Exploring reinforcement learning in the Atari environment: A survey

Seoul National University Artificial Intelligence Institute BioIntelligence Lab
Mentor: Junseok Park
Faculty Advisor: Dr. Byung-tak Zhang

Sungwook Min

Table of Contents

- Introduction
- Q-Learning
- Deep Q-Network (DQN)
- OpenAI Gymnasium
- Cartpole
  - Method
  - Result and Analysis
- Atari
- Pong Game
  - Method
  - Result and Analysis
- Conclusion
- Extension
- References

Introduction

Reinforcement Learning (RL) is based on the reward (R_t) hypothesis and the *agent* is trained to learn an optimal policy that maximizes the reward function.
R_t indicates how well the agent is doing at step t.
At each step t the agent:

- Executes action A_t
- Receives observation O_t
- Receives scalar reward R_t

Environment:

- Receives action A_t
- Outputs updated observation O_{t+1}
- Outputs updated scalar reward R_{t+1}

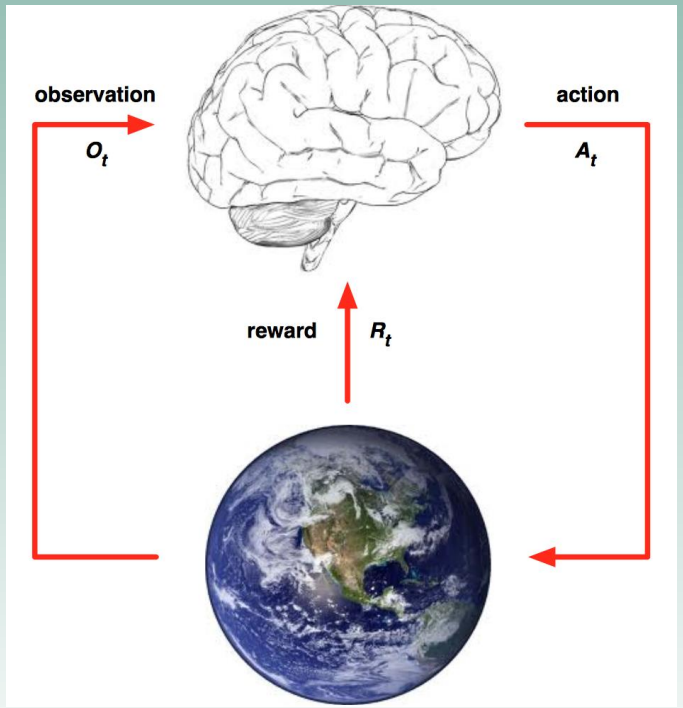


Figure 1: Illustration of RL process [1]

Introduction (cont'd)

History is the complete sequence of observations, actions, rewards

$$H_t = O_1, R_1, A_1, \dots A_{t-1}, O_t, R_t$$

State is the information used to determine what happens next

$$S_t = f(H_t)$$

An RL agent include these components:

- Policy: agent's behavior function
- Value function: how good each state is, is a prediction of future reward

$$v_\pi(s) = \mathbb{E}[R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \cdots | S_t = s]$$

- Model: agent's representation of the environment, predicts what the environment will do next

- P predicts the next state

$$P^{a'}_{ss} = \mathbb{P}[S_{t+1} = s' | S_t = s, A_t = a]$$

- R predicts the next reward

$$R^a_s = \mathbb{E}[R_{t+1} | S_t = s, A_t = a]$$

Q-Learning

Q-learning is a model-free algorithm
Objective is in a Factored Markov Decision Process (FMDP), the agent learns the optimal policy for performing a specific action in a particular step t, where the probable reward earned at the end of the process is maximized.
$\therefore$ Q stands for quality of the reward occurred from the particular action performed by the agent.
Q-learning employs the Q-value function which calculates the value of the action performed in (state S and action A.

$$Q_\pi(s,a) = E_\pi[R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \cdots | S_t = s, A_t = a]$$

Here the function uses a discount factor γ between 0-1 which accounts for how significant the current reward is compared to the future reward.

Deep Q-Network (DQN)

DQN is simply an application of the Q-function to DNN
Structure consists of:

- CNN (convolutional neural network)
- experience relay
- target network

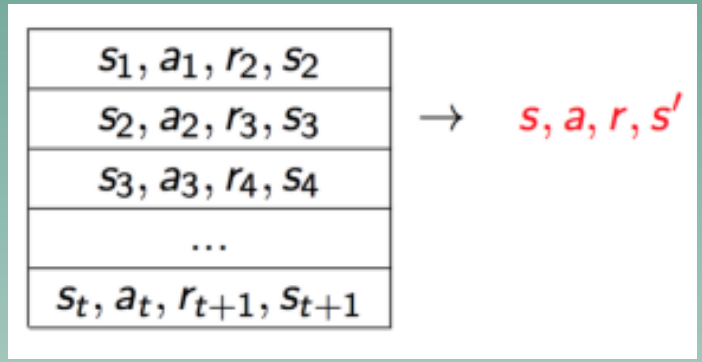


Figure 2: Demonstration of replay buffer [3]

Fixed Q-targets

$$\triangle w = \alpha(r + \gamma max\hat{Q}(s', a'; w^-) - \hat{Q}(s, a; w))\nabla_w \hat{Q}(s, a; w)$$

Equation above shows for every n steps, $w^- \leftarrow w$, which improves stability and prevents w from increasing exponentially.
With fixed Q-targets and experience relay, DQN improves stability.

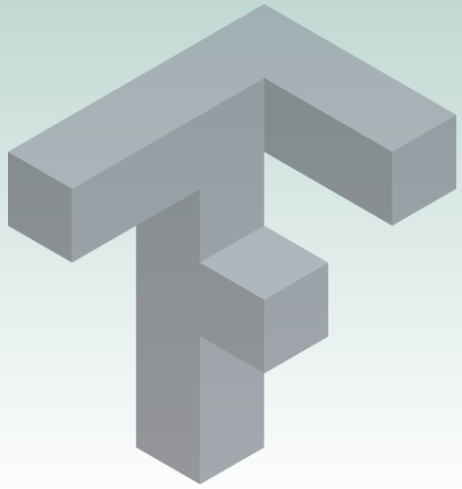
Deep Q-Network (DQN)

DQN Pseudocode

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 ϵ 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

Figure 3: DQN Pseudocode [4]



OpenAI Gymnasium

OpenAI Gymnasium is an open-source toolkit for developing RL algorithms and features a variety of different environments ranging from simple classic-control tasks to more complex simulations.

