Deep Learning @Unity3D & Cognitive Neuroscience PhD student.

Nov 15, 2016 · 8 min read

Simple Reinforcement Learning with Tensorflow Part 7: Action-Selection Strategies for Exploration

- In this entry of my RL series I would like to focus on the role that exploration plays in an agent's behavior. I will go over a few of the commonly used approaches to exploration which focus on action-selection, and show their comparative strengths and weaknesses, as well as demonstrate how to implement each using Tensorflow. The methods are discussed here in the context of a Q-Network, but can be applied to Policy Networks as well. To make things more intuitive, I also built an interactive visualization to provide a better sense of how each exploration strategy works (It uses simulated Q-values, so there is no actual neural network running in the browser—though such things do exist!). Since I can't embed it in Medium, I have linked to it here, and below. I highly recommend playing with it as you read through the post. Let's get started!

Reinforcement Learning Exploration

Interactive Visualization of Action Selection Strategies: awjuliani.github.io

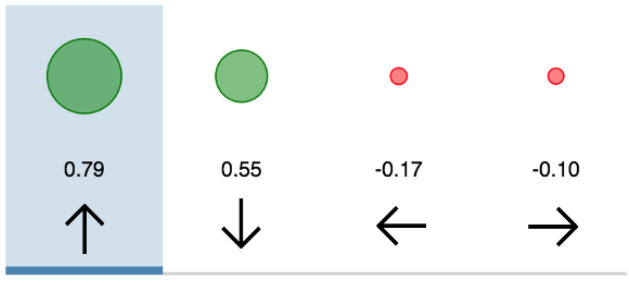
Why Explore?

The first question one may ask is: why do we need exploration at all? The problem can be framed as one of obtaining representative training data. In order for an agent to learn how to deal optimally with all possible states in an environment, it must be exposed to as many of those states as possible. Unlike in traditional supervised learning settings however, the agent in a

reinforcement learning problem only has access to the environment through its own actions. As a result, there emerges a chicken and egg problem: An agent needs the right experiences to learn a good policy, but it also needs a good policy to obtain those experiences.

From this problem has emerged an entire subfield within reinforcement learning that has attempted to develop techniques for meaningfully balancing the *exploration* and *exploitation* tradeoff. Ideally, such an approach should encourage exploring an environment until the point that it has learned enough about it to make informed decisions about optimal actions. There are a number of frequently used approaches to encouraging exploration. In this post I want to go over some of the basic approaches related to the *selection of actions*. In a later post in this series I will cover more advanced methods which encourage exploration through the use of *intrinsic motivation*.

Greedy Approach



Each value corresponds to the Q-value for a given action at a random state in an environment. The height of the light blue bar corresponds to the probability of choosing a given action. The dark blue bar corresponds to a chosen action. To try an interactive version, go here.

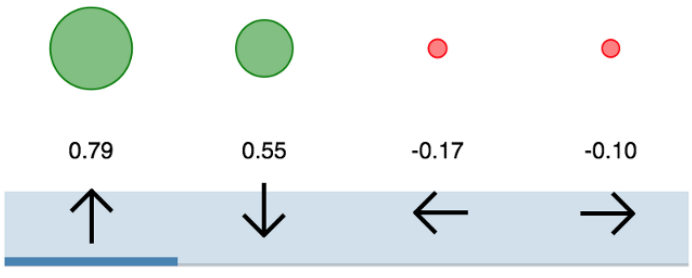
Explanation*:* All reinforcement learning algorithms seek to maximize reward over time. A naive approach to ensuring the optimal action is taken at any given time is to simply choose the action which the agent expects to provide the greatest reward. This is referred to as a *greedy method*. Taking the action which the agent estimates to be the best at the current moment is an example of exploitation: the agent is exploiting its current knowledge about the reward structure of the environment to act. This approach can be thought of as providing little to no exploratory potential.

Shortcomings*:* The problem with a greedy approach is that it almost universally arrives at a suboptimal solution. Imagine a simple two-armed bandit problem (for an introduction to multi-armed bandits, see Part 1 of this series). If we suppose one arm gives a reward of 1 and the other arm gives a reward of 2, then if the agent's parameters are such that it chooses the former arm first, then regardless of how complex a neural network we utilize, under a greedy approach it will never learn that the latter action is more optimal.

Implementation*:*

```
#Use this for action selection.
#Q_out referrs to activation from final layer of Q-Netw
Q_values = sess.run(Q_out, feed_dict={inputs:[state]})
action = np.argmax(Q_values)
```

Random Approach



Each value corresponds to the Q-value for a given action at a random state in an environment. The height of the light blue bars correspond to the probability of choosing a given action. The dark blue bar corresponds to a chosen action. To try an interactive version, go here.

