Published in Towards Data Science

Chris Yoon
Jul 17, 2019 · 5 min read · Listen

DQN FAMILY

Double Deep Q Networks

Tackling maximization bias in Deep Q-learning

Introduction

In this post, we will look into the motivation behind double Q-learning networks, and look at three different ways this has been done:

- The original algorithm in "Double Q-learning" (Hasselt, 2010)
- The updated algorithm from the same author in "Deep Reinforcement Learning with Double Q-learning" (Hasselt et al., 2015),
- The most recent method, Clipped Double Q-learning, in "Addressing Function Approximation Error in Actor-Critic Methods" (Fujimoto et al., 2018).

If you aren't completely familiar with Q-learning, I would recommend taking a quick look at my post about Q-learning!

Motivation

Consider the target Q value:

$$Q^*(s,a) = \mathbb{E}_{s' \sim \mathcal{E}}[r + \gamma \ \max_{a'} Q^*(s',a') |\ s,a]$$

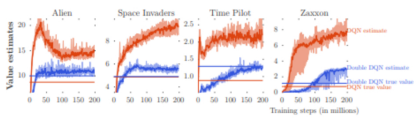
Specifically,

$$max_{a'} Q^*(s',a')$$

Taking the maximum overestimated values as such is implicitly taking the estimate of the maximum value. This systematic overestimation introduces a maximization bias in learning. And since Q-learning involves bootstrapping — learning estimates from estimates — such overestimation can be problematic.

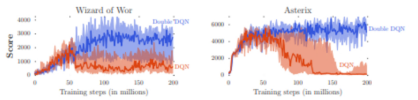
> *Here is an example: consider a single state s where the true Q value for all actions equal 0, but the estimated Q values are distributed some above and below zero. Taking the maximum of these estimates (which is obviously bigger than zero) to update the Q function leads to the overestimation of Q values.*

Hasselt et al. (2015) illustrates this overestimation bias in experiments across different Atari game environments:



Source: "Deep Reinforcement Learning with Double Q-learning" (Hasselt et al., 2015),

As we can see, traditional DQN tends to significantly overestimate action-values, leading to unstable training and low quality policy:



. . .

Solution: Double Q learning

Chris Yoon
581 Followers
Student in NYC. https://www.linkedin.com/in/chris-yoon-75847418b/

The solution involves using two separate Q-value estimators, each of which is used to update the other. Using these independent estimators, we can unbiased Q-value estimates of the actions selected using the opposite estimator [3]. We can thus avoid maximization bias by disentangling our updates from biased estimates.

Below, we will take a look at 3 different formulations of Double Q learning, and implement the latter two.

1. The original algorithm in "Double Q-learning" (Hasselt, 2010)

Algorithm 1 Double Q-learning

Initialize Q^A, Q^B, s
repeat
  Choose a, based on $Q^A(s,\cdot)$ and $Q^B(s,\cdot)$, observe r, s'
  Choose (e.g. random) either UPDATE(A) or UPDATE(B)
  if UPDATE(A) **then**
    Define $a^* = \arg\max_a Q^A(s', a)$
    $Q^A(s,a) \leftarrow Q^A(s,a) + \alpha(s,a)\left(r + \gamma Q^B(s', a^*) - Q^A(s,a)\right)$
  else if UPDATE(B) **then**
    Define $b^* = \arg\max_a Q^B(s', a)$
    $Q^B(s,a) \leftarrow Q^B(s,a) + \alpha(s,a)(r + \gamma Q^A(s', b^*) - Q^B(s,a))$
  end if
  $s \leftarrow s'$
until end

Pseudo-code Source: "Double Q-learning" (Hasselt, 2010)

The original Double Q-learning algorithm uses two independent estimates `Q^{A}` and `Q^{B}` . With a 0.5 probability, we use estimate `Q^{A}` to determine the maximizing action, but use it to update `Q^{B}` . Conversely, we use `Q^{B}` to determine the maximizing action, but use it to update `Q^{A}` . By doing so, we obtain an unbiased estimator `Q^{A}(state, argmaxQ^{next state, action)` for the expected Q value and inhibit bias.

