was the first time, but lucky DQN got the jackpot at the slot machines. Optimistic as she is, DQN immediately adjusted her value function. She thought, "Going to the casino is very rewarding (the value of Q(s,a) should be very high) because at the casino you can go to the slot machines (next state s') and by playing the slot machines, you get the jackpot [max_a' Q(s', a')]".

But, there are multiple issues with this thinking. To begin with, not every time DQN goes to the casino, she plays the slot machines. She likes to try new things too (she explores), and sometimes she tries the roulette, poker, or blackjack (tries a different action). Sometimes the slot machines area is under maintenance and not accessible (the environment transitions her somewhere else.) Additionally, most of the time DQN plays the slot machines, she doesn't get the jackpot (the environment is stochastic.) After all, slot machines are called bandits for a reason, not those bandits, the other – never mind.

©Manning Publications Co. To comment go to liveBook
https://forums.manning.com/forums/grokking-deep-reinforcement-learning

WOW! eBook
www.wowebook.org

Separating action selection and action evaluation

One way to better understand the positive bias and how we can address it when using function approximation is by unwrapping the *max* operator in the target calculations. The *max* of a Q-function is the same as the Q-function of the *argmax* action.

Refresh My Memory

What's an argmax, again?

The argmax function is defined as the arguments of the maxima. The argmax action-value function, argmax Q-function, "*$\text{argmax}_a Q(s,a)$*" is just the *index* of the action with the maximum value at the given state *s*.

So, for example, if you have a *Q(s)* with values *[-1, 0 , -4, -9]* for actions 0-3, the *$\max_a Q(s, a)$* is *0*, which is the maximum value, and the *$\text{argmax}_a Q(s, a)$* is *1* which is the index of the maximum value.

So, let's unpack the previous sentence with the max and argmax. Notice that we made pretty much the same changes when we went from Q-learning to Double Q-learning, but given we are using function approximation, we need to be cautious. At first, this unwrapping might seem like a silly step, but it actually helps me understand how to mitigate this problem.

Show Me the Math

Unwrapping the argmax

$$\nabla_{\theta_i} L_i(\theta_i) = \mathbb{E}_{(s,a,r,s')\sim\mathcal{U}(\mathcal{D})}\Big[\big(r + \gamma \max_{a'} Q(s', a'; \theta^-) - Q(s, a; \theta_i)\big)\nabla_{\theta_i} Q(s, a; \theta_i)\Big]$$

(1) What we are doing here is something silly. Take a look at the equations at the top and bottom of the box and compare them.

$$\max_{a'} Q(s', a'; \theta^-) \qquad Q(s', \operatorname{argmax}_{a'} Q(s', a'; \theta^-); \theta^-)$$

(2) There is no real difference between the two equations since both are using the same Q-values for the target. Bottom line is these two bits are the same thing written differently.

$$\nabla_{\theta_i} L_i(\theta_i) = \mathbb{E}_{(s,a,r,s')\sim\mathcal{U}(\mathcal{D})}\Big[\big(r + \gamma Q(s', \operatorname{argmax}_{a'} Q(s', a'; \theta^-); \theta^-) - Q(s, a; \theta_i)\big)\nabla_{\theta_i} Q(s, a; \theta_i)\Big]$$

©Manning Publications Co. To comment go to liveBook
https://forums.manning.com/forums/grokking-deep-reinforcement-learning

WOW! eBook
www.wowebook.org

I SPEAK PYTHON

Unwrapping the max in DQN

(1) This is the original DQN-way of calculating targets.

```
q_sp = self.target_model(next_states).detach()
max_a_q_sp = q_sp.max(1)[0].unsqueeze(1)
```

(2) It's important that we 'detach' the target so that we do not back-propagate through it.

(3) We pull the q-values of the next state and get their max.

```
max_a_q_sp *= (1 - is_terminals)
target_q_sa = rewards + self.gamma * max_a_q_sp
```

(4) Set the value of terminal states to 0, and calculate the targets.

(5) This is an equivalent way to calculating targets, "unwrapping the max".

```
argmax_a_q_sp = self.target_model(next_states).max(1)[1]
```

(6) First, get the argmax action of the next state.

```
q_sp = self.target_model(next_states).detach()
```

(7) Then, get the q-values of the next state, just as before.