Explanation*:* The opposite approach to greedy selection is to simply always take a random action.

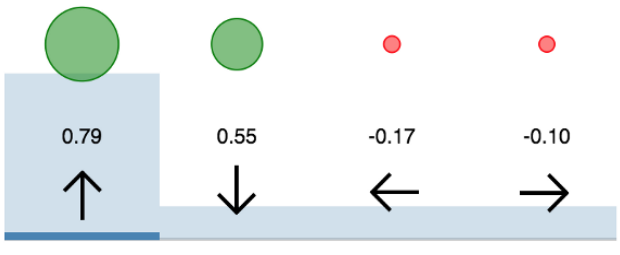
Shortcomings*:* Only in circumstances where a random policy is optimal would this approach be ideal. However it can be useful as an initial means of sampling from the state space in order to fill an experience buffer when using DQN.

Implementation*:*

```
#Assuming we are using OpenAI gym environment.
action = env.action_space.sample()

#Otherwise:
#total_actions = ??
```

ϵ-Greedy Approach



Each value corresponds to the Q-value for a given action at a random state in an environment. The height of the light blue bars correspond to the probability of choosing a given action. The dark blue bar corresponds to a chosen action. To try an interactive version, go here.

Explanation*:* A simple combination of the greedy and random approaches yields one of the most used exploration strategies: *ϵ-greedy*. In this approach the agent chooses what it believes to be the optimal action most of the time, but occasionally acts randomly. This way the agent takes actions which it may not estimate to be ideal, but may provide new information to the agent. The ϵ in *ϵ-greedy* is an adjustable parameter which determines the probability of taking a random, rather than principled, action. Due to its simplicity and surprising power, this approach has become the defacto technique for most recent reinforcement learning algorithms, including DQN and its variants.

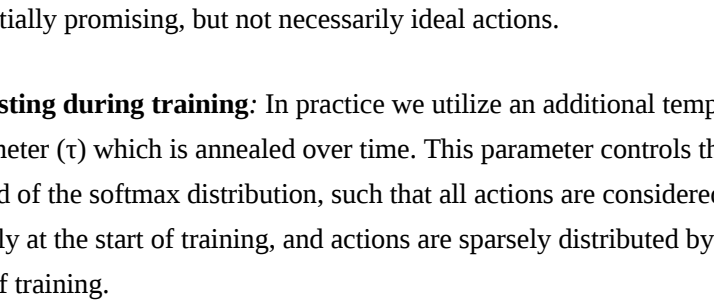
Adjusting during training*:* At the start of the training process the e value is often initialized to a large probability, to encourage exploration in the face of knowing little about the environment. The value is then annealed down to a small constant (often 0.1), as the agent is assumed to learn most of what it needs about the environment.

Shortcomings*:* Despite the prevalence of usage that it enjoys, this method is far from optimal, since it takes into account only whether actions are most rewarding or not.

Implementation*:*

```
e = 0.1
if np.random.rand(1) < e:
    action = env.action_space.sample()
else:
    Q_dist = sess.run(Q_out,feed_dict={inputs:[state]})
```

Boltzmann Approach



Each value corresponds to the Q-value for a given action at a random state in an environment. The height of the light blue bars correspond to the probability of choosing a given action. The dark blue bar corresponds to a chosen action. To try an interactive version, go here.

Explanation*:* In exploration, we would ideally like to exploit all the information present in the estimated Q-values produced by our network. *Boltzmann exploration* does just this. Instead of always taking the optimal action, or taking a random action, this approach involves choosing an action with weighted probabilities. To accomplish this we use a softmax over the networks estimates of value for each action. In this case the action which the agent estimates to be optimal is most likely (but is not guaranteed) to be chosen. The biggest advantage over e-greedy is that information about likely value of the other actions can also be taken into consideration. If there are 4 actions available to an agent, in e-greedy the 3 actions estimated to be non-optimal are all considered equally, but in Boltzmann exploration they are weighed by their relative value. This way the agent can ignore actions which it estimates to be largely sub-optimal and give more attention to potentially promising, but not necessarily ideal actions.

Adjusting during training*:* In practice we utilize an additional temperature parameter (τ) which is annealed over time. This parameter controls the spread of the softmax distribution, such that all actions are considered equally at the start of training, and actions are sparsely distributed by the end of training.

$$P_t(a) = \frac{\exp(q_t(a)/\tau)}{\sum_{i=1}^{n} \exp(q_t(i)/\tau)},$$

The Boltzmann softmax equation.

