Learning Super Mario Bros. with Double Q-learning

Jeffrey Sheldon
`jeffsh@uw.edu`

Abstract

Deep Q-learning is a technique previously applied to enabling an agent to play various Atari games. While capable of training an agent to play Atari games to various degrees of success, it is prone to overestimation of rewards as the neural network matures. Double Q-learning corrects this behavior by applying a second neural network to the task of future reward estimation, thus inducing the neural network to remain within a reasonable estimated range of the true reward. I have applied this technique to an agent that has learned to play Super Mario Bros. in a reinforcement learning scenario. The use of a neural network as an estimation of the traditional Q function allows the agent to accurately value all possible actions, given a game state.

1 Introduction

Traditional Q-learning requires an exhaustive search of a state space by an agent, in order to enable the agent to optimally exploit the environment. The exact nature of every state-action pair must be explored and the resulting reward stored in the agent's memory, so that the agent may directly reference that knowledge when it encounters that same state again.

The environment of Super Mario Bros. is sufficiently large so that Q-learning is not an efficient solution. To correct this, the Q function is replaced by $\hat{Q}$, an estimation of Q.The function $\hat{Q}$ is implemented using a convolutional neural network. This estimation of Q allows an agent to value the characteristics of the game state, rather than the state itself.

The framework of the Q-learning algorithm remains in place. The agent is presented with a state, from which it must infer the reward associated with taking each action from that state. The agent then takes the action associated with the highest reward, and the result is used to update the agent's knowledge.

The goal of applying this approach is to produce an agent capable of optimal and generalized performance across all stages in Super Mario Bros.

2 Q-learning and its limitations

Standard Q-learning applies an iterative approach to learning the true Q value of a state-action pair. The Bellman Equation forms the basis for Q-learning, wherein a concrete function Q is learned for a given environment.

$$Q(s,a) \leftarrow (1-\alpha)Q(s,a) + \alpha\left[R(s,a,s') + \gamma \max_{a'} Q(s',a')\right]$$

An agent is presented with an environment to explore. The agent updates its knowledge of the environment as it moves through it and observes rewards.

The Q-learning algorithm will converge to the optimal policy. But to do so, the agent must perform a complete exploration of all possible state-action pairs. As the dimensionality of an environment grows, the complexity of the problem increases exponentially, thus making this an untenable solution for large environments. Additionally, this method will not generalize to unseen environments.

3 Estimating Q with a deep neural network

Rather than learning the true value of Q, we will apply a neural network to estimate the value of Q. This will allow an agent to learn to value the features that describe a state, rather than the state itself, thus allowing the agent to generalize its policy to previously unseen states. Feature definitions will be learned by the network. The function will be non-linear, allowing it to define a more complex space.

`Playing Atari with Deep Reinforcement Learning` [1] proposed applying a convolutional neural network for the application of playing Atari games. The abstracted state is the rendered output of a game at a discrete temporal step. By applying a convolutional neural network, convolutional filters are learned by the network to identify the features within the rendered image that the network believes important to the task of inferring the subsequent action to take. A version of the Bellman Equation is used to train the network:

$$Q(s,a) \leftarrow R(s,a,s') + \gamma \max_{a'} Q(s',a')$$

The goal of the algorithm is to train the function Q, so that it may infer the appropriate action to take, given state. Because Q is also a function of the value used to optimize Q, the algorithm chases itself and thus leads to exploding reward definitions.

`Deep Reinforcement Learning with Double Q-learning` [2] remedies this by providing the algorithm with two instances of the function Q: a *Q estimator* and *Q target*. We apply this technique to learning Super Mario Bros.

Given two neural networks, Q_{est} and Q_{target}, we may determine the inferred action by our estimated function of Q and a targeted function of Q, which we treat as the supervised label. Comparing the two values, we may compute loss and thus update Q_{est} accordingly. Rather than optimizing the Q function as a function of itself, we provide a target value that remains somewhat constant, for the Q function to optimize toward.

$$\hat{Q} \leftarrow \max_{a} Q_{est}(s,a)$$

$$Q \leftarrow R(s,a,s') + \gamma Q_{target}(s', \arg\max_{a'} Q_{est}(s',a'))$$

$$loss(\hat{Q}, Q)$$

Plainly said, Q_{est} is asked to learn to appropriately value the transition from the current state to the subsequent state, given an action, capturing both the immediate reward and the forward-looking decaying reward. Q_{target} provides the value of the forward-looking reward for loss. Of note, the selection of action used for Q_{target} is the responsibility of Q_{est}. That is, Q_{est} defines the policy to follow. Q_{target} determines the value of that policy.