(8) Now, we use the indices to get the max values of the next states.

```
max_a_q_sp = q_sp[np.arange(batch_size), argmax_a_q_sp]
max_a_q_sp = max_a_q_sp.unsqueeze(1)
max_a_q_sp *= (1 - is_terminals)
target_q_sa = rewards + self.gamma * max_a_q_sp
```

(9) And proceed as before.

All we are saying here is that taking the *max* is like asking the network:

"What's the value of the highest-valued action in state s?"

But, we are really asking two questions with a single question. First, we do an *argmax*, which is equivalent to asking:

"Which action is the highest-valued action in state s?"

And then, we use that action to get its value. Equivalent to asking:

"What's the value of this action (which happens to be the highest-valued action) in state s?"

One of the problems is that we are asking both questions to the same Q-function, which shows bias in the same direction in both answers.

In other words, the function approximator will answer:

"I think this one is the highest-valued action in state s, and this is its value."

©Manning Publications Co. To comment go to liveBook
https://forums.manning.com/forums/grokking-deep-reinforcement-learning

WOW! eBook
www.wowebook.org

A solution

A way to reduce the chance of positive bias is to have two instances of the action-value function, just like we did in chapter 6.

If you had another source of the estimates, you could then ask one of the questions to one and the other question to the other. It's somewhat like taking votes, or like an "I cut, you choose first" procedure, or just like getting a second doctor's opinion on health matters.

In double learning, one estimator selects the index of what it believes to be the highest-valued action, and the other estimator gives the value of this action.

Refresh My Memory

Double learning procedure

We did this procedure with tabular reinforcement learning in Chapter 6 under the Double Q-learning algorithm. It goes like this:

You create two action-value functions, Q_A and Q_B.

You flip a coin to decide which action-value function to update. E.g.: Q_A on heads, Q_B on tails.

If you got a heads and thus get to update Q_A: You select the action *index* to evaluate from Q_B, and *evaluate* it using the estimate Q_A predicts. Then, you proceed to update Q_A as usual, and leave Q_B alone.

If you got a tails and thus get to update Q_B, you do it the other way around: Get the index from Q_A, and get the value estimate from Q_B. Q_B gets updated, and Q_A is left alone.

However, implementing this double learning procedure exactly as described when using function approximation (for DQN) creates unnecessary overhead. If we did so, we would end-up with four networks: two networks for training (Q_A, Q_B) and two target networks, one for each online network.

Additionally, it creates a slowdown in the training process, since we would be training only one of these networks at a time. Therefore, only one network would improve per step. This is certainly a waste.

Doing this double learning procedure with function approximators may still be better than not doing it at all, despite the extra overhead. Fortunately for us, there is a simple modification to the original double learning procedure that adapts it to DQN and give us substantial improvements without the extra overhead.

©Manning Publications Co. To comment go to liveBook
https://forums.manning.com/forums/grokking-deep-reinforcement-learning

WOW! eBook
www.wowebook.org

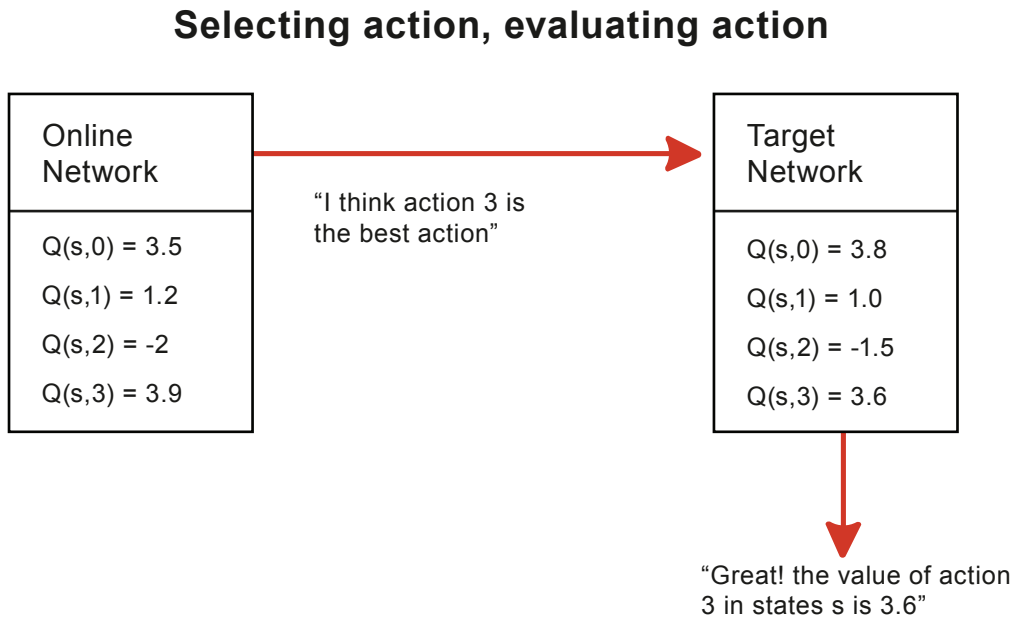
A more practical solution

Instead of adding this overhead that is a detriment to training speed, we can perform double learning with the other network we already have, which is the target network.