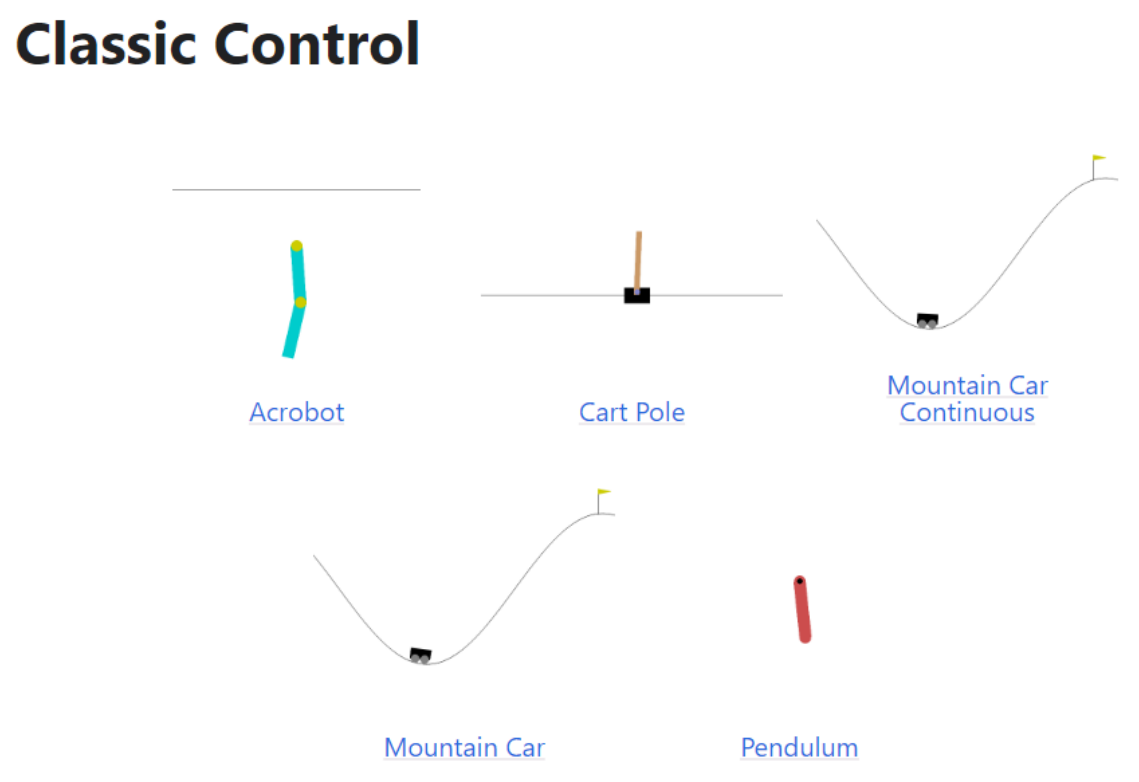


Figure 4: Gymnasium Classic Control

Cartpole

A pole is attached by an un-actuated joint to a cart, which moves along a frictionless track. The pendulum is placed upright on the cart and the goal is to balance the pole by applying forces in the left and right direction on the cart.

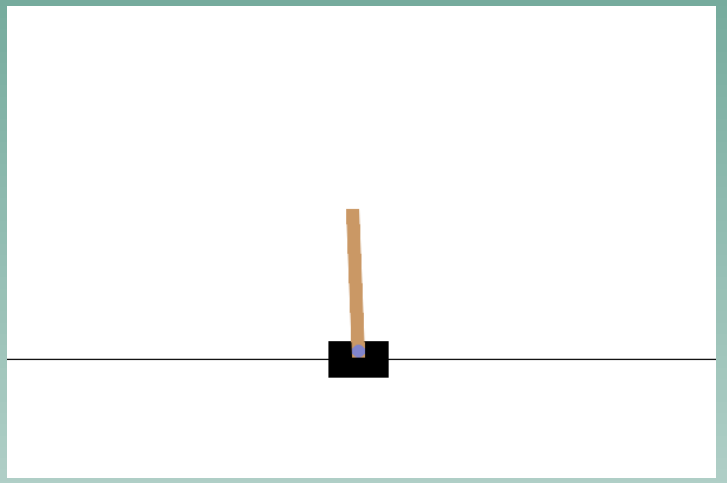


Figure 5: Gymnasium CartPole

Since the goal is to keep the pole upright for as long as possible, a reward of +1 for every step taken, including the termination step, is allotted. The threshold for rewards is 500 for v1 and 200 for v0.

The episode ends if any one of the following occurs:

- Termination: Pole Angle is greater than ±12°
- Termination: Cart Position is greater than ±2.4 (center of the cart reaches the edge of the display)
- Truncation: Episode length is greater than 500 (200 for v0)

Method

Hyperparameters:

- 50 episodes
- exploration rate: (0.05, 0.9)
  - set upper-epsilon to 0.9 to maximize exploration at early stages
- epsilon_decay = 200
- gamma = 0.8
  - discount rate
- learning_rate = 0.001
- batch_size = 64

*all other rewards, presets, and environmental factors were kept same as the original ‘CartPole-v0’ environment in gymnasium

DQN Agent – Learn Function

```
def learn(self):
    if len(self.memory) < BATCH_SIZE:#if no_of_episode_stored_in_memory < batch_size skip learning
        return
    #learning start
    batch = random.sample(self.memory, BATCH_SIZE)
    states, actions, rewards, next_states = zip(*batch)

    #list to tensor form
    states = torch.cat(states)
    actions = torch.cat(actions)
    rewards = torch.cat(rewards)
    next_states = torch.cat(next_states)

    #gather q-value in current state to current_q
    current_q = self.model(states).gather(1, actions)

    max_next_q = self.model(next_states).detach().max(1)[0] #calculation of proceeding action's quality
    expected_q = rewards + (GAMMA * max_next_q)# rewards + future value

    #, MSE_loss for error calculation, neural network learning
    loss = F.mse_loss(current_q.squeeze(), expected_q)
    self.optimizer.zero_grad()
    loss.backward()
    self.optimizer.step()
```

