Going Deeper Into Reinforcement Learning: Understanding Deep-Q-Networks

Dec 1, 2016

The Deep Q-Network (DQN) algorithm, as introduced by DeepMind in a NIPS 2013 workshop paper, and later published in Nature 2015 can be credited with revolutionizing reinforcement learning. In this post, therefore, I would like to give a guide to a subset of the DQN algorithm. This is a continuation of an earlier reinforcement learning article about linear function approximators. My contribution here will be orthogonal to my previous post about the preprocessing steps for game frames.

Before I proceed, I should mention upfront that there are already a lot of great blog posts and guides that people have written about DQN. Here are the prominent ones I was able to find and read:

- Denny Britz's notes and implementation of DQN and Double-DQN. I have not read the code yet, but I imagine this will be my second-favorite DQN implementation aside from spragnur's version.
- Nervana's "Demystifying Deep Reinforcement Learning, by Tambet Matiisen. This has a useful figure to show intuition on why we want the network to take only the state as input (not the action), and an intuitive argument as to why we don't want pooling layers with Atari games (as opposed to object recognition tasks).
- Ben Lau's post on using DQN to play FlappyBird. The part on DQN might be useful. I won't need to use his code.
- A post from Machine Learning for Artists (huh, interesting) with some source code and corresponding descriptions.
- A long post from Ruben Fiszel, which also covers some of the major extensions of DQN. I will try to write more details on those papers in future blog posts, particularly A3C.
- A post from Arthur Juliani, who also mentions the target network and Double-DQN.

I will not try to repeat what these great folks have already done. Here, I focus specifically on the aspect of DQN that was the most challenging for me to understand,[1] which was about how the loss function works. I draw upon the posts above to aid me in this process.

In general, to optimize any function $f(\theta)$ which is complicated, has high-dimensional parameters and high-dimensional data points, one needs to use an algorithm such as stochastic gradient descent which relies on sampling and then approximating the gradient step $\theta_{i+1} = \theta_i - \alpha \nabla f(\theta_i)$. The key to understanding DQN is to understand how we characterize the loss function.

Recall from basic reinforcement learning that the *optimal* Q-value is defined from the Bellman optimality conditions:

$$\begin{aligned} Q^*(s,a) &= \sum_{s'} P(s'|s,a) \left[R(s,a,s') + \gamma \max_{a'} Q^*(s',a') \right] \\ &= \mathbb{E}_{s'} \left[R(s,a,s') + \gamma \max_{a'} Q^*(s',a') \mid s,a \right] \end{aligned}$$

But is this the definition of a Q-value (i.e. a $Q(s,a)$ value)?

NOPE!

This is something I had to think carefully before it finally hit home to me, and I think it's *crucial* to understanding Q-values. When we refer to $Q(s,a)$, all we are referring to is some arbitrary value. We might pull that value out of a table (i.e. tabular Q-Learning), or it might be determined based on a linear or neural network function approximator. In the latter case, it's better to write it as $Q(s,a;\theta)$. (I actually prefer writing it as $Q_\theta(s,a)$ but the DeepMind papers use the other notation, so for the rest of this post, I will use their notation.)

Our **GOAL** is to get it to satisfy the Bellman optimality criteria, which I've written above. **If** we have "adjusted" our $Q(s,a)$ function value so that for all (s,a) input pairings, it satisfies the Bellman requirements, then we rewrite the function as $Q^*(s,a)$ with the asterisk.

Let's now define a loss function. We need to do this so that we can perform stochastic gradient descent, which will perform the desired "adjustment" to $Q(s,a)$. What do we want? Given a state-action pair (s,a), the *target* will be the Bellman optimality condition. We will use the standard squared error loss. Thus, we write the loss L as:

$$L = \frac{1}{2}(\text{Bellman} - Q(s,a))^2$$

But we're not quite done, right? This is for a specific (s,a) pair. We want the Q to be close to the "Bellman" value across all such pairs. Therefore, we take expectations. Here's the tricky part: we need to take expectations with respect to samples (s,a,r,s'), so we have to consider a "four-tuple", instead of a tuple, because the target (Bellman) depends on r and s. The loss function becomes:

$$\begin{aligned} L(\theta) &= \mathbb{E}_{(s,a,r,s')}\left[\frac{1}{2}(\text{Bellman} - Q(s,a;\theta))^2\right] \\ &= \mathbb{E}_{(s,a,r,s')}\left[\frac{1}{2}\left(R(s,a,s') + \gamma \max_{a'} Q(s',a';\theta) - Q(s,a;\theta)\right)^2\right] \end{aligned}$$

Note that I have now added the parameter θ. However, this actually includes the tabular case. Why? Suppose we have two states and three actions. Thus, the total table size is six, with elements indexed by: $\{(s_1,a_1),(s_1,a_2),(s_1,a_3),(s_2,a_1),(s_2,a_2),(s_2,a_3)\}$. Now let's define a six-dimensional vector $\theta \in \mathbb{R}^6$. We will decide to encode each of the six $Q(s,a)$ values into one component of the vector. Thus, $Q(s_i,a_j;\theta) = \theta_{i,j}$. In other words, we *parameterize* the arbitrary function by θ, and we *directly* look at θ to get the Q-values! Think about how this differs from the linear approximation case I discussed in my last post. Instead of $Q(s_i,a_i;\theta)$ corresponding to one element in the parameter vector θ, it turns out to be a linear combination of the full θ along with a state-action dependent vector $\phi(s,a)$.

