Baby's First Reinforcement Learning

Brandon Foltz

Abstract

I review a pair of Reinforcement Learning algorithms for training a Markov Decision Process, as well as a Neural Network emulating a discrete Markov Decision Process. The training and application of these algorithms is demonstrated in a minimalistic environment in which an agent can navigate to collect a reward repeatedly. The algorithms will be evaluated on their training and performance characteristics. In addition, conceptual comparisons with more popular algorithms will be provided.

Introduction

Finding powerful learning algorithms for controlling agents to navigate complex environments has been a long standing goal of AI researchers and remains an open research problem. Markov Decision Processes[4] were developed in the 1950s and proved to be very useful in the study of optimization problems as well as reinforcement learning problems.

Related Work

A well known recent work by the DeepMind Technologies group entitled *Playing Atari with Deep Reinforcement Learning*[3] addresses the problem of training an agent to navigate a high dimensional environment directly from the pixel output of Atari games. Their approach was a variation on the well-known Q-Learning[2] algorithm where convolutional neural networks were used to filter the pixel inputs and provide useful features that the rest of the algorithm could learn from. Additionally the authors made use of a technique called "experience replay" to reduce or eliminate training problems caused by the temporal correlations between recent actions and target values.

My work in this paper is largely inspired by this work both from a motivational perspective as well as regarding technical details I implemented in my own algorithms. Specifically, I will use the same "experience replay" technique to stabilize the training of my own algorithm.

Training Discrete Markov Decision Process

Markov Decision Processes can be thought of as discrete functions mapping states to actions according to some probability mapping, also known as the *policy*. We can formalize this as follows:

$S :=$ set of finite states
$A :=$ set of finite actions
$P_a(s, s') = P(s_{t+1} = s' \mid s_t = s, s_t = a)$ is the probability that an action a in state s at time t will leard to state s' at time $t+1$
$R_a(s, s')$ is the reward received after transitioning from state s to s' due to action a

In the case of discrete Markov Decision Processes, this can be both visualized and represented as a table mapping states to actions according to probabilities contained within the table.

	Action 1	Action 2
State 1	0.1	0.9
State 2	0.5	0.5

My work explores a technique for finding this mapping in a simplified environment which is ideal for learning an experimentation by the unexperienced reinforcement learning developer. The environment in which this experimentation takes place is a finite grid (Illustration 1), where an agent can move in one of four cardinal directions at each time step. The location of the reward is fixed until the agent manages to navigate to the same location as the reward, at which point the reward is moved to a new random location.

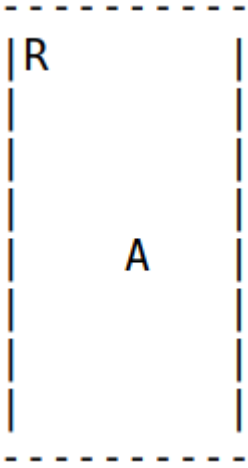

Illustration 1: Visualization of the environment where the agent "A" can navigate to collect a reward "R".

In contrast to the approach taken by DeepMind Technologies, I do not use the direct pixel values from the environment to train my agent. Instead, I have chosen to represent the "state" of the environment as a vector indicating the direction of the reward with respect to the agent. This process of deciding a useful representation of the state from which to learn an ideal policy is conceptually similar to the "feature mapping" step that a developer might use when working on a traditional supervised learning problem. Ideally this step could be skipped and the algorithm could learn directly from sensory input, but this is difficult and requires much more advanced techniques than those that might be fully understandable by a beginner. Hopefully the choice of state representation chosen (Illustration 2) is an obvious one, although it is certainly not the only one which could work well.

10000000	Up and left
01000000	Up
00100000	Up and right
00010000	Right
00001000	Down and right
00000100	Down
00000010	Down and left
00000001	Left

Illustration 2: State representation, or "feature mapping" of the agent-environment system.