Figure 6: DQN Learn function

Result

Parameter: 50 episodes

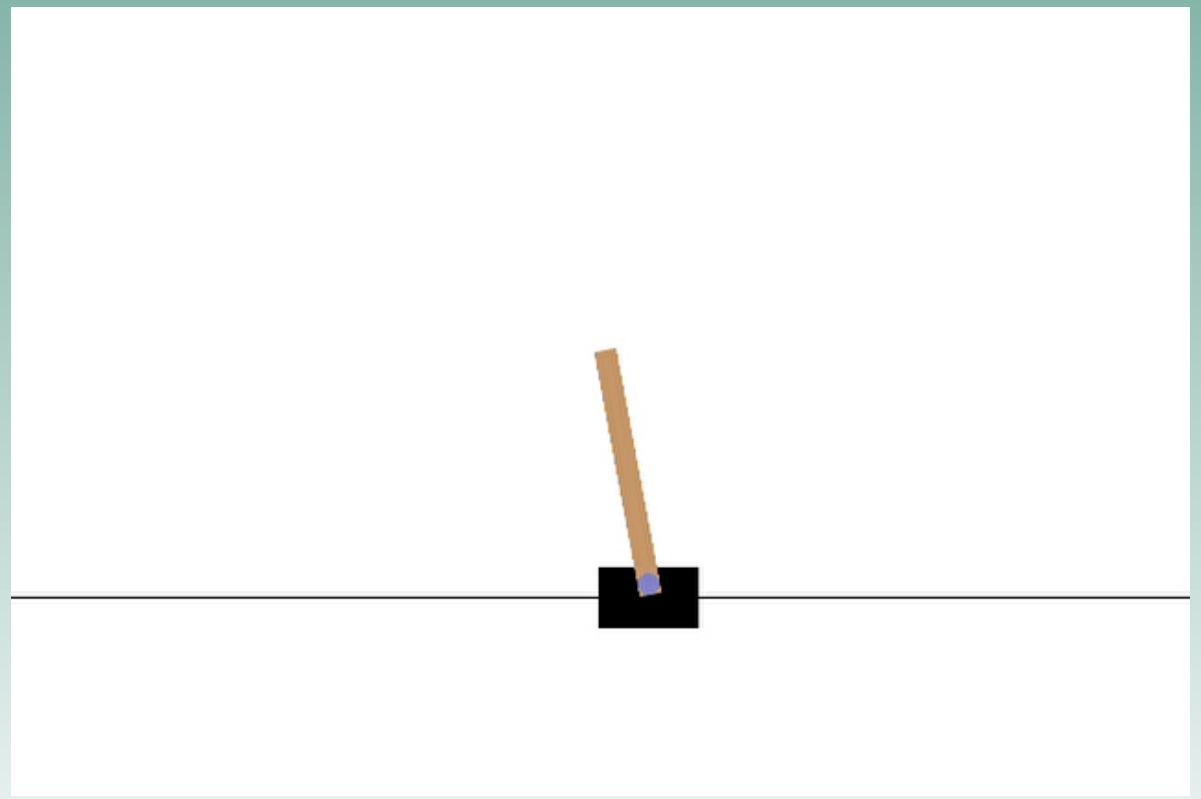
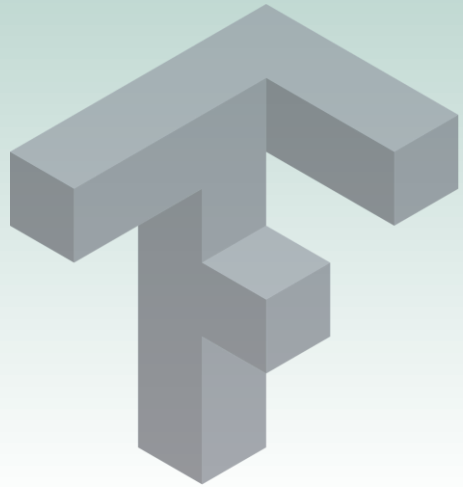


Figure 7: Cartpole demonstration



Result (cont'd)

