Deep Reinforcement Learning

Jeremy Morton

November 29, 2017
Reinforcement Learning Challenges

1. Balancing Exploration with Exploitation
2. Credit Assignment
3. Generalizing from experience

How can we generalize effectively in large (and possibly continuous) state and action spaces?

Driving from Images

Continuous Action Space
Recall global approximation:
\[ Q(s, a) = \theta_a \beta(s) \]

What is \( \beta(s) \)? If \( \beta(s) = s \), then this is may not be a good approximation of \( Q(s, a) \).

*But how can we learn appropriate features and weights so that this approximation is accurate?*
Here, we have a supervised learning problem and would like to approximate $y$. 
\[ i_1 = w_1^T x + b_1 \]
Neural Networks: Overview

\[ o_1 = \sigma(w_1^\top x + b_1) \]
$o_1 = \sigma(w_1^\top x + b_1)$
\[
\hat{y} = w_3 \sigma \left( w_2 \sigma (w_1^\top x + b_1) + b_2 \right) + b_3
\]
Neural networks transform inputs via a set of matrix multiplications and element-wise nonlinearities.

\[ \hat{y} = W_3 \sigma(W_2 \sigma(W_1 x + b_1) + b_2) + b_3 \]
Neural networks are universal function approximators.

Recall the properties of the sigmoid function: we can control its sharpness and shift it.

\[
\sigma(x) \quad \sigma(1000x) \quad \sigma(1000x - 5000)
\]
Neural networks are universal function approximators.

Imagine we wanted to approximate \( f(x) = x \):
Neural networks are universal function approximators.

Imagine you have a neural network with three neurons that take in $x$ and output the following:

\[
\sigma(1000x - 1000) \quad \quad -\sigma(1000x - 2000) \quad \quad 2\sigma(1000x - 2000)
\]
Neural networks are universal function approximators.

Now sum the neuron outputs together:

![Graph of neural network output](image-url)
Neural networks are universal function approximators.

As more neurons are added, the approximation to $f(x)$ improves.

Good explanation here.
Neural networks can be trained efficiently through backpropagation.

Example for network with single hidden layer:

\[ \hat{y} = W_2 \sigma(W_1 x + b_1) + b_2 \]
Neural networks can be trained efficiently through backpropagation.

Example for network with single hidden layer:

\[ \hat{y} = W_2 \sigma(W_1 x + b_1) + b_2 \]

If loss function \( L \) is a function of \( \hat{y} \) then by chain rule:

\[
\frac{\partial L}{\partial W_2} = \frac{\partial L}{\partial \hat{y}} \left( \frac{\partial \hat{y}}{\partial W_2} \right)^T \\
= \frac{\partial L}{\partial \hat{y}} \sigma(W_1 x + b_1)^T
\]
Neural networks can be trained efficiently through backpropagation.

Example for network with single hidden layer:

\[ \hat{y} = W_2 \sigma(W_1x + b_1) + b_2 \]

If loss function \( L \) is a function of \( \hat{y} \) then by chain rule:

\[
\frac{\partial L}{\partial W_2} = \frac{\partial L}{\partial \hat{y}} \left( \frac{\partial \hat{y}}{\partial W_2} \right)^T
= \frac{\partial L}{\partial \hat{y}} \sigma(W_1x + b_1)^T
\]

Finally, can update \( W_2 \) using gradient descent:

\[ W_2 \leftarrow W_2 - \alpha \frac{\partial L}{\partial W_2} \]
Deep $Q$-Learning

Recall the Bellman equation:

\[ Q^*(s, a) = \mathbb{E}_{s' \sim \mathcal{E}} \left[ R(s, a) + \gamma \max_{a'} Q^*(s', a') \right] \]