The algorithm I chose to use to train the "vanilla" discrete Markov Decision Process comes from a family of algorithms used to solve multi-armed bandit[5] referred to as an "Epsilon-decreasing strategy". At the start of training, all state→action pairs are given the same weight and we assume that the agent does not have an optimal policy with this initialization. Thus the agent must perform some amount of "exploration" in order to gain experience and see what actions lead to a reward. The most straight-forward way to implement this is by choosing a random action at each time step according to some probability defined by the hyperparameter ε. Eventually the agent will collect a reward, and the actions which led to the collecting of this reward are given a higher weight (probability of occurring) for future time steps. Over the course of training, we

decrease the value of ε such that fewer actions are taken at random and more actions are taken from the thus-far learned policy. Pseudo-code for this algorithm follows:

```
epsilon = initial_epsilon = 1

for i in range(0, training_steps):

    state, action, reward = act_and_observe_environment(environment, markov_decision_process, epsilon)

    history.append({"state": state, "action": action})

    while history.length > max_history_length:

        history.remove_oldest()

    if reward > 0:

        for state, action in history:

            markov_decision_process.encourage(state, action)

    #anneal epsilon

    epsilon_lerp = initial_epsilon / training_steps

    epsilon = epsilon - epsilon_lerp
```

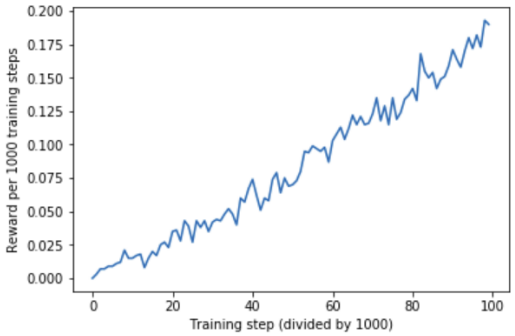

Illustration 3: Learning curve for the MDP training algorithm denoting average reward per 1000 training steps.

As can be seen in the learning curve to the right (Illustration 3), this algorithm is effective at increasing the average reward per training step gained by the agent as training progresses.

While this algorithm works well for the simple environment used in this experiment, it is not difficult to imagine situations where this algorithm would struggle or fail outright to train in a practical amount of time. Consider an environment where there are 10^9 unique states, and 4 actions associated with each state. In this scenario, there are $4x10^9$ unique state→action pairs and the agent must *randomly* navigate to a significant portion of them for a vanilla MDP to begin to learn a good policy. This can take quite a lot of time, and this is still assuming there are a finite number of discrete states! Clearly we do no have infinite memory to store all possible state→action pairs, so this algorithm will not work at all.

Training Neural Network to Emulate a Markov Decision Process

Two immediate advantages we can see to using a Neural Network in place of a discrete Markov Decision Process are that a Neural Network requires only a constant amount of memory, and they take continuous (real-valued) inputs. This (at least in theory) allows us to use them to learn in a state space consisting of an infinite number of states. Make no mistake, Neural Networks learn an *approximation* of an arbitrary function. Therefore they still cannot map every possible state→action pair possible in a finite amount of memory. However they have shown great success in learning *nearly* correct mappings over large input→output spaces.

To keep things simple, I'll continue to use the same environment and state representations. There will only be small changes at this point in time; instead of a discrete MDP being used to choose actions there will be a Neural Network. Additionally instead of updating the probability values in the MDP state→action table to "encourage" certain state→action pairs, back-propogation will be used as is

typical for training a Neural Network. There is no need to copy the pseudocode again since the structure is the same. How does this new formulation perform?