Performance in first learning sequence vs. third learning sequence

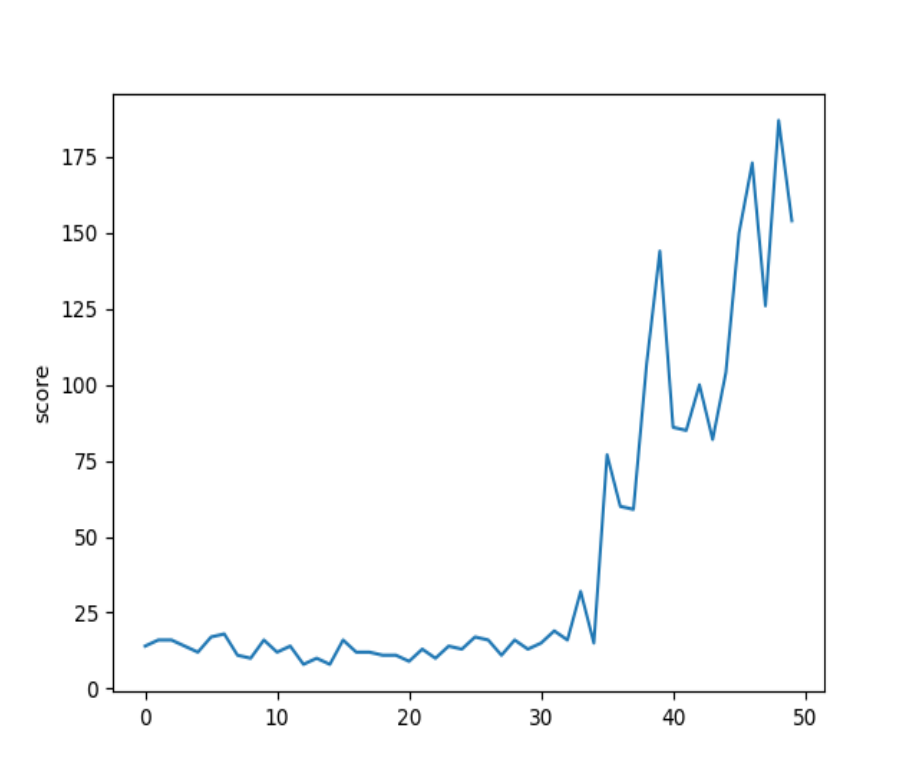


Figure 8: Cartpole First Sequence Performance

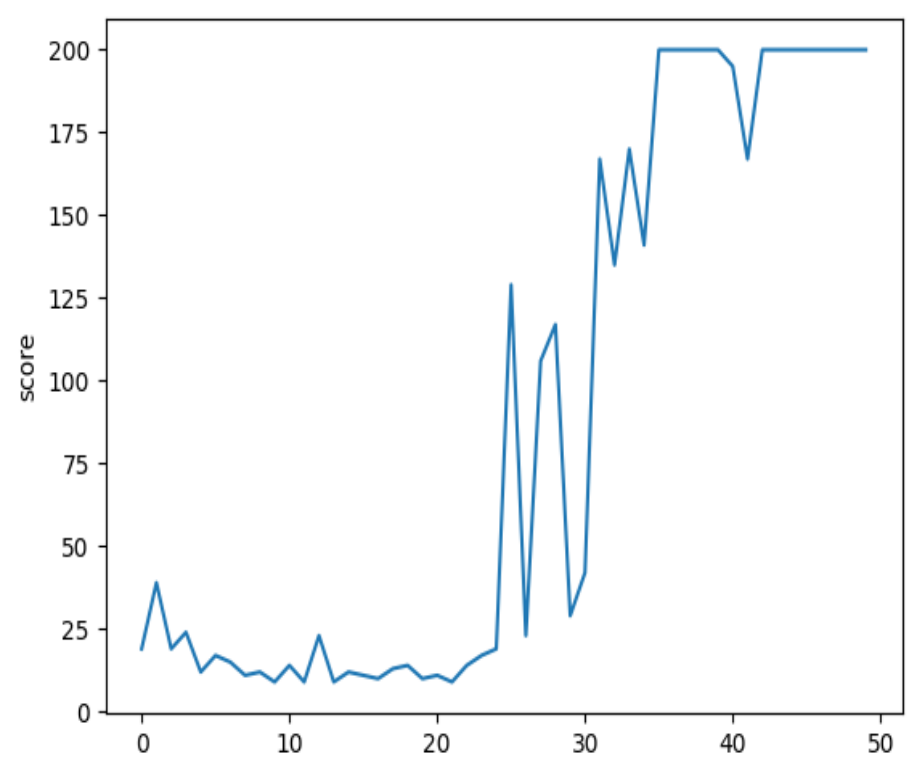


Figure 9: Cartpole Third Sequence Performance

Analysis

DQN Agent Network

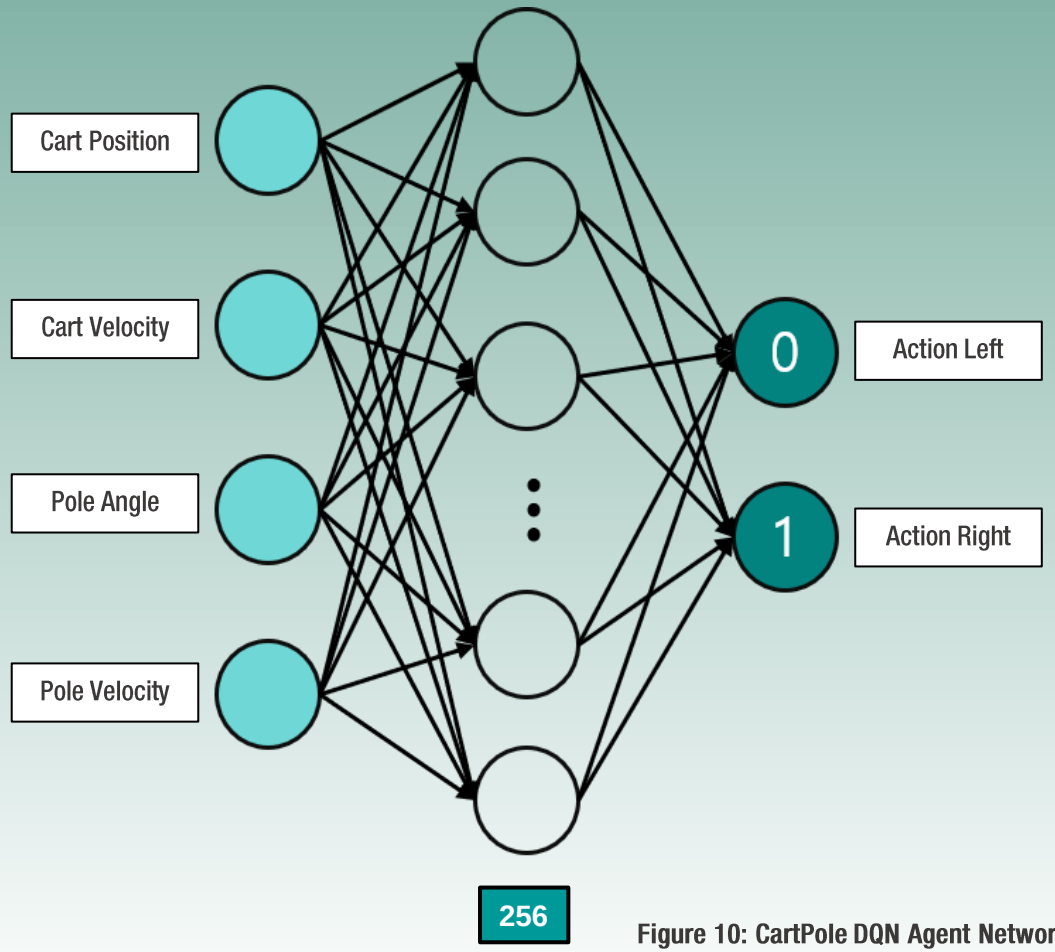
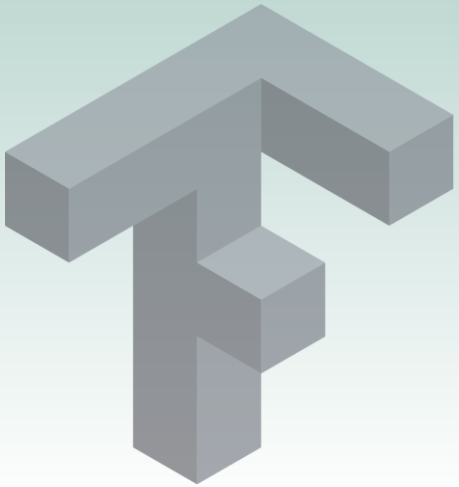


Figure 10: CartPole DQN Agent Network

Figure 8 and Figure 9 are both iterations of learning cycles in CartPole.

Although differing due to probability, there is a clear pattern and trend in performance rate and shows that the CartPole game can be excelled through the use of replay buffers and approximation of Q values.



Atari

The Atari environment is based off the Atari 2600 arcade games.

- High dimensional visual-input (210 x 160 RGB video at 60hz)
- Diverse games and objectives → diverse rewards, actions
- Difficult for human players