However, instead of training both the online and target networks, we continue training only the online network, but use the target network to help us, in a sense, cross-validate the estimates.

We want to be cautious as to which network to use for action selection and which network to use for action evaluation. Initially, we added the target network to stabilize training by preventing chasing a moving target. To continue on this path, we want to make sure we use the network we are training, the online network, for answering the first question. In other words, we use the online network to find the index of the best action. Then, use the target network to ask the second question, that is, to evaluate the previously selected action.

This is the ordering that works best in practice, and it makes sense why it works. By using the target network for value estimates, we make sure the target values are frozen as needed for stability. If we were to implement it the other way around, the values would come from the online network, which is getting updated at every time step, and therefore changing continuously.



©Manning Publications Co. To comment go to liveBook
https://forums.manning.com/forums/grokking-deep-reinforcement-learning

WOW! eBook
www.wowebook.org

0001 A Bit Of History

Introduction of the Double DQN Algorithm

Double DQN was introduced in 2015 by Hado van Hasselt, shortly after the release of the 2015 version of DQN (The 2015 version of DQN is sometimes referred to as 'Nature' DQN — because it was published in the Nature scientific journal, and sometimes as 'Vanilla' DQN — because it is the first of many other improvements over the years).

In 2010, Hado also authored the Double Q-learning algorithm (double learning for the tabular case), as an improvement to the Q-learning algorithm. This is the algorithm you learned about and implemented in chapter 6.

Double DQN, also referred to as DDQN, was the first of many improvements proposed over the years for DQN. Back in 2015 when it was first introduced, DDQN obtained state-of-the-art (best at the moment) results in the ATARI domain.

Hado obtained his Ph.D. from the University of Utrecht in the Netherlands in Artificial Intelligence (Reinforcement Learning). After a couple of years as a postdoctoral researcher, he got a job at Google DeepMind as a Research Scientist.

Show Me the Math

DDQN gradient update

(1) So far the gradient updates look as follows.

(2) We sample uniformly at random from the replay buffer a experience tuple (s, a, r, s').

$$\nabla_{\theta_i} L_i(\theta_i) = \mathbb{E}_{(s,a,r,s')\sim\mathcal{U}(\mathcal{D})}\Big[\big(r + \gamma Q(s', \underset{a'}{\operatorname{argmax}}\, Q(s', a'; \theta^-); \theta^-) - Q(s, a; \theta_i)\big)\nabla_{\theta_i} Q(s, a; \theta_i)\Big]$$

(3) We then calculate the TD target and error using the target network.

(4) Finally calculate the gradients only through the predicted values.

(1) The only difference in DDQN is now we use the online weights to select the action, but still use the frozen weights to get the estimate.

$$\nabla_{\theta_i} L_i(\theta_i) = \mathbb{E}_{(s,a,r,s')\sim\mathcal{U}(\mathcal{D})}\Big[\big(r + \gamma Q(s', \underset{a'}{\operatorname{argmax}}\, Q(s', a'; \theta_i); \theta^-) - Q(s, a; \theta_i)\big)\nabla_{\theta_i} Q(s, a; \theta_i)\Big]$$

©Manning Publications Co. To comment go to liveBook
https://forums.manning.com/forums/grokking-deep-reinforcement-learning

WOW! eBook
www.wowebook.org

I Speak Python

Double DQN

```
def optimize_model(self, experiences):
    states, actions, rewards, \
        next_states, is_terminals = experiences
    batch_size = len(is_terminals)
```

(1) In Double DQN, we use the online network to get the index of the highest-valued action of the next state, the 'argmax'.

```
    #argmax_a_q_sp = self.target_model(next_states).max(1)[1]
    argmax_a_q_sp = self.online_model(next_states).max(1)[1]
```

(2) Then, extract the q-values of the next state according to the target network.

```
    q_sp = self.target_model(next_states).detach()
```