2. The updated version from the same author in "Deep Reinforcement Learning with Double Q-learning" (Hasselt et al., 2015),

Algorithm 1 : Double Q-learning (Hasselt et al., 2015)

Initialize primary network Q_θ, target network $Q_{\theta'}$, replay buffer $\mathcal{D}$, $\tau << 1$
for each iteration **do**
  for each environment step **do**
    Observe state s_t and select $a_t \sim \pi(a_t, s_t)$
    Execute a_t and observe next state s_{t+1} and reward $r_t = R(s_t, a_t)$
    Store (s_t, a_t, r_t, s_{t+1}) in replay buffer $\mathcal{D}$
  for each update step **do**
    sample $e_t = (s_t, a_t, r_t, s_{t+1}) \sim \mathcal{D}$
    Compute target Q value:
      $Q^*(s_t, a_t) \approx r_t + \gamma\ Q_\theta(s_{t+1}, argmax_{a'} Q_{\theta'}(s_{t+1}, a'))$
    Perform gradient descent step on $(Q^*(s_t, a_t) - Q_\theta(s_t, a_t))^2$
    Update target network parameters:
      $\theta' \leftarrow \tau * \theta + (1 - \tau) * \theta'$

In the second Double Q-learning algorithm, we have a model `Q` and a target model `Q'` instead of two independent models, as in (Hasselt, 2010). We use the `Q'` for action selection and `Q` for action evaluation. That is:

$$Q^*(s_t, a_t) \approx r_t + \gamma\ Q(s_{t+1}, \text{argmax}_{a'} Q'(s_t, a_t))$$

We minimize the mean squared error between `Q` and `Q*` , but we have `Q'` slowly copy the parameters of `Q` . We can do so by periodically hard-copying over the parameters, but also through Polyak averaging:

$$\theta' \leftarrow \tau\theta + (1 - \tau)\theta'$$

where θ' is the target network parameter, θ is the primary network parameter, and τ (rate of averaging) is usually set to 0.01.

3. Clipped Double Q-learning, in "Addressing Function Approximation Error in Actor-Critic Methods" (Fujimoto et al., 2018).

Algorithm 1 : Clipped Double Q-learning (Fujimoto et al., 2018)

Initialize networks Q_{θ_1}, Q_{θ_1}, replay buffer $\mathcal{D}$
for each iteration **do**
  for each environment step **do**
    Observe state s_t and select $a_t \sim \pi(a_t, s_t)$
    Execute a_t and observe next state s_{t+1} and reward $r_t = R(s_t, a_t)$
    Store (s_t, a_t, r_t, s_{t+1}) in replay buffer $\mathcal{D}$
  for each update step **do**
    sample $e_t = (s_t, a_t, r_t, s_{t+1}) \sim \mathcal{D}$
    Compute target Q value:
      $Q^*(s_t, a_t) \approx r_t + \gamma min_{i=1,2} Q_{\theta_i}(s_{t+1}, argmax_{a'} Q_{\theta_i}(s_{t+1}, a'))$
    Perform gradient descent step on $(Q^*(s_t, a_t) - Q_\theta(s_t, a_t))^2$

Update target network parameters:
$\theta' \leftarrow \tau * \theta + (1 - \tau) * \theta'$

In Clipped Double Q-learning, we follow the original formulation of Hasselt 2015. We have two independent estimates of the true Q value. Here, for computing the update targets, we take the minimum of the two next-state action values produced by our two Q networks; When the Q estimate from one is greater than the other, we reduce it to the minimum, avoiding overestimation.

Fujimoto et al. presents another benefit of this setting: the minimum operator should provide higher value to states with lower variance estimation error. This means that the minimization will lead to a preference for states with low-variance value estimates, leading to safer policy updates with stable learning targets.

. . .

Implementation Guide

We will begin with the same DQN agent setup as Part 1 of this series. If you would like to see a more complete implementation of the setup, please see my Q-learning post or my Github Repository (link in the bottom).