Perfect for RL application

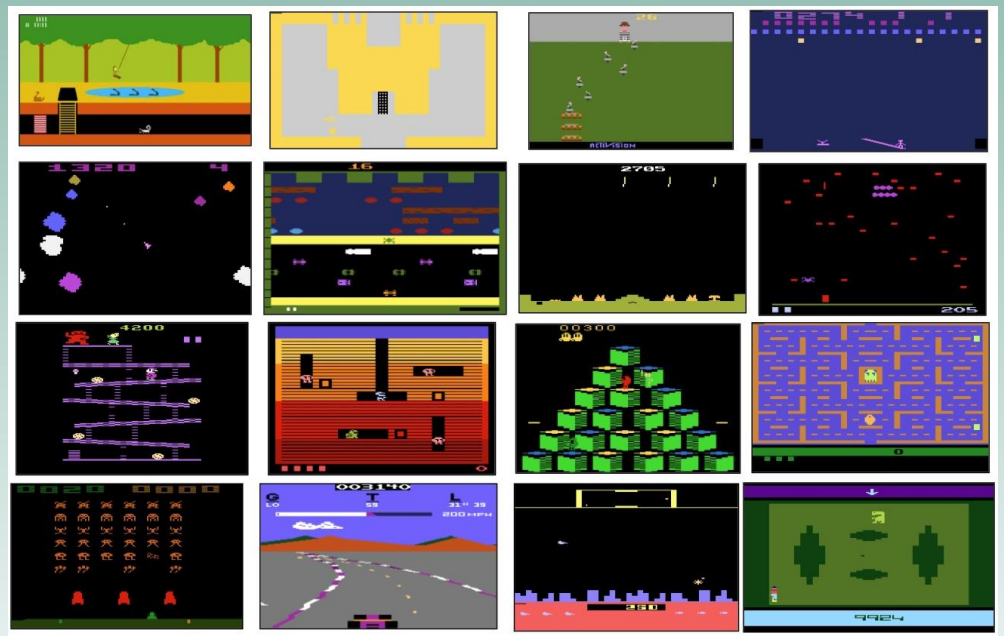


Figure 11: Atari 2600 games

Pong Game

You control the right paddle, you compete against the left paddle controlled by the computer. You each try to keep deflecting the ball away from your goal and into your opponent's goal.

You get score points for getting the ball to pass the opponent's paddle. You lose points if the ball passes your paddle.

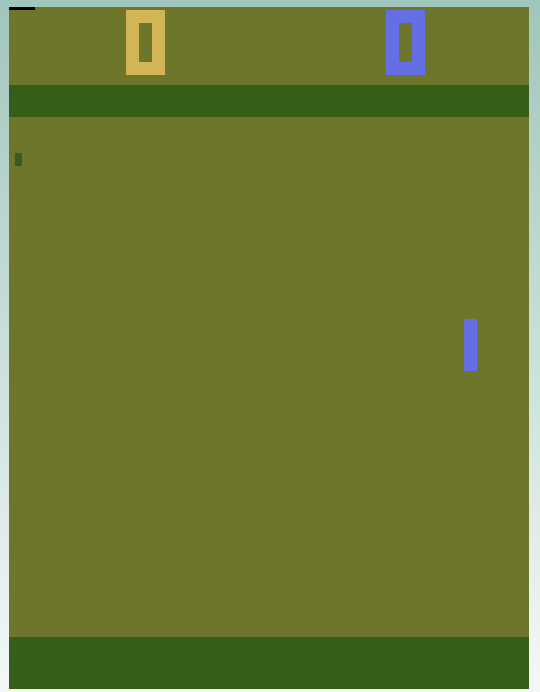
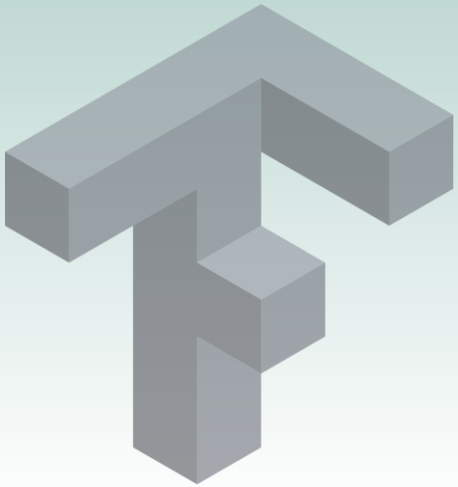


Figure 12: Atari Pong game



Method

Hyperparameters:

- 530 episodes
- exploration rate: (0.05, 0.9)
  - same settings as cartpole
- epsilon_decay = 200
- gamma = 0.8
  - discount rate
- learning_rate = 0.001
- batch_size = 32
- epsilon-greedy = 0.1

*all other rewards, presets, and environmental factors were kept same as the original ‘Pong’ environment in Atari