(3) We then index the q-values provided by the target network with the action indices provided by the online network.

```
    max_a_q_sp = q_sp[np.arange(batch_size), argmax_a_q_sp]
```

(4) Then setup the targets as usual.

```
    max_a_q_sp = max_a_q_sp.unsqueeze(1)
    max_a_q_sp *= (1 - is_terminals)
    target_q_sa = rewards + (self.gamma * max_a_q_sp)
```

(5) Get the current estimates. Note this is where the gradients are flowing through.

```
    q_sa = self.online_model(states).gather(1, actions)
    td_error = q_sa - target_q_sa
    value_loss = td_error.pow(2).mul(0.5).mean()
    self.value_optimizer.zero_grad()
    value_loss.backward()
    self.value_optimizer.step()
```

(6) Calculate the loss, and step the optimizer.

```
def interaction_step(self, state, env):
    action = self.training_strategy.select_action(
                    self.online_model, state)
    new_state, reward, is_terminal, _ = env.step(action)
    return new_state, is_terminal
```

(7) Here we keep using the online network for action selection.

```
def update_network(self):
    for target, online in zip(
                        self.target_model.parameters(),
                        self.online_model.parameters()):
        target.data.copy_(online.data)
```

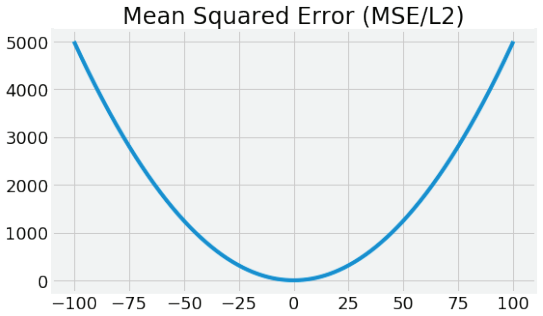
(8) Updating the target network is still the same as before.

©Manning Publications Co. To comment go to liveBook
https://forums.manning.com/forums/grokking-deep-reinforcement-learning
WOW! eBook
www.wowebook.org

A more forgiving loss function

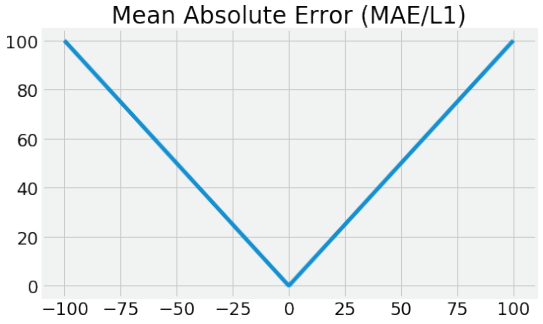
In the previous chapter, we selected the L2 loss, also known as Mean Square Error (MSE), as our loss function mostly for its widespread use and simplicity. And, in reality, in a problem such as the cart-pole environment, there might not be a good reason to look any further. However, because I'm teaching you the ins and outs of the algorithms and not just "how to hammer the nail," I'd also like to make you aware of the different knobs available so you can play around when tackling more challenging problems.

MSE is a ubiquitous loss function because it is simple, it makes sense, and it works well. But, one of the issues with using MSE for reinforcement learning is that it penalizes large errors more than small errors. This makes sense when doing supervised learning because our targets are the true value from the get-go, and are fixed throughout the training process. That means we are confident that, if the model is very wrong, then it should be penalized more heavily than if it is just wrong.



But as stated now several times, in reinforcement learning, we do not have these true values, and the values we use to train our network are dependent on the agent itself. That's a mind shift. Besides, targets are constantly changing; even when using target networks, they still change often. In reinforcement learning, being very wrong is something we expect and welcome. At the end of the day, if you think about it, we are not really "training" agents, our agents learn on their own. Think about that for a second.