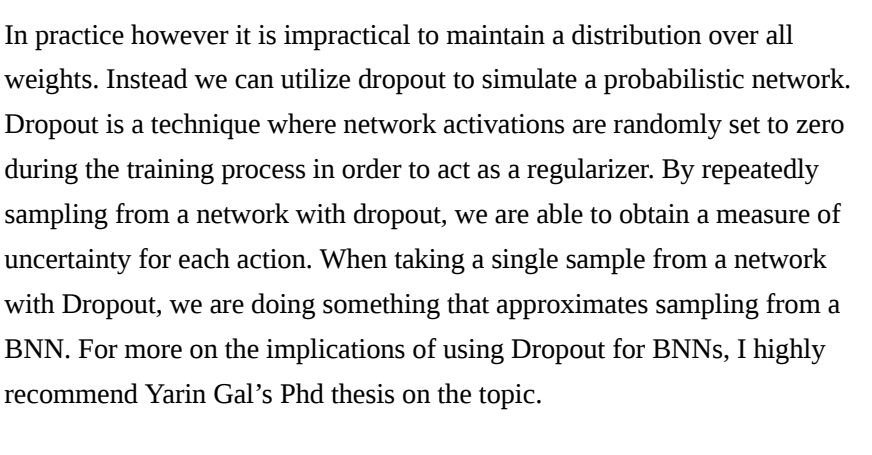
Shortcomings*:* The underlying assumption made in Boltzmann exploration is that the softmax over network outputs provides a measure of the agent's confidence in each action. If action 2 is 0.7 and action 1 is 0.2 the tempting interpretation is that the agent believes that action 2 is 70% likely to be optimal, whereas action 1 is 20% likely to be optimal. In reality this isn't the case. Instead what the agent is estimating is a measure of how optimal the agent thinks the action is, not how certain it is about that optimality. While this measure can be a useful proxy, it is not exactly what would best aid exploration. What we really want to understand is the agent's uncertainty about the value of different actions.

Implementation*:*

```
#Add this to network to compute Boltzmann probabilities
Temp = tf.placeholder(shape=None,dtype=tf.float32)
Q_dist = slim.softmax(Q_out/Temp)

#Use this for action selection.
t = 0.5
Q_probs = sess.run(Q_dist,feed_dict={inputs:[state],Tem
```

Bayesian Approaches (w/ Dropout)



Each value corresponds to the Q-value for a given action at a random state in an environment. The height of the light blue bars correspond to the probability of choosing a given action. The dark blue bar corresponds to a chosen action. Additionally each change in value corresponds to a new sampling from a Bayesian Neural Network using dropout. To try an interactive version, go here.

Explanation*:* What if an agent could exploit its own uncertainty about its actions? This is exactly the ability that a class of neural network models referred to as Bayesian Neural Networks (BNNs) provide. Unlike traditional neural network which act deterministically, BNNs act probabilistically. This means that instead of having a single set of fixed weights, a BNN maintains a probability distribution over possible weights. In a reinforcement learning setting, the distribution over weight values allows us to obtain distributions over actions as well. The variance of this distribution provides us an estimate of the agent's uncertainty about each action.

In practice however it is impractical to maintain a distribution over all weights. Instead we can utilize dropout to simulate a probabilistic network. Dropout is a technique where network activations are randomly set to zero during the training process in order to act as a regularizer. By repeatedly sampling from a network with dropout, we are able to obtain a measure of uncertainty for each action. When taking a single sample from a network with Dropout, we are doing something that approximates sampling from a BNN. For more on the implications of using Dropout for BNNs, I highly recommend Yarin Gal's Phd thesis on the topic.