We estimate the value of \( Q^*(s, a) \) using a neural network with parameters \( \theta \). At iteration \( i \), the loss function is given by the temporal difference error:

\[ \mathcal{L}_i(\theta_i) = \mathbb{E}_{s, a \sim \rho(\cdot); s' \sim \mathcal{E}} \left[ \left( R(s, a) + \gamma \max_{a'} Q(s', a'; \theta_{i-1}) - Q(s, a; \theta_i) \right)^2 \right] \]

where \( \theta_{i-1} \) are the parameter values from the previous iteration.
Deep Q-Learning: Challenges

- Often in supervised learning it is assumed that successive samples are iid.
- However, in reinforcement learning successive samples are highly correlated.
- To combat this, transitions were stored in replay memory $\mathcal{D}$.
- During training, random transitions $\{s_t, a_t, r_t, s_{t+1}\}$ were sampled from $\mathcal{D}$ and used to train the network.
When performing regression in supervised learning, it is common to use the least-squares loss:

\[ \mathcal{L} = \frac{1}{2} (y - \hat{y})^2 \]

where \( y \) is the target value and \( \hat{y} \) is the model prediction. In the case of \( Q \)-Learning, this target value is:

\[ y = r_t + \gamma \max_{a'} Q(s_{t+1}, a'; \theta). \]

Because of this, the network target value changes as the network is trained. The deep \( Q \)-Learning algorithm uses a target network whose parameters are changed less frequently in order to have a stationary target value.
Deep $Q$-Learning: Challenges

- Input images are $210 \times 160$ pixels with three color channels, so huge inputs and state space → Images are downsampled, converted to grayscale, and cropped to be $84 \times 84$ pixels.

- Images are static, so many states are perceptually aliased → The input to the network is a sequence of 4 preprocessed frames.
Deep $Q$-Learning: Network Architecture
Algorithm 1 Deep Q-learning with Experience Replay

Initialize replay memory $\mathcal{D}$ to capacity $N$
Initialize action-value function $Q$ with random weights

for episode = 1, $M$ do
    Initialise sequence $s_1 = \{x_1\}$ and preprocessed sequenced $\phi_1 = \phi(s_1)$
    for $t = 1, T$ do
        With probability $\epsilon$ select a random action $a_t$
        otherwise select $a_t = \max_a Q^*(\phi(s_t), a; \theta)$
        Execute action $a_t$ in emulator and observe reward $r_t$ and image $x_{t+1}$
        Set $s_{t+1} = s_t, a_t, x_{t+1}$ and preprocess $\phi_{t+1} = \phi(s_{t+1})$
        Store transition $(\phi_t, a_t, r_t, \phi_{t+1})$ in $\mathcal{D}$
        Sample random minibatch of transitions $(\phi_j, a_j, r_j, \phi_{j+1})$ from $\mathcal{D}$
        Set $y_j = \begin{cases} r_j & \text{for terminal } \phi_{j+1} \\ r_j + \gamma \max_{a'} Q(\phi_{j+1}, a'; \theta) & \text{for non-terminal } \phi_{j+1} \end{cases}$
        Perform a gradient descent step on $(y_j - Q(\phi_j, a_j; \theta))^2$ according to equation 3
    end for
end for
Deep $Q$-Learning: Results
In Deep Q-learning, a neural network is used to approximate $Q^*(s, a)$. In policy gradient methods, neural networks are instead used to represent a policy $\pi_{\theta}(a|s)$. Here, the policy is stochastic, i.e. it provides a distribution over possible actions. Parameters $\theta$ optimized to maximize expectation over future rewards $E_{a \sim \pi_{\theta}}[Q_{\pi_{\theta}}(s,a)]$. Can be applied to both discrete and continuous action spaces.
In Deep Q-learning, a neural network is used to approximate $Q^*(s, a)$.

In policy gradient methods, neural networks are instead used to represent a policy $\pi_\theta(a \mid s)$.

Parameters $\theta$ optimized to maximize expectation over future rewards $E_{a \sim \pi_\theta}[Q_{\pi_\theta}(s, a)]$. Can be applied to both discrete and continuous action spaces.
In Deep Q-learning, a neural network is used to approximate $Q^*(s, a)$.