A loss function not as unforgiving, and also more robust to outliers, is the Mean Absolute Error, also known as MAE or L1 loss. MAE is defined as the average absolute difference between the predicted and true values, that is, the predicted action-value function and the TD target. Given that MAE is a linear function as opposed to quadratic like MSE, we can expect MAE to be more successful at treating large errors the same way as small errors. This can come in handy in our case because we expect our action-value function to give wrong values at some point during training, particularly at the



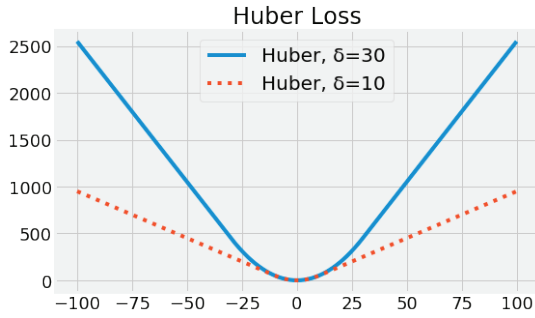
©Manning Publications Co. To comment go to liveBook
https://forums.manning.com/forums/grokking-deep-reinforcement-learning

WOW! eBook
www.wowebook.org

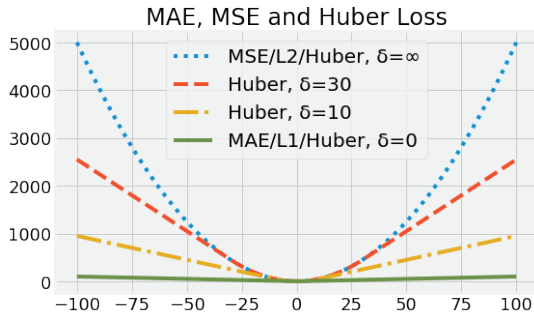
beginning. Being more resilient to outliers often implies errors have less effect, as compared to MSE, in terms of changes to our network, which means more stable learning.

Now, on the flip side, one of the helpful things of MSE that MAE does not have is the fact that its gradients decrease as the loss goes to zero. This feature is helpful for optimization methods as it makes it easier to reach the optima because lower gradients mean small changes to the network. But luckily for us, there is a loss function that is somewhat a mix of MSE and MAE, called the Huber loss.

The Huber loss has the same useful property as MSE of quadratically penalizing the errors near zero, but it is not quadratic all the way out for huge errors. Instead, the Huber loss is quadratic (curved) near-zero error, and it becomes linear (straight) for errors larger than a pre-set threshold. Having the best of both worlds makes the Huber loss robust to outliers, just like MAE, and differentiable at 0, just like MSE.



The Huber loss uses a hyperparameter, δ, to set this threshold in which the loss goes from quadratic to linear, basically, from MSE to MAE. If δ is zero, you are left precisely with MAE, and if δ is infinite, then you are left precisely with MSE. A typical value for δ is 1, but be aware that your loss function, optimization, and learning rate interaction in complex ways. So, if you change one, you may need to tune some of the others. Check out the Notebook for this chapter so you can play around.



Interestingly, there are at least two different ways of implementing the Huber loss function. You could either compute the Huber loss as defined, or compute the MSE loss instead, and then set all gradients larger than a threshold to a fixed magnitude value. You clip the magnitude of the gradients. The former depends on the deep learning framework you use, but the problem is, some frameworks don't give you access to the δ hyperparameter, so you are stuck with δ set to 1, which doesn't always work, and is not always the best. The latter often referred to as "loss clipping," or better yet "gradient clipping," is more flexible and, therefore, what I implement in the Notebook.

©Manning Publications Co. To comment go to liveBook
https://forums.manning.com/forums/grokking-deep-reinforcement-learning

WOW! eBook
www.wowebook.org

I Speak Python

Double DQN with Huber Loss

```
def optimize_model(self, experiences):
    states, actions, rewards, \
        next_states, is_terminals = experiences
    batch_size = len(is_terminals)
                    (1) First, you calculate the targets and get the
    <...>           current values just as before, using double learning.
    td_error = q_sa - target_q_sa
                    (2) Then, calculate the loss function as Mean
                    Squared Error, just as before.
    value_loss = td_error.pow(2).mul(0.5).mean()
                    (3) Zero the optimizer and calculate the
                    gradients in a backward step.
    self.value_optimizer.zero_grad()
    value_loss.backward()
                    (4) Now, clip the gradients to the max_gradient_norm, this
                    value can be virtually any value, but know that this interacts
                    with other hyperparameters, such as learning rate.
    torch.nn.utils.clip_grad_norm_(
                        self.online_model.parameters(),
                        self.max_gradient_norm)
                    (5) Finally, step the optimizer.
    self.value_optimizer.step()
```

Know that there is such a thing as "reward clipping," which is different than "gradient clipping." These are two very different things, so beware. One works on the rewards and the other on the errors (the loss). Now, above all is not to confuse either of these with "Q-value clipping," which is undoubtedly a mistake.