3.1 Model structure

The neural network used is a convolutional neural network, like that used in prior papers. As input, it receives a four-channel, 84x84 image. Each channel is a gray-scale 84x84 image. Four images are captured at subsequent time steps and layered on top of one another to form the four-channel image.

The network is composed of three convolutional layers. The first convolves the image with 32 8x8 kernels with a stride of 4. The second layers convolves with 64 4x4 kernels with a stride of 2. The third layer convolves with 64 3x3 kernels. Each layer's output is activated with a rectified linear unit.

The convolved and flattened output is the input to a fully connected layer of 512 units, activated by a rectified linear unit. A final fully connected layer outputs a regression for every action the agent can perform. Normalizing the set of outputs using a softmax produces a set of probabilities for each action. The maximum probability denotes the action the network believes should be performed.

3.2 Training methodology and framework

The goal of this experiment is to produce an agent capable of playing Super Mario Bros. The agent is presented with a Super Mario Bros. environment in which it is allowed to play a randomly drawn episode to completion. Completion is episode success or failure. The episode is navigated by requesting Q_{est} to infer the action to take at each discrete temporal steps. The agent observes the prior state, the action taken, the subsequent reward, and the subsequent state. These properties are stored as a tuple in a replay memory buffer.

Each reward observed by the agent consists of three properties: pixels moved on the X plane; seconds passed on the game clock; and Mario's death. Super Mario Bros. is a simple game, where success can be achieved by moving to the right as quickly as is possible without dying.

An ϵ greedy method is used to select the actions taken during an episode. Initially, the agent is almost certain to take a random action, rather than the inferred optimal action. Over time, ϵ decays and the agent will rely more on its own inference. This allows the agent to explore the environment more initially, and later rely on its own knowledge.

Following each step in the episode, a random batch of samples are retrieved from the buffer and used to optimize the model. This approach induces variability in the number of training iterations per episode. A poorly optimized agent may believe that the best course of action is to do nothing and allow the game clock to countdown. As a result, this agent will allow more training iterations for the episode than would an agent who believe the best policy is to run to the right, directly into an enemy.

The parameters of Q_{target} are updated with the parameters of Q_{est} every 10,000 training iterations (not episodes). This allows Q_{target} to provide a fairly consistent target for Q_{est}, while still periodically being improved.

Algorithm 1 Training algorithm

```
$Q_{est} \leftarrow CNN()$
$Q_{target} \leftarrow CNN(Q_{est}.parameters)$
$buffer \leftarrow MemoryBuffer(batchSize)$
$\epsilon \leftarrow 1.0$
for episode in numEpisodes do
    $env, s_t \leftarrow initializeEnvToRandomLevel()$
    while env not done do
        $a \leftarrow \arg\max_a Q_{est}(s_t, a, \epsilon)$
        $r_t, s_{t+1}, done \leftarrow env.step(a_t)$
        $buffer.store(s_t, a_t, r_t, s_{t+1}, done)$
        $batch \leftarrow buffer.sample()$
        $\hat{Q} \leftarrow Q_{est}(batch.s_t, batch.a_t)$
        if $batch.done$ then
            $Q \leftarrow batch.r_t$
        else
            $Q \leftarrow batch.r_t + \gamma Q_{target}(batch.s_{t+1}, \arg\max_a Q_{est}(batch.s_{t+1}, a)$
        end if
        $loss = SmoothL1Loss(\hat{Q}, Q)$
        $AdamW.optimize(Q_{est})$
        $Q_{target} \leftarrow Q_{est}.parameters$ every 10,000 iterations
        $s_t \leftarrow s_{t+1}$
    end while
    $\epsilon \leftarrow max(1.0, 0.99998\epsilon)$
end for
```

3.2.1 Hyperparameters

The model was trained for 175,000 episodes. Each action selected by the agent was repeated for 4 time steps. The memory buffer stores 100,000 memories at a time. As new memories are inserted, the oldest memories are purged. ϵ is initialized to 1.0 and decays at a rate of 0.99998ϵ per episode, to a minimum of 0.1. γ is 0.95. The loss function is *Smooth L1 Loss*, optimization is *AdamW* with a learning rate of 0.00025. Training batches are 64 samples.

The agent is given access to the following actions: no action; right, right + jump, right + run, right + run + jump. Of the 32 stages available in Super Mario Bros., two stages use a different physics model while Mario is forced to navigate an underwater environment. These underwater stages were omitted from training. The current-optimal agent is tested and evaluated against stage 8-1 every 10 episodes.