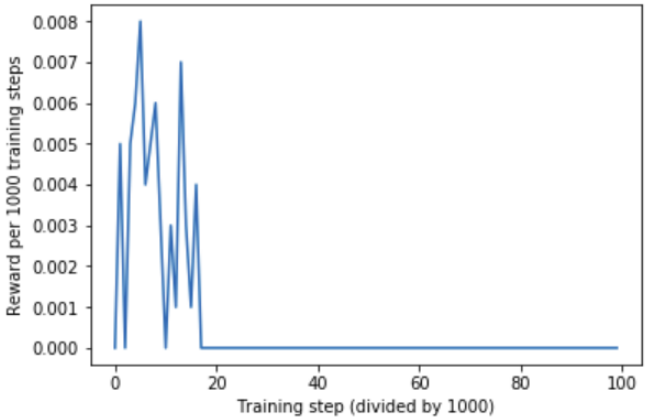

Illustration 4: Learning curve for Neural Network trained with the same algorithm as the discrete MDP.

By examining the learning curve seen at the left, it is clear that this algorithm is not even remotely successful at training the Neural Network to perform the same function as the Markov Decision Process. Some rewards were gained early in training, but these can be attributed primarily to random chance since epsilon is near 1.0 (completely random actions) at the start of training.

It is not immediately clear why this training algorithm fails. Intuitively it seems that it should work trivially, since the network is capable of performing the same function as the MDP and certain outputs can be "encouraged" in a similar manner. This of course ignores the added complexity of Neural Networks. Encouraging one state→action pair in an MDP is completely independent from all other state→action pairs, however with a Neural Network this is not so. Additionally, choice of hyper-parameters such as learning rate, optimization algorithm (Stochastic Gradient Descent, Adam, RMSProp, etc), number of training iterations, can have a huge effect on the training characteristics of the Neural Network. Thus, the failure to train could be attributed to any number of these new variables.

Improving the Neural Network MDP Training Algorithm

Since my first attempt at replacing the MDP with a Neural Network did not succeed, I began troubleshooting the algorithm. My first step of troubleshooting was to verify that my network as formulated could in fact learn the necessary mapping directly. The network I constructed for these experiments is a 1-hidden layer network with 8 input nodes, 32 hidden nodes, 4 output nodes, and sigmoid activations on the hidden nodes and output nodes. I trained the network for 8000 back-propogation steps against a hand-made mapping of state-action pairs. As can be seen in the learning curve to the right, this training went exceptionally smoothly with very quick convergence to a low loss value.

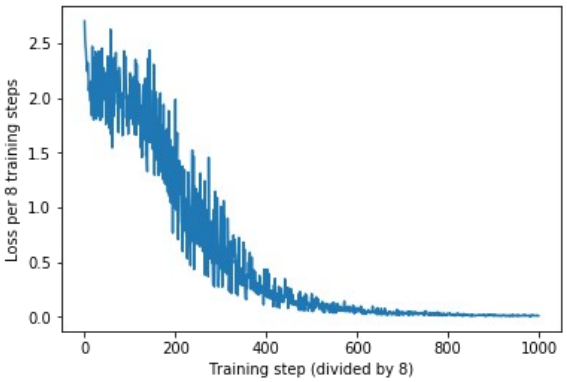

Illustration 5: Learning curve for "test case" network trained directly on known policy.

This made it clear that the problem was with my technique for training the algorithm from the agent's experiences in the environment. Looking back at Illustration 4, we can note that the network received fewer than 2000 back-propogation steps since that original algorithm only back-propogates when a reward is received, and only reinforces the most recent successful sequence of state→action pairs.

Clearly more “experience” is needed. However, I tried training the network with one-million and one-billion training steps yet this yielded very much the same result while taking quite a long time to complete. This led me to believe that there was something fundamentally wrong with my algorithm.