Result

Performance in first episode vs. 530th episode

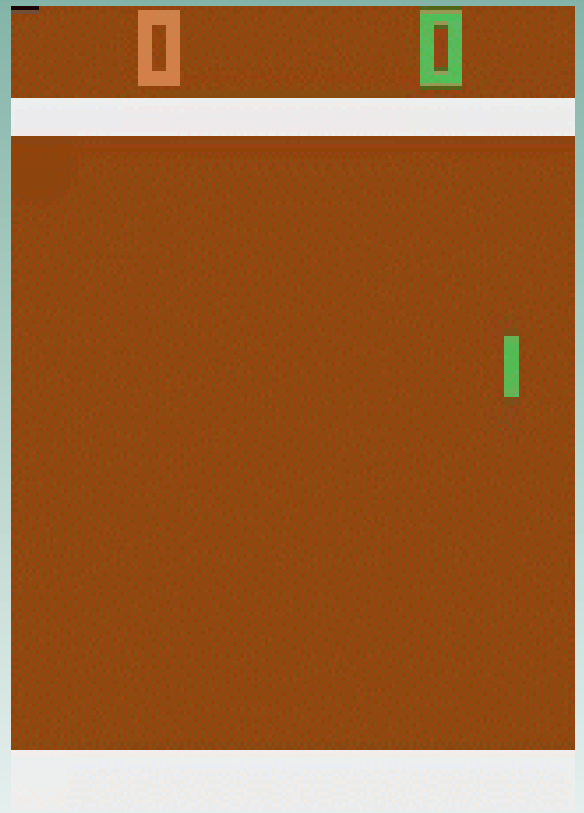


Figure 13: Pong Episode 0

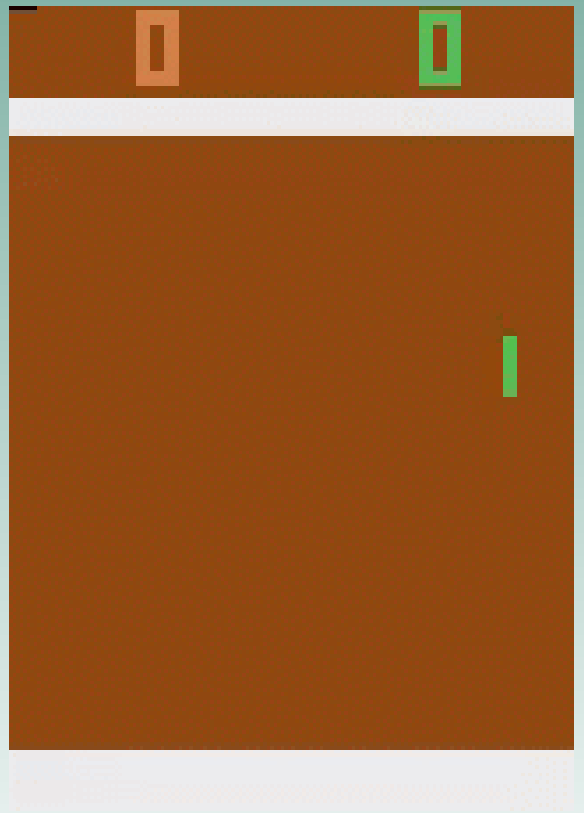


Figure 14: Pong Episode 530

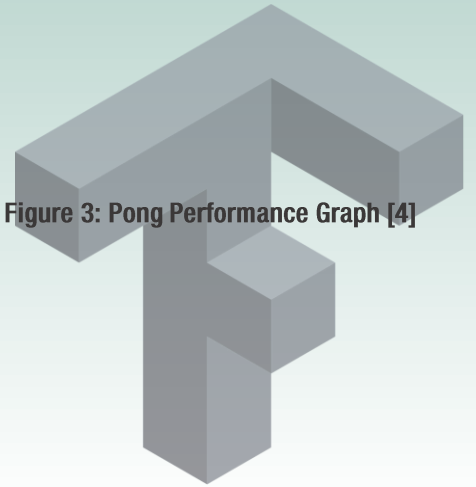


Figure 3: Pong Performance Graph [4]

Result (cont'd)

Parameter: 530 episodes

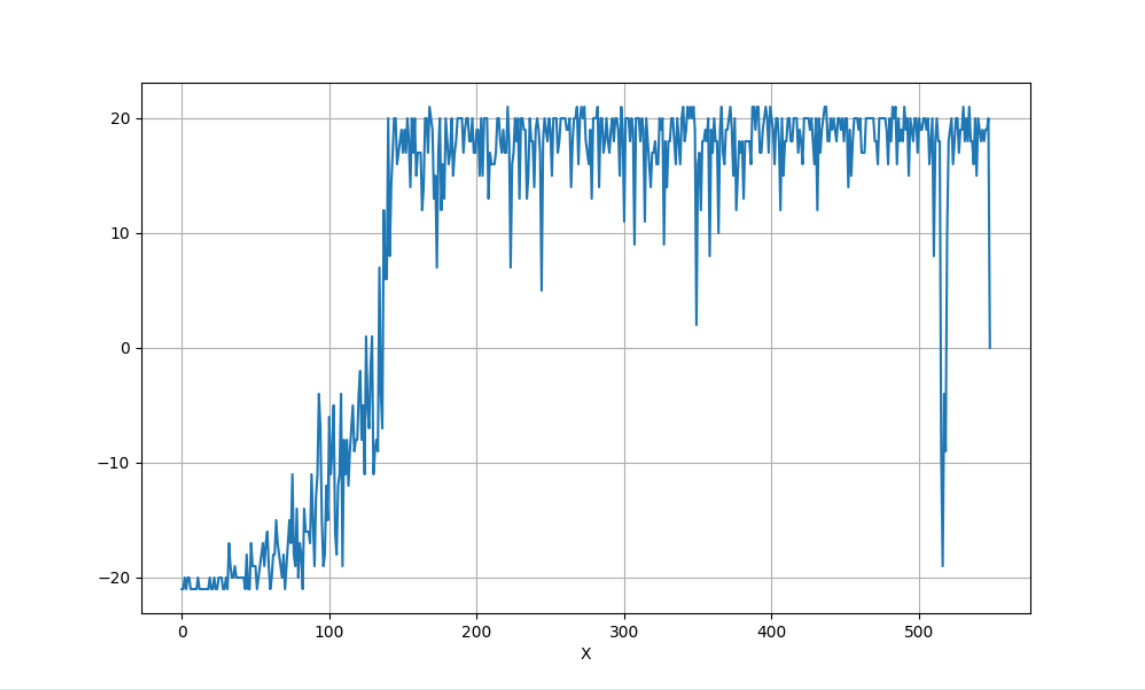


Figure 15: Pong Performance

Figure 3: Pong Performance Graph [4]