Remember, the goal in our case is to prevent gradients from becoming too large. For this, we either make the loss linear outside a given absolute TD error threshold or make the gradient constant outside a max gradient magnitude threshold.

In the cart-pole environment experiments that you find in the Notebook, I implemented the Huber loss function by using the "gradient clipping" technique: That is, I calculate MSE and then clip the gradients. However, as I mentioned before, I set the hyperparameter setting the maximum gradient values to infinity. Therefore, it is effectively using good-old MSE. But, please, experiment, play around, explore! The Notebooks I created should help you learn almost as much as the book. So, set yourself free over there.

©Manning Publications Co. To comment go to liveBook
https://forums.manning.com/forums/grokking-deep-reinforcement-learning

WOW! eBook
www.wowebook.org

It's In The Details

The full Double Deep Q-Network (DDQN) algorithm

DDQN is almost identical to DQN, but there are still some differences. We still:

- Approximate the action-value function *Q(s,a; θ)*.
- Use a state-in-values-out architecture (nodes: 4, 512,128, 2).
- Optimize the action-value function to approximate the optimal action-value function *q*(s,a)*.
- Use off-policy TD targets (*r + gamma*max_a'Q(s',a'; θ)*) to evaluate policies.

Notice that we now:

- Use an adjustable Huber loss, which since we set the 'max_gradient_norm' variable to 'float('inf')', we are effectively just using mean squared error (MSE) for our loss function.
- Use RMSprop as our optimizer with a learning rate of 0.0007. Note that before we used 0.0005 because without double learning (vanilla DQN) some seeds fail if we train with a learning rate of 0.0007. Perhaps stability? In DDQN, on the other hand, training with a higher learning rate works best.

In DDQN we are still using:

- An exponentially decaying epsilon-greedy strategy (from 1.0 to 0.3 in roughly 20,000 steps) to improve policies.
- A replay buffer with 320 samples min, 50,000 max, and a batch of 64.
- A target network that freezes for 15 steps and then updates fully.

DDQN, just like DQN has the same 3 main steps:

1. Collect experience: (S_t, A_t, R_{t+1}, S_{t+1}, D_{t+1}), and insert it into the replay buffer.
2. Randomly sample a mini-batch from the buffer and calculate the off-policy TD targets for the whole batch: *r + gamma*max_a'Q(s',a'; θ)*.
3. Fit the action-value function Q(s,a; θ): Using MSE and RMSprop.

The bottom line is the DDQN implementation and hyperparameters are *identical* to those of DQN, *except* that we now use double learning and therefore train with a slightly higher learning rate. The addition of the Huber loss does not change anything because we are "clipping" gradients to a max value of infinite, which is equivalent to using MSE. However, for many other environments you will find it useful, so tune this hyperparameter.

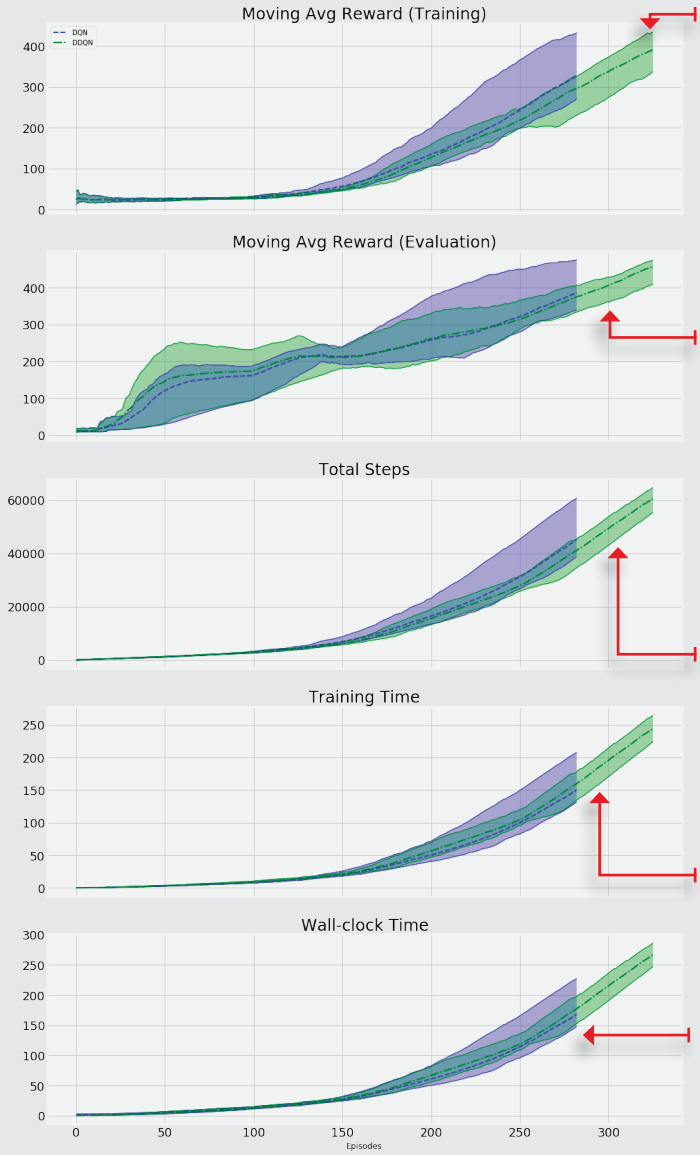
©Manning Publications Co. To comment go to liveBook
https://forums.manning.com/forums/grokking-deep-reinforcement-learning