With the above intuition in mind, we can perform stochastic gradient descent to minimize the loss function. It is over expectations of all (s,a,r,s') samples. Intuitively, we can think of it as a function of an infinite amount of such samples. In practice, though, to approximate the expectation, we can take a finite amount of samples and use that to form the gradient updater.

We can continue with this logic until we get to the smallest possible update case, involving just one sample. What does the update look like? With one sample, we have approximated the loss as:

$$\begin{aligned} L(\theta) &= \frac{1}{2}\left(R(s,a,s') + \gamma \max_{a'} Q(s',a';\theta) - Q(s,a;\theta)\right)^2 \\ &= \frac{1}{2}\left(R(s,a,s') + \gamma \max_{a'} Q(s',a';\theta) - \theta_{s,a}\right)^2 \end{aligned}$$

I have substituted $\theta_{s,a}$, a single element in θ, since we assume the tabular case for now. I didn't do that for the target, since for the purpose of the Bellman optimality condition, we assume it is fixed for now. Since there's only one component of θ "visible" in the loss function, the gradient for all components[2] other than $\theta_{s,a}$ is zero. Hence, the gradient update is:

$$\begin{aligned}\theta_{s,a} &= \theta_{s,a} - \alpha \frac{\partial}{\partial \theta_{s,a}} L(\theta) \\ &= \theta_{s,a} - \alpha \left(R(s,a,s') + \gamma \max_{a'} Q(s', a'; \theta) - \theta_{s,a} \right) (-1) \\ &= (1-\alpha)\theta_{s,a} + \alpha \left(R(s,a,s') + \gamma \max_{a'} Q(s', a'; \theta) \right)\end{aligned}$$

Guess what? **This is exactly the standard tabular Q-Learning update taught in reinforcement learning courses!** I wrote the same exact thing in my MDP/RL review last year. Here it is, reproduced below:

$$Q(s,a) \leftarrow (1-\alpha)Q(s,a) + \alpha \underbrace{[R(s,a,s') + \gamma \max_{a'} Q(s', a')]}_{\text{sample}}$$

Let's switch gears to the DQN case, so we express $Q(s,a;\theta)$ with the implicit assumption that θ represents neural network weights. In the Nature paper, they express the loss function as:

$$L_i(\theta_i) = \mathbb{E}_{(s,a,r,s') \sim U(D)} \left[\frac{1}{2} \left(r + \gamma \max_{a} Q(s', a'; \theta_i^-) - Q(s,a;\theta_i) \right)^2 \right]$$

(I added in an extra $1/2$ that they were missing, and note that their $r = R(s,a,s')$.)

This looks *very* similar to the loss function I had before. Let's deconstruct the differences:

- The samples come from $U(D)$, which is assumed to be an experience replay history. This helps to break correlation among the samples. *Remark*: it's important to know the purpose of experience replay, but nowadays it's probably out of fashion since the A3C algorithm runs learners in parallel and thus avoids sample correlation across threads.

- Aside from the fact that we're using neural networks instead of a tabular parameterization, the weights used for the target versus the current (presumably non-optimal) Q-value are different. At iteration i, we assume we have some giant

weight vector θ_i representing all of the neural network weights. We do *not*, however, use that same vector to parameterize the network. We *fix* the targets with the *previous* weight vector θ_i^-.

These two differences are mentioned in the Nature paper in the following text on the first page:

> *We address these instabilities with a novel variant of Q-learning, which uses two key ideas. […]*

I want to elaborate on the second point in more detail, as I was confused by it for a while. Think back to my tabular Q-Learning example in this post. The target was parameterized using $Q(s', a'; \theta)$. When I perform an update using SGD, I updated $\theta_{s,a}$. If this turns out to be the same component as $\theta_{s',a'}$, then this will automatically update the target. Think of the successor state as being equal to the current state. And, again, during the gradient update, the target was assumed fixed, which is why I did not re-write $Q(s', a'; \theta)$ into a component of θ; it's as if we are "drawing" the value from the table once for the purposes of computing the target, so the value is ignored when we do the gradient computation. *After* the gradient update, the $Q(s', a'; \theta)$ value *may* be different.

DeepMind decided to do it a different way. Instead, we have to *fix* the weights θ for some number of iterations, and then allow the samples to accumulate. The argument for why this works is a bit hand-wavy and I'm not sure if there exists any rigorous mathematical justification. The DeepMind paper says that this reduces correlations with the target. This is definitely different from the tabular case I just described, where one update immediately modifies targets. If I wanted to make my tabular case the same as DeepMind's scenario, I would update the weights the normal way, but I would also fix the vector θ^-, so that when computing *targets only*, I would draw from θ^- instead of the current θ.