Analysis

DQN Agent Network

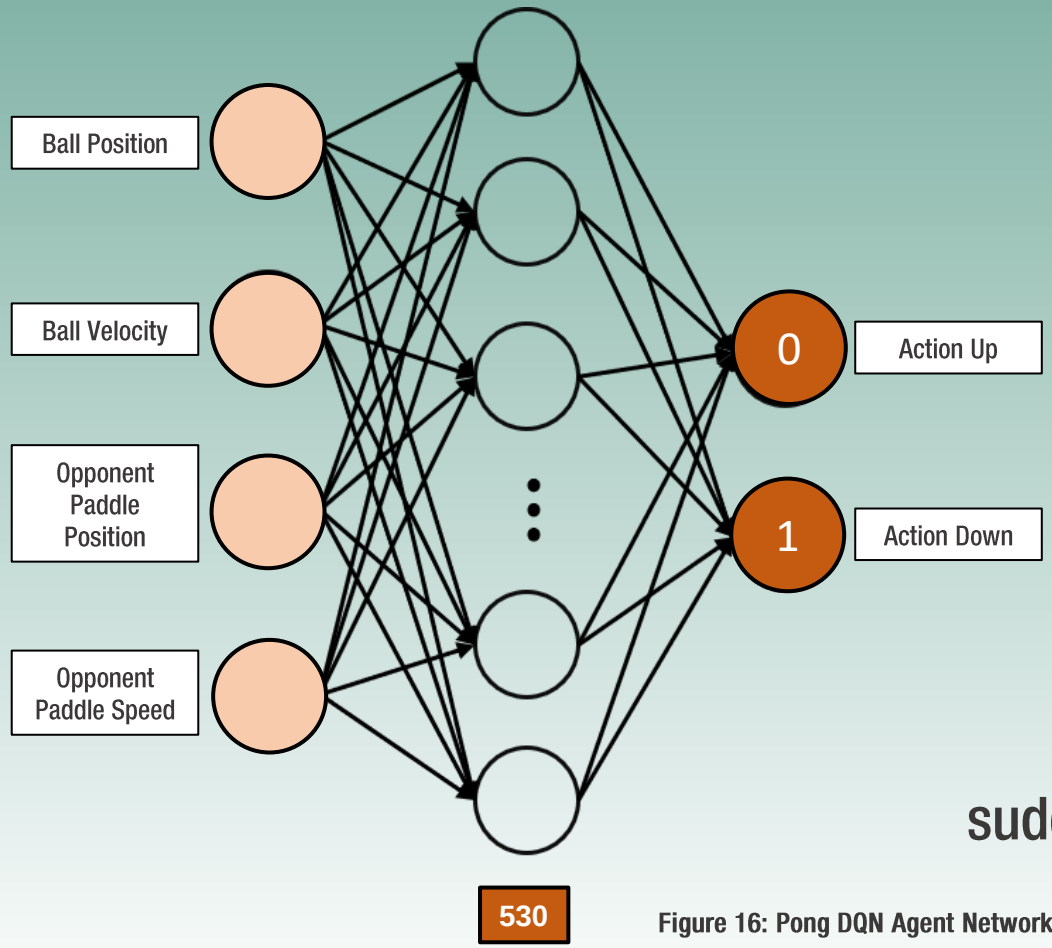


Figure 16: Pong DQN Agent Network

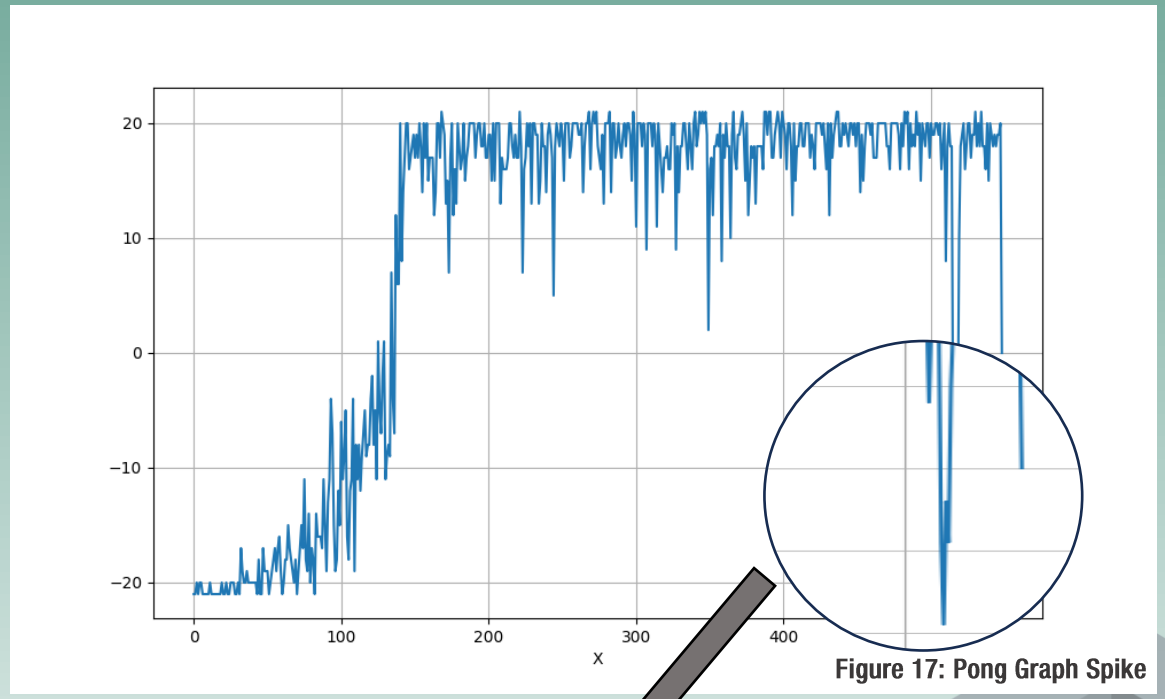


Figure 17: Pong Graph Spike

sudden decay due to exploration rate and epsilon-greedy rate

Analysis (cont'd)

DQN Agent Network

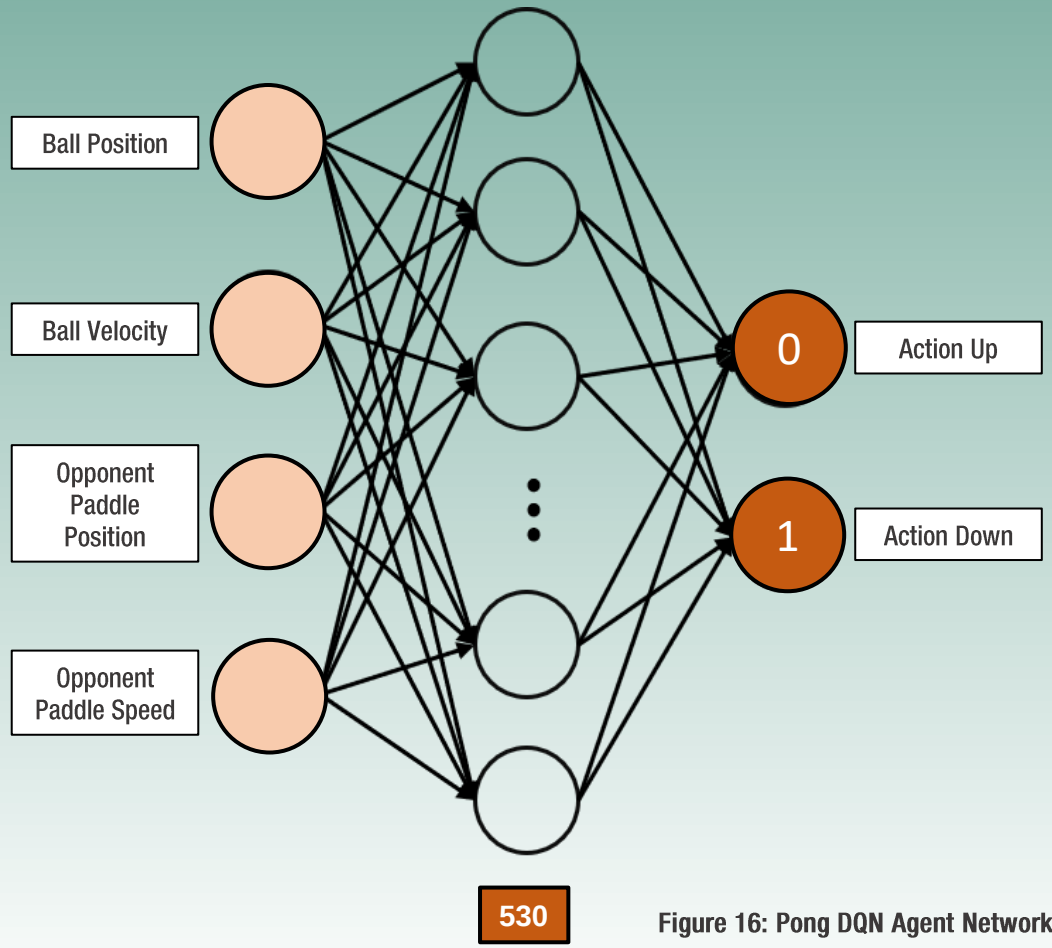
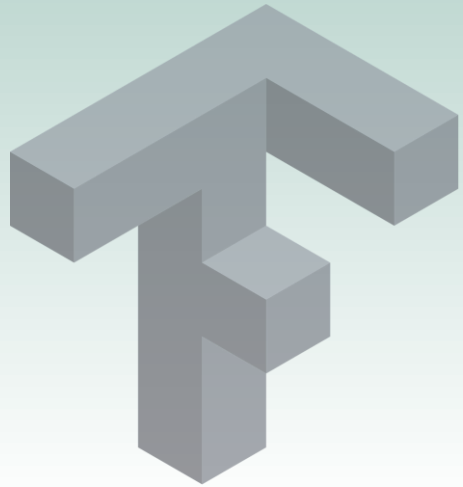


Figure 16: Pong DQN Agent Network

Despite spikes, the graph has an increasing trend and as shown in Figure 13 and 14, there is a drastic change in performance after 530 episodes of learning training.