DQN Agent:

```
class DQNAgent:

    def __init__(self, env, use_conv=True, learning_rate=3e-4, gamma=0.99, buffer_size=10(
        self.env = env
        self.learning_rate = learning_rate
        self.gamma = gamma
        self.replay_buffer = BasicBuffer(max_size=buffer_size)

        self.device = torch.device("cuda" if torch.cuda.is_available() else "cpu")

        # initialization of models of Hasselt et al. (2015) and Fujimoto et al. (2018) ar
        # We will go through each.
        # ..
        # ..

    def get_action(self, state, eps=0.20):
        state = torch.FloatTensor(state).float().unsqueeze(0).to(self.device)
        qvals = self.model.forward(state)
        action = np.argmax(qvals.cpu().detach().numpy())

        if(np.random.randn() < eps):
            return self.env.action_space.sample()

        return action

    def compute_loss(self, batch):
        # Hasselt et al. (2015) and Fujimoto et al. (2018) will be different.
        # We will go through each.
        # ..
        # ..
        return loss

    def update(self, batch_size):
        # Hasselt et al. (2015) and Fujimoto et al. (2018) will have slightly different up
        # We will go through each.
        # ..
        # ..
```

agent.py hosted with ❤ by GitHub view raw

1. **Double Q-learning in Hasselt et al., 2015:**

We will initialize a model and a target model:

```
self.use_conv = use_conv
if self.use_conv:
    self.model = ConvDQN(env.observation_space.shape, env.action_space.n).to(self.device)
    self.target_model = ConvDQN(env.observation_space.shape, env.action_space.n).to(self.d
else:
    self.model = DQN(env.observation_space.shape, env.action_space.n).to(self.device)
    self.target_model = DQN(env.observation_space.shape, env.action_space.n).to(self.devi

# hard copy model parameters to target model parameters
for target_param, param in zip(self.model.parameters(), self.target_model.parameters()):
    target_param.data.copy_(param)

self.optimizer = torch.optim.Adam(self.model.parameters())
```

hasselt_init.py hosted with ❤ by GitHub view raw

For computing the loss, we use our target model to compute our next Q values:

```
def compute_loss(self, batch):
```

```
def compute_loss(self, batch):
    states, actions, rewards, next_states, dones = batch
    states = torch.FloatTensor(states).to(self.device)
    actions = torch.LongTensor(actions).to(self.device)
    rewards = torch.FloatTensor(rewards).to(self.device)
    next_states = torch.FloatTensor(next_states).to(self.device)
    dones = torch.FloatTensor(dones)

    # resize tensors
    actions = actions.view(actions.size(0))
    dones = dones.view(dones.size(0))

    # compute loss
    curr_Q = self.model.forward(states).gather(1, actions.view(actions.size(0), 1))
    next_Q = self.target_model.forward(next_states)
    max_next_Q = torch.max(next_Q, 1)[0]
    max_next_Q = max_next_Q.view(max_next_Q.size(0), 1)
    expected_Q = rewards + (1 - dones) * self.gamma * max_next_Q

    loss = F.mse_loss(curr_Q, expected_Q.detach())

    return loss
```

ddqn_hasselt.py hosted with ❤ by GitHub — view raw

And then we slowly copy/average the model parameters over to the target model parameters:

```
def update(self, batch_size):
    batch = self.replay_buffer.sample(batch_size)
    loss = self.compute_loss(batch)

    self.optimizer.zero_grad()
    loss.backward()
    self.optimizer.step()

    # target network update
    for target_param, param in zip(self.target_model.parameters(), self.model.parameters(
        target_param.data.copy_(self.tau * param + (1 - self.tau) * target_param)
```

update.py hosted with ❤ by GitHub — view raw

2. **Clipped Double Q-learning in Fujimoto et al., 2018:**

We initialize two Q networks:

```
self.use_conv = use_conv
if self.use_conv:
    self.model1 = ConvDQN(env.observation_space.shape, env.action_space.n).to(self.device
    self.model2 = ConvDQN(env.observation_space.shape, env.action_space.n).to(self.device
else:
    self.model1 = DQN(env.observation_space.shape, env.action_space.n).to(self.device)
    self.model2 = DQN(env.observation_space.shape, env.action_space.n).to(self.device)

self.optimizer1 = torch.optim.Adam(self.model1.parameters())
self.optimizer2 = torch.optim.Adam(self.model2.parameters())
```