WOW! eBook
www.wowebook.org

Tally It Up

DDQN is more stable than NFQ or DQN

DQN and DDQN have very similar performance in the cart-pole environment. However, this is a simple environment with a very smooth reward function. In reality, DDQN should always give better performance.



(1) Pay attention, not just to the mean lines in the middle, but to the top and bottom bounds representing the maximum and minimum values obtained by any of the 5 seeds during that episode. DDQN shows tighter bounds, basically, showing more stability on performance.

(2) In the second plot, you see the same pattern, DDQN has narrower bounds. In terms of performance, DQN reaches the max in less number of episodes on the cart-pole environment for a seed, but DDQN reaches the max in similar number of episodes across all seeds: Stability.

(3) DQN goes through more steps in fewer episodes in that "lucky" seed and arguably related to performance (remember the cart-pole environment is about "lasting").

(4) In terms of time, DDQN takes a bit longer than DQN to successfully pass the environment.

(5) For both, training and wall-clock time.

©Manning Publications Co. To comment go to liveBook
https://forums.manning.com/forums/grokking-deep-reinforcement-learning

WOW! eBook
www.wowebook.org

Things we can still improve on

Surely our current value-based deep reinforcement learning method is not perfect, but it is pretty solid. DDQN can reach super-human performance in many of the ATARI games. To replicate those results, you would have to change the network to take images as input (a stack of 4 images to be able to infer things such as direction and velocity from the images), and, of course, tune the hyperparameters.

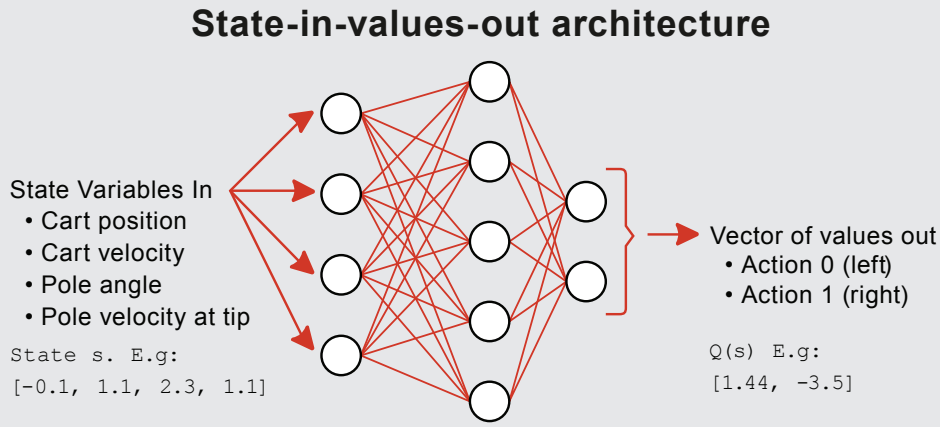
Yet, we can still go a little further. There are at least a couple of other improvements to consider that are easy to implement and impact performance in a very positive way.

The first improvement requires us to reconsider the current network architecture. As of right now, we have a very naive representation of the Q-function on our neural network architecture.

Refresh My Memory

Current neural network architecture

We are literately "making reinforcement learning look like supervised learning". But, we can, and should, break free from this constraint, and think out of the box.