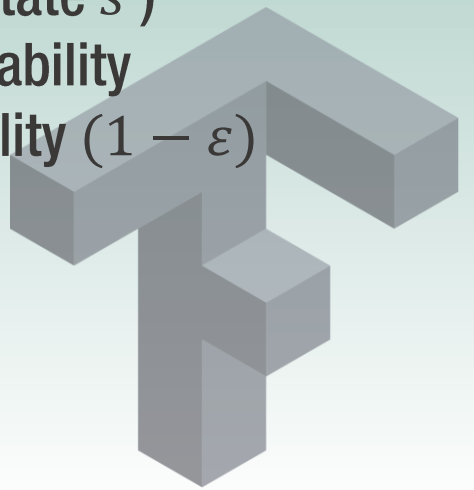


Conclusion

In this survey presentation, I introduced Deep Q-Learning (DQN), a reinforcement learning method that combines Q-learning with deep neural networks to handle high-dimensional state spaces with complexities.

To summarize, DQN consists of

- Q-learning: a model-free algorithm used for finding optimal policy given a finite Markov Decision Process (MDP)
- Deep neural networks: used to calculate Q-function
- Experience relay: stores and sample experiences (state s, action a, reward r, next state s')
- Target network: a fixed network (for a certain number of iterations) that improves stability
- Epsilon-greedy exploration: agent chooses the best-known action with high probability $(1 - \varepsilon)$



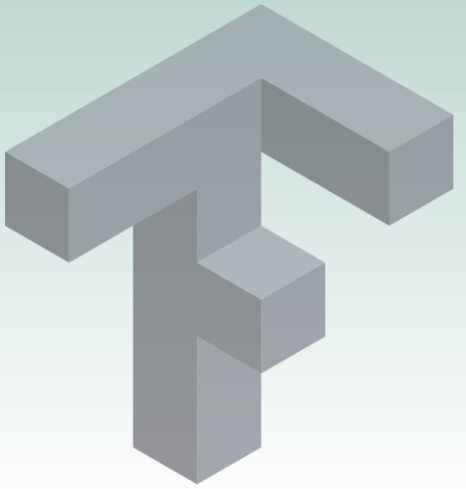
Conclusion (cont'd)

The training process of DQN can be summarized to:

1. Agent interaction with environment to collect experiences (s, a, r, s') and storing them into the replay buffer
2. Agent samples batches of experiences from the replay buffer periodically
3. Agent computes target Q-value using target network and updates online network weights using a loss function

$$L(\theta) = \mathbb{E}[(Q(s, a;\ \theta) - (r + \gamma max_{a'} Q(s', a';\ \theta^{-})))^2]$$

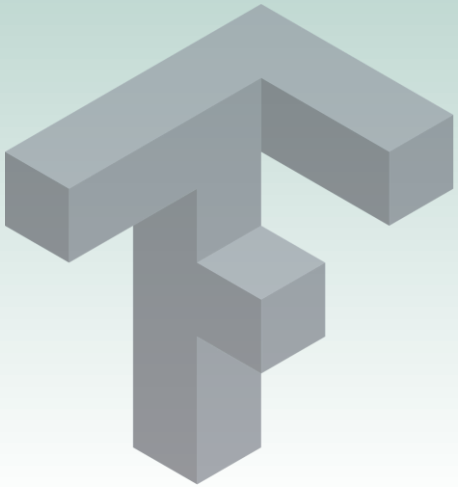
4. Steps 1-3 are repeated to improve the Q-function estimation



Conclusion (cont'd)

The experiments conducted within the survey presentation show how effective DQN is for video games for the following reasons:

- reinforcement learning from pixels
  - DQN can take raw visual input and can operate directly from pixel-level which makes it highly effective for environments such as Atari, etc.
- exploration and exploitation
  - the epsilon-greedy policy used for DQN ensures that the agent explores a variety of different actions and consequential rewards throughout the process, resulting in an optimal policy
- approximation of q-values
  - ensures that DQN performs consequential action based on highest reward value
- nonlinear function approximation
  - ensures that DQN captures any ongoing patterns or relationships

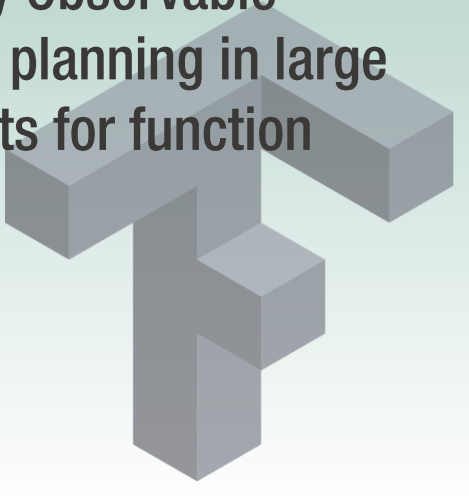


Extension

The Atari game *Pong* provides a visual output and an observable environment in which the agent is already given some information.

Further work can be done in other Atari games such as *Montezuma's Revenge* for Partially Observable Markov Decision Process (POMDP) testing, where the game's initial stages only provides the agent with limited information. In order to 'win' the game, the agent has to explore and remember the layout as it navigates the game.

As POMDP and DQN employ different approaches, it would be wise to employ a Partially Observable Monte Carlo Planning (POMCP) approach, which is an algorithm used for searching and planning in large state spaces with uncertainty and extend the algorithm with neural network components for function approximation.



References

1. Silva, D. (2023, July). *Introduction to Reinforcement Learning. COMPM050*. London, UK; University College London Department of Computer Science.
2. Ladosz, P., Weng, L., Kim, M., & Oh, H. (2022). Exploration in deep reinforcement learning: A survey. *Information Fusion*, *85*, 1–22. https://doi.org/10.1016/j.inffus.2022.03.003
3. Brunskill, E. (2023, July). *CNNs and Deep Q Learning. CS234 Reinforcement Learning.* Stanford, CA; Stanford University Computer Science Department.
4. Mnih, Volodymyr & Kavukcuoglu, Koray & Silver, David & Graves, Alex & Antonoglou, Ioannis & Wierstra, Daan & Riedmiller, Martin. (2013). Playing Atari with Deep Reinforcement Learning.