init_fujimoto.py hosted with ❤ by GitHub — view raw

For computing the loss, we compute the current-state-Q-values and the next-state-Q-values of both models, but use the minimum of the next-state-Q-values to compute the expected Q value. We then update both models using the expected Q value.

```
def compute_loss(self, batch):
    states, actions, rewards, next_states, dones = batch
    states = torch.FloatTensor(states).to(self.device)
    actions = torch.LongTensor(actions).to(self.device)
    rewards = torch.FloatTensor(rewards).to(self.device)
    next_states = torch.FloatTensor(next_states).to(self.device)
    dones = torch.FloatTensor(dones)

    # resize tensors
    actions = actions.view(actions.size(0), 1)
    dones = dones.view(dones.size(0), 1)

    # compute loss
    curr_Q1 = self.model1.forward(states).gather(1, actions)
    curr_Q2 = self.model2.forward(states).gather(1, actions)

    next_Q1 = self.model1.forward(next_states)
    next_Q2 = self.model2.forward(next_states)
    next_Q = torch.min(
        torch.max(self.model1.forward(next_states), 1)[0],
        torch.max(self.model2.forward(next_states), 1)[0]
    )
    next_Q = next_Q.view(next_Q.size(0), 1)
    expected_Q = rewards + (1 - dones) * self.gamma * next_Q

    loss1 = F.mse_loss(curr_Q1, expected_Q.detach())
    loss2 = F.mse_loss(curr_Q2, expected_Q.detach())

    return loss1, loss2
```

clipped_compute_loss.py hosted with ❤ by GitHub — view raw

And finally the update function:

```python
def update(self, batch_size):
    batch = self.replay_buffer.sample(batch_size)
    loss1, loss2 = self.compute_loss(batch)

    self.optimizer1.zero_grad()
    loss1.backward()
    self.optimizer1.step()

    self.optimizer2.zero_grad()
    loss2.backward()
    self.optimizer2.step()
```

update_fujimoto.py hosted with ❤ by GitHub view raw

. . .

This concludes our implementations of double Q-learning algorithms. Double Q-learning is frequently used in state-of-the-art Q-learning variants and Actor Critic methods. We will see this technique appear time and again in our later posts.

Thanks for reading!

Find my full implementation here:

cyoon1729/deep-Q-networks

Modular Implementations of algorithms from the Q-learning family (PyTorch). Implementations inlcude: DQN, DDQN, Dueling...

github.com

References

- "Issues in Using Function Approximation for Reinforcement Learning" (Thrun and Schwartz, 1993)
- "Double Q-learning" (Hasselt, 2010)
- "Deep Reinforcement Learning with Double Q-learning" (Hasselt et al., 2015),
- "Addressing Function Approximation Error in Actor-Critic Methods" (Fujimoto et al., 2018)
- Reinforcement Learning: An Introduction (Sutton and Barto)

Your dataset is a giant inkblot test

The danger of apophenia in analytics and what you can do about it — There's a fine line between telling stories with data and telling lies. Before I tell you how to spot a top-notch data analyst and boost your analytical…

Data Science 6 min read

Rising Odegua · Jul 17, 2019 ★

Creating Machine Learning Models

Using, testing and comparing multiple machine learning models — Now that we have successfully done the hard part-data cleaning and wrangling- of every data science project, we will move on to the interesting part,…

Machine Learning 8 min read

Evan Morris · Jul 17, 2019

Top 5 Data Governance Framework Tools To Look Out For

Understanding Data Governance Data has been hailed as oil of the 21st century, and organizations have finally woken up to a world where data i…

Big Data 5 min read

Matthew Wasserman · Jul 17, 2019

Face/Off: High speed facial tracking using the Viola Jones Method

Ever wonder how that box on your digital camera knows exactly where and what a face is? Back in 2011, before Convolutional Neural Networks or a…

Machine Learning 5 min read

Read more from Towards Data Science

Recommended from Medium

Irene Pham in Analytics Vidhya

Ensemble Learning — Your Machine Learning savoir and here is why(Part 1)

Gio at QRC

Supervised vs. Unsupervised Learning

Jean Salvi

Build mini TensorFlow-like library from scratch.

Johnny Wa… in Towards Data Scie…

Time series forecasting with 2D convolutions

Sanskruti Panda

Some popular Python Libraries used in Machine Learning :-

Kenneth Ong

SPOBET

Ahmed G… in Towards Data Scien…

Feature Reduction using Genetic Algorithm with Python

Harini Narasimhan

What is Learning Rate?