Having previously read the paper “Playing Atari with Deep Reinforcement Learning”[3] I was reminded of the “experience replay” technique the authors used to stabilize the training of their algorithm. In essence, when a successful state→action pair is discovered, it is kept in memory to be re-used for many future back-propogation steps. This is a conceptually very simple addition, and adding it to my algorithm allowed it to train successfully (after some hyper-parameter tuning). The new algorithm (pseudo-code) for training the Neural Network to emulate the Markov Decision Process is as follows:

```
epsilon = initial_epsilon = 1

for i in range(0, training_steps):

    state, action_predicted, action_taken, reward = act_and_observe_environment(environment,
neural_mdp, epsilon)

    history.append((state, action_predicted, action_taken, reward))

    while history.length > max_history_length:

        history.remove_oldest()

    if reward > 0:

        replay_memory.append(history.copy())

        while replay_memory.length > max_replay_memory:

            replay_memory.remove_oldest()

    if replay_memory.length > 0:

        for j in range(0,100):

            successful_sequence =
select_random_history_from(replay_memory)

            h = select_random_state_action_transition_from(successful_sequence)

            loss = criteria(neural_mdp(h["state"]), h["action_taken"])

            loss.back_propogate()

            optimizer.update_weights()

    if anneal_epsilon:

        epsilon = initial_epsilon - (i / training_steps)
```

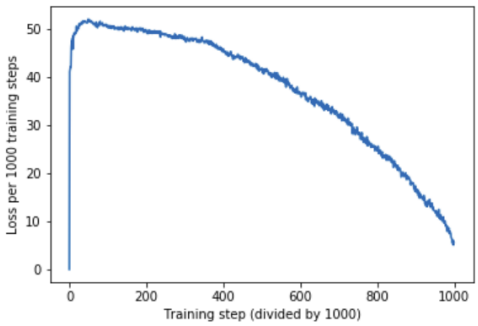

Illustration 6: Learning curve for "experience replay" modified training algorithm.

Examine Illustration 6 for the learning curve produced by this algorithm. The initial upward spike of loss value from zero is due to the fact that at the start of training, no successful sequences have been found and so the loss remains initialized at zero. As soon as at least one successful sequence is found, the actual training loss can be calculated. As training progresses, the speed of convergence increases due to the improving quality of sequences being sampled from the experience replay memory.

Furthermore, examine the curve of average reward gained per step over the course of training. You may note that it looks extremely reminiscent of the same graph obtained by training the discrete Markov Decision Process (Illustration 3).

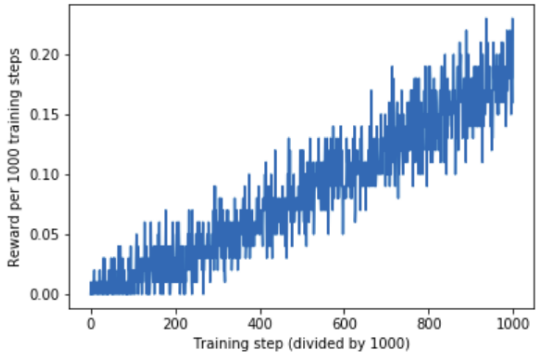

Illustration 7: Average reward per training step of Neural Network trained with "experience replay".

Future Work

Going forward in future experiments, I would like to explore more recent state-of-the-art reinforcement learning techniques. The ones covered here are not particularly novel, and have significant disadvantages in more complex environments. Newer techniques in reinforcement learning focus much more on learning to “plan” effectively rather than merely repeating a successful history. There is significant knowledge to be gained from these techniques and it would be a good use of time to review them.

Conclusion

In this paper I have shown two Reinforcement Learning algorithms for training an agent to navigate a simple environment. Each algorithm focused on using a different Machine Learning tool, first Markov Decision Processes and then Neural Networks. Hopefully this has provided sufficient background and motivation for someone else to begin experimenting in the field of Reinforcement Learning.

References

[1] Markov Decision Process : https://en.wikipedia.org/wiki/Markov_decision_process

[2] Q-Learning : https://en.wikipedia.org/wiki/Q-learning

[3] Mnih, Volodymyr, et al. "Playing atari with deep reinforcement learning." arXiv preprint arXiv:1312.5602 (2013).

[4] Bellman, Richard. "A Markovian decision process." Journal of Mathematics and Mechanics (1957): 679-684.

[5] Multi-armed bandit problems : https://en.wikipedia.org/wiki/Multi-armed_bandit