In policy gradient methods, neural networks are instead used to represent a policy $\pi_\theta(a | s)$.

Here, the policy is *stochastic*, i.e. it provides a distribution over possible actions.
In Deep Q-learning, a neural network is used to approximate $Q^*(s, a)$.

In policy gradient methods, neural networks are instead used to represent a policy $\pi_{\theta}(a \mid s)$.

Here, the policy is *stochastic*, i.e. it provides a distribution over possible actions.

Parameters $\theta$ optimized to maximize expectation over future rewards $\mathbb{E}_{a \sim \pi_{\theta}} [Q_{\pi_{\theta}}(s, a)]$. 
In Deep Q-learning, a neural network is used to approximate $Q^*(s, a)$.

In policy gradient methods, neural networks are instead used to represent a policy $\pi_\theta(a \mid s)$.

Here, the policy is *stochastic*, i.e. it provides a distribution over possible actions.

Parameters $\theta$ optimized to maximize expectation over future rewards $\mathbb{E}_{a \sim \pi_\theta} [Q^{\pi_\theta}(s, a)]$.

Can be applied to both discrete and continuous action spaces.
Calculating Gradients

How do we find gradients for the policy parameters?

$$\nabla_{\theta} \mathbb{E}_{a \sim \pi_{\theta}} [Q_{\pi_{\theta}}(s, a)] = \nabla_{\theta} \int \pi_{\theta}(a | s) Q_{\pi_{\theta}}(s, a) da$$
Calculating Gradients

How do we find gradients for the policy parameters?

\[ \nabla_\theta \mathbb{E}_{a \sim \pi_\theta} [Q^\pi_\theta(s, a)] = \nabla_\theta \int \pi_\theta(a \mid s) Q^\pi_\theta(s, a) \, da \]

\[ = \int \nabla_\theta \pi_\theta(a \mid s) Q^\pi_\theta(s, a) \, da \]
Calculating Gradients

How do we find gradients for the policy parameters?

\[
\nabla_\theta \mathbb{E}_{a \sim \pi_\theta} [Q^\pi_\theta(s, a)] = \nabla_\theta \int \pi_\theta(a \mid s) Q^\pi_\theta(s, a) da \\
= \int \nabla_\theta \pi_\theta(a \mid s) Q^\pi_\theta(s, a) da \\
= \int \frac{\nabla_\theta \pi_\theta(a \mid s)}{\pi_\theta(a \mid s)} \pi_\theta(a \mid s) Q^\pi_\theta(s, a) da
\n\]
Calculating Gradients

How do we find gradients for the policy parameters?

\[
\nabla_\theta \mathbb{E}_{a \sim \pi_\theta} \left[ Q^{\pi_\theta}(s, a) \right] = \nabla_\theta \int \pi_\theta(a \mid s) Q^{\pi_\theta}(s, a) da
\]

\[
= \int \nabla_\theta \pi_\theta(a \mid s) Q^{\pi_\theta}(s, a) da
\]

\[
= \int \frac{\nabla_\theta \pi_\theta(a \mid s)}{\pi_\theta(a \mid s)} \pi_\theta(a \mid s) Q^{\pi_\theta}(s, a) da
\]

\[
= \int \nabla_\theta \log \pi_\theta(a \mid s) \pi_\theta(a \mid s) Q^{\pi_\theta}(s, a) da
\]
Calculating Gradients

How do we find gradients for the policy parameters?