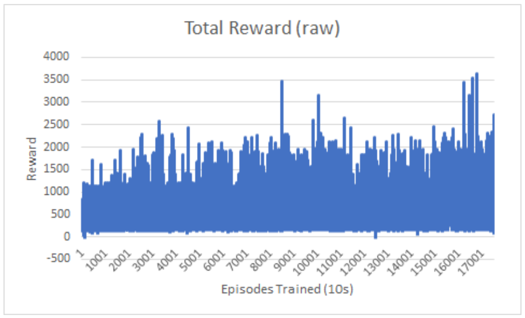
4 Experiments and results

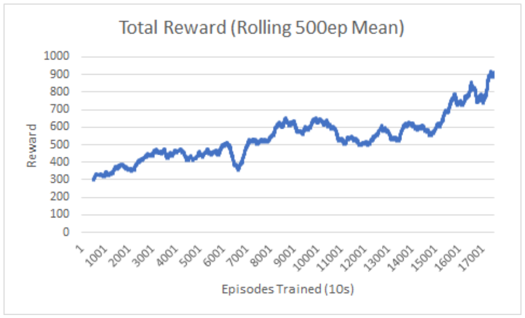
Every 10 episodes played, the agent is tested against stage 8-1. This stage tests the player against most of the obstacles available in the game. Every action performed is inferred by the agent, ϵ greedy is not used to select actions.

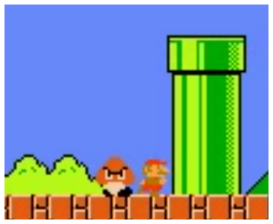
The agent is evaluated on the following data points: total reward gained; total horizontal movement (in pixels); time spent in the environment; success or failure.

Total reward accumulated is used as the most important measurement of the agent's ability. When the goal of the experiment and the environment are abstracted away, the task of the training algorithm is to minimize loss between estimated and observed reward values, so that the agent may select the optimal action at a time step, thus maximizing its reward. Given this abstraction, an agent capable of producing higher reward accumulations is desired.

Initial analysis of accumulated reward was challenging. The agent's tested reward accumulation varied significantly between episodes. By instead using a rolling 500-episode mean of total reward, a clearer indication of the model's performance becomes evident. That trajectory indicates that not only is the model improving, it could benefit from more training episodes.







The horizontal movement evaluation follows reward closely. Time spent in each episode is useful only in identifying models that tend to get stuck behind obstacles. For each stage, Mario is given 400 seconds to complete it. Identifying time=400 in the metrics typically indicates an agent that is not capable of clearing some obstacles. Tall obstacles and enemies moving right, rather than left, are frequent challenges that the agent fails to overcome.

While the agents trained have shown success in navigating the environment, no agent produced is capable of completing a stage. Qualitatively, later agents have shown a tendency to navigate the environment quickly and consistently move to the right. By holding the B button on the Nintendo Entertainment System controller, Mario will run while moving left or right. Later models tend to hold B while moving to the right the majority of the time.

5 Conclusion

The application of Double Q-learning with a deep convolutional neural network to training an agent to play Super Mario Bros. has shown to be a successful approach. The use of a neural network to estimate the value of Q has proven robust in valuing Q across a variety of states. Future experiments would consider allowing the model more episodes to learn from, a larger replay buffer, and a reduced epsilon decay, with a higher minimum threshold.

References

[1] Mnih, V., Kavukcuoglu, K., Silver, D., Graves, A., Antonoglou, I., Wierstra, D. & Riedmiller, M. (2013) *Playing Atari with Deep Reinforcement Learning*, `https://arxiv.org/pdf/1312.5602.pdf`

[2] van Hasselt, H., Guez, A. & Silver, D. (2015) *Deep Reinforcement Learning with Double Q-learning*, `https://arxiv.org/pdf/1509.06461.pdf`

A Appendix

Project source code
`https://github.com/j-confusatron/SuperMarioBot`

Agent Demonstrations
`https://www.youtube.com/watch?v=vmoiv1MUzds`
`https://www.youtube.com/watch?v=gQ-Yjdk4kuQ` [Note: this agent was trained separately, only on 1-1, and only allowed to move right and jump]

Python NES Implementations
Nes-py `https://pypi.org/project/nes-py/`
Super Mario Bros. Gym `https://github.com/Kautenja/gym-super-mario-bros`

Recurrent DQN approach that was initially considered
`https://arxiv.org/pdf/1704.07978v5.pdf`

DQN implementation examples studied for this project
`https://mlpeschl.com/post/tiny_adrqn/`
`https://github.com/blackredscarf/pytorch-DQN`