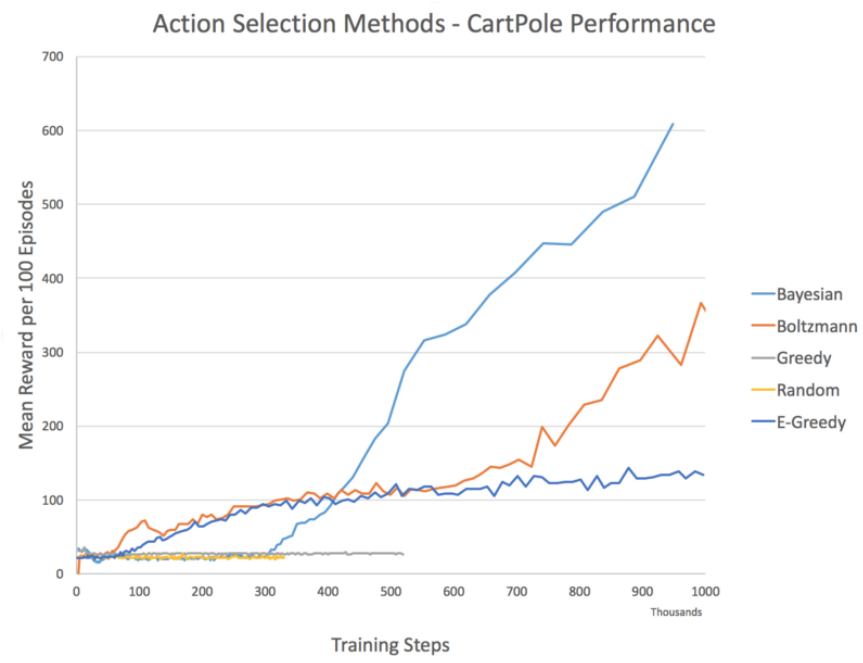
Shortcomings*:* In order to get true uncertainty estimates, multiple samples are required, thus increasing computational complexity. In my own experiments however I have found it sufficient to sample only once, and use the noisy estimates provided by the network. In order to reduce the noise in the estimate, the dropout keep probability is simply annealed over time from 0.1 to 1.0.

Implementation*:*

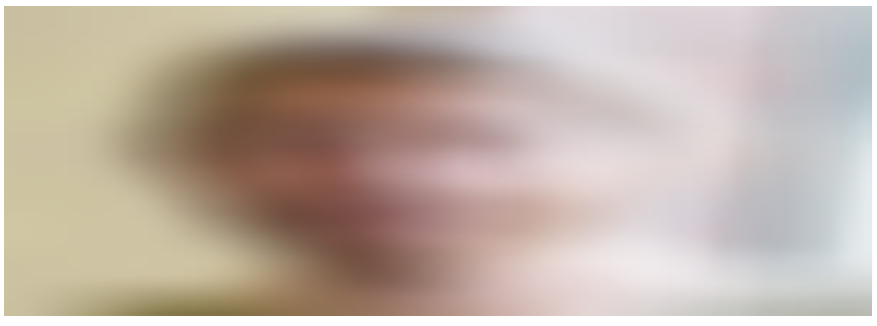
```
#Add to network
keep_per = tf.placeholder(shape=None,dtype=tf.float32)
hidden = slim.dropout(hidden,keep_per)


keep = 0.5
```

Comparison & Full Code



I compared each of the approaches using a DQN trained on the CartPole environment available in the OpenAI gym. The Bayesian Dropout and Boltzmann methods proved most helpful, at least in my experiment. I encourage those interested to play around with the hyperparameters, as I am sure better performance can be gained from doing so. Indeed, different approaches may be best depending on what hyperparameters are used. Below is the full implementation of each method in Tensorflow:



Advanced Approaches

All of the methods discussed above deal with the selection of actions. There is another approach to exploration that deals with the nature of the reward signal itself. These approaches fall under the umbrella of intrinsic motivation, and there has been a lot of great work in this area. In a future post I will be exploring these approaches in more depth, but for those interested, here is a small selection of notable recent papers on the topic:

Variational Information Maximizing Exploration

Incentivizing Exploration In Reinforcement Learning With Deep Predictive Models

Unifying Count-Based Exploration and Intrinsic Motivation

Hierarchical Deep Reinforcement Learning: Integrating Temporal Abstraction and Intrinsic Motivation

. . .

I hope that this survey of approaches has been helpful for those interested in learning how to improve exploration in their RL agents!

. . .

If this post has been valuable to you, please consider *donating* to help support future tutorials, articles, and implementations. Any contribution is greatly appreciated!

If you'd like to follow my work on Deep Learning, AI, and Cognitive Science, follow me on Medium @Arthur Juliani, or on twitter @awjliani.

. . .

More from my Simple Reinforcement Learning with Tensorflow series:

1. *Part 0—Q-Learning Agents*
2. *Part 1—Two-Armed Bandit*
3. *Part 1.5—Contextual Bandits*
4. *Part 2—Policy-Based Agents*
5. *Part 3—Model-Based RL*
6. *Part 4—Deep Q-Networks and Beyond*
7. *Part 5—Visualizing an Agent's Thoughts and Actions*
8. *Part 6—Partial Observability and Deep Recurrent Q-Networks*
9. **Part 7—Action-Selection Strategies for Exploration**
10. *Part 8—Asynchronous Actor-Critic Agents (A3C)*