\[ \nabla_\theta \mathbb{E}_{a \sim \pi_\theta} [Q^{\pi_\theta}(s, a)] = \nabla_\theta \int \pi_\theta(a \mid s)Q^{\pi_\theta}(s, a)da \]

\[ = \int \nabla_\theta \pi_\theta(a \mid s)Q^{\pi_\theta}(s, a)da \]

\[ = \int \frac{\nabla_\theta \pi_\theta(a \mid s)}{\pi_\theta(a \mid s)} \pi_\theta(a \mid s)Q^{\pi_\theta}(s, a)da \]

\[ = \int \nabla_\theta \log \pi_\theta(a \mid s)\pi_\theta(a \mid s)Q^{\pi_\theta}(s, a)da \]

\[ = \mathbb{E}_{a \sim \pi_\theta} [\nabla_\theta \log \pi_\theta(a \mid s)Q^{\pi_\theta}(s, a)] \]
Policy Gradients: Estimating $Q$

Given a sample trajectory $\{s_1, a_1, r_1, \ldots s_{T-1}, a_{T-1}, r_T\}$, can create an empirical estimate of $Q$:

$$\hat{Q}^{\pi_\theta}(s_t, a_t) = \sum_{k=t}^{T} \gamma^{k-1} r_k$$

Then perform the following parameter update at each time step:

$$\theta \leftarrow \theta + \alpha \nabla_\theta \log \pi_\theta(a_t \mid s_t) \hat{Q}^{\pi_\theta}(s_t, a_t).$$

This is known as the **REINFORCE** algorithm. It is also possible to use a second neural network to estimate $Q$. This is known as an **actor-critic** algorithm.
Policy Gradients: Demonstration

Iteration 0

Full video [here](#).
Further Reading


**Performance of Asynchronous Advantage Actor-Critic (A3C) algorithm**

<table>
<thead>
<tr>
<th>Method</th>
<th>Training Time</th>
<th>Mean</th>
<th>Median</th>
</tr>
</thead>
<tbody>
<tr>
<td>DQN</td>
<td>8 days on GPU</td>
<td>121.9%</td>
<td>47.5%</td>
</tr>
<tr>
<td>Gorila</td>
<td>4 days, 100 machines</td>
<td>215.2%</td>
<td>71.3%</td>
</tr>
<tr>
<td>D-DQN</td>
<td>8 days on GPU</td>
<td>332.9%</td>
<td>110.9%</td>
</tr>
<tr>
<td>Dueling D-DQN</td>
<td>8 days on GPU</td>
<td>343.8%</td>
<td>117.1%</td>
</tr>
<tr>
<td>Prioritized DQN</td>
<td>8 days on GPU</td>
<td>463.6%</td>
<td>127.6%</td>
</tr>
<tr>
<td>A3C, FF</td>
<td>1 day on CPU</td>
<td>344.1%</td>
<td>68.2%</td>
</tr>
<tr>
<td>A3C, FF</td>
<td>4 days on CPU</td>
<td>496.8%</td>
<td>116.6%</td>
</tr>
<tr>
<td>A3C, LSTM</td>
<td>4 days on CPU</td>
<td>623.0%</td>
<td>112.6%</td>
</tr>
</tbody>
</table>

Credit: Medium post by A. Juliani [here](https://example.com)
Further Reading

Further Reading

Currently, the (arguably) two most popular frameworks are:

- TensorFlow (https://www.tensorflow.org)
- PyTorch (http://pytorch.org)

With both, you can construct and train networks in Python. Other frameworks include Theano, CNTK, Caffe, MXNet, etc.

For easy network construction and training, check out Keras.

A much more in-depth discussion of deep learning frameworks can be found in the CS 231n course slides.
Additional Resources

Neural Networks:
- CS 231n Course Notes (http://cs231n.stanford.edu)
- Michael Nielsen’s free textbook Neural Networks and Deep Learning (http://neuralnetworksanddeeplearning.com)
- Deep Learning by Bengio et al. (http://www.deeplearningbook.org)

Deep Reinforcement Learning:
- CS 294 Course Notes (UC Berkeley) (http://rll.berkeley.edu/deeprlcourse/)
- David Silver’s RL Course Notes (http://www0.cs.ucl.ac.uk/staff/D.Silver/web/Teaching.html)
- OpenAI Gym (https://gym.openai.com/docs/)
Questions?