You can see explanations about DeepMind's special use of the target network that others have written. Denny Britz argues that it leads to more stable training:

> *Trick 2 - Target Network: Use a separate network to estimate the TD target. This target network has the same architecture as the function approximator but with frozen parameters. Every T steps (a hyperparameter) the parameters from the Q network are copied to the target network. This leads to more stable training because it keeps the target function fixed (for a while).*

Arthur Juliani uses a similar line of reasoning:

> *Why not use just use one network for both estimations? The issue is that at every step of training, the Q-network's values shift, and if we are using a constantly shifting set of values to adjust our network values, then the value estimations can easily spiral out of control. The network can become destabilized by falling into feedback loops between the target and estimated Q-values. In order to mitigate that risk, the target network's weights are fixed, and only periodically or slowly updated to the primary Q-networks values. In this way training can proceed in a more stable manner.*

I want to wrap up by doing some slight investigation of spragnur's DQN code which relate to the points above about the target network. (The neural network portion in his code is written using the Theano and Lasagne libraries, which by themselves have some learning curve; I will not elaborate on these as that is beyond the scope of this blog post.) Here are three critical parts of the code to understand:

- The `q_network.py` script contains the code for managing the networks. Professor Sprague uses a shared Theano variable to manage the input to the network, called `self.imgs_shared`. What's key is that he has to make the dimension $(N, |\phi| + 1, H, W)$, where $|\phi| = 4$ (just like in my last post). So why the +1? This is needed so that he can produce the Q-values for $Q(s', a'; \theta_i^-)$ in the loss function, which uses the *successor* state s' rather than the current state s (not to mention the previous weights θ_i^- as well!). The Nervana blog post I listed made this distinction, saying that a separate forward pass is needed to compute Q-values for the successor states. By wrapping everything together into one shared variable, spragnur's code efficiently utilizes memory.

  Here's the corresponding code segment:

```
# Shared variables for training from a minibatch of replayed
# state transitions, each consisting of num_frames + 1 (due to
# overlap) images, along with the chosen action and resulting
# reward and terminal status.
self.imgs_shared = theano.shared(
    np.zeros((batch_size, num_frames + 1, input_height, input_width
             dtype=theano.config.floatX))
```

- On a related note, the "variable" `q_vals` contains the Q-values for the current minibatch, while `next_q_vals` contains the Q-values for the successor states. Since the minibatches used in the default setting are $(32, 4+1, 84, 84)$-dimensional numpy arrays, both `q_vals` and `next_q_vals` can be thought of as arrays with 32 values each. In both cases, they are computed by calling `lasagne.layers.get_output`, which has an intuitive name.

  The code separates the `next_q_vals` cases based on a `self.freeze_interval` parameter, which is -1 and 10000 for the NIPS and Nature versions, respectively. For the NIPS version, spragnur uses the `disconnected_grad` function, meaning that the expression is *not* computed in the gradient. I believe this is similar to what I did before with the tabular Q-Learning case, when I didn't "convert" $Q(s', a'; \theta)$ to $\theta_{s',a'}$.

  The Nature version is different. It will create a second network called `self.next_l_out` which is independent of the first network `self.l_out`. During the training process, the code periodically calls `self.reset_q_hat()` to update the weights of `self.next_l_out` by copying from the current weights of `self.l_out`.

  Both of these should accomplish the same goal of having a separate network whose weights are updated periodically. The Nature version is certainly easier to understand.

- I briefly want to mention a bit about the evaluation metrics used. The code, following the NIPS and Nature papers, reports the average reward per episode, which is intuitive and what we ultimately want to optimize. The NIPS paper argued that the average reward metric is extremely noisy, so they also reported on the average action value encountered during testing. They collected a random amount of states $\mathcal{S}$ and for each $s \in \mathcal{S}$, tracked the $Q(s, a)$-value obtained from that state, and found that the average discounted reward kept increasing. Professor Sprague's code computes this value in the `ale_agent.py` file in the `finish_testing` method. He seems to do it different from the way the NIPS paper reports it, though, by subsampling a random set of states after each epoch instead of having that random set fixed for all 100 epochs (I don't see `self.holdout_data` assigned anywhere). But, meh, it does basically the same thing.

I know I said this before, but I really like spragnur's code.

That's all I have to say for now regarding DQNs. I hope this post serves as a useful niche

in the DQN blog-o-sphere by focusing specifically on how the loss function gets derived. In subsequent posts, I hope to touch upon the major variants of DQN in more detail, particularly A3C. Stay tuned!

---

1. The most *annoying* part of DQN, though, was figuring out how the preprocessing works. As mentioned earlier, I resolved that through a previous post here. ↩

2. Taking the gradient of a function from a vector to a scalar, such as $L(\theta)$, involves taking partial derivatives for each component. All components other than $\theta_{s,a}$ in this example here will have derivatives of 0. ↩

Seita's Place

Seita's Place
seita@berkeley.edu

DanielTakeshi
(Never!)

This is my blog, where I have written over 250 articles on a variety of topics, most of which are about one of two major

themes. The first is computer science, which is my area of specialty as a Ph.D. student at UC Berkeley. The second can be broadly categorized as "deafness," which relates to my experience and knowledge of being deaf.