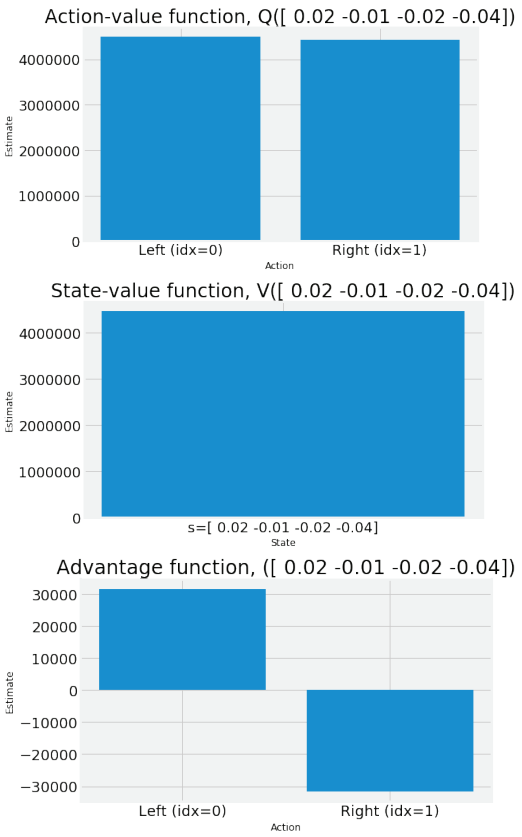


Is there any better way of representing the Q-function? Think about this for a second while you look at the images on the next page.

©Manning Publications Co. To comment go to liveBook
https://forums.manning.com/forums/grokking-deep-reinforcement-learning

WOW! eBook
www.wowebook.org

The images on the right are bar plots representing the estimated action-value function Q, state-value function V, and action-advantage function A for the cart-pole environment with a state in which the pole is near vertical.



Notice the different functions and values and start thinking about how to better architect the neural network so that data is used more efficiently. As a hint, let me remind you that the Q-values of a state are related through the V-function. That is, the action-value function Q has an essential relationship with the state-value function V, because of both actions in Q(s) and indexed by the same state s (in the example to the right s=[0.02, -0.01, -0.02, -0.04]).

The question is, would you be able to learn anything about Q(s, 0) if you are using a Q(s, 1) sample? Look at the plot showing the action-advantage function A(s) and notice how much easier it is for you to eyeball the greedy action with respect to these estimates than when using the plot with the action-value function Q(s). What can you do about this? In the next chapter, we look at a network architecture called the Dueling network that helps us exploit these relationships.

The other thing to consider improving is the way we sample experiences from the replay buffer. As of now, we pull samples from the buffer uniformly at random, and I'm sure your intuition questions this approach and suggests we can do better, and we can.

Humans don't go around the world, just remembering random things to learn from at random times. There is a more systematic way in which intelligent agents "replay memories." I'm pretty sure my dog chases rabbits in her sleep. Some experiences are more important than others to our goals. Humans often replay experiences that caused them unexpected joy or pain. And it makes sense, and you need to learn from these experiences to generate more or less of them. In the next chapter, we look at ways of prioritizing the sampling of experiences to get the most out of each sample, when we learn about the Prioritized Experience Replay (PER) method.

©Manning Publications Co. To comment go to liveBook
https://forums.manning.com/forums/grokking-deep-reinforcement-learning

WOW! eBook
www.wowebook.org

Summary

In this chapter, you learned about stabilizing value-based deep reinforcement learning methods. You dug deep on the components that make value-based methods more stable. You learned about replay buffers and target networks on an algorithm known as DQN ('Nature' DQN, or 'Vanilla' DQN). You then improved on this by implementing a double learning strategy that, when using function approximation in an algorithm called DDQN, works efficiently.

In addition to these new algorithms, you learned about different exploration strategies to use with value-based methods. You learned about linearly and exponentially decaying epsilon-greedy and SoftMax exploration strategies, this time, in the context of function approximation. Also, you learned about different loss functions and which ones make more sense for reinforcement learning and why. You learned that the Huber loss function allows you to tune between MSE and MAE with a single hyperparameter, and it is, therefore, one of the preferred loss functions used in value-based deep reinforcement learning methods.

By now you:

- Can solve reinforcement learning problems with continuous state-spaces with algorithms that are more stable and therefore give more consistent results.
- Have an understanding of state-of-the-art value-based deep reinforcement learning methods and are able to solve complex problems.

©Manning Publications Co. To comment go to liveBook
https://forums.manning.com/forums/grokking-deep-reinforcement-learning

WOW! eBook
www.wowebook.org