Reinforced Learning

Sources: http://incompleteideas.net/sutton/book/RLbook2018trimmed.pdf,

Basic Definitions:

- **Reinforcement Learning (RL):** a computational approach to understanding and automating goal-directed learning and decision making. Learning what to do so as to maximize numerical reward signal. Agent is not instructed on which actions to take. Agent discovers which action yields the most reward via trial and error. Actions could yield immediate rewards and/or affect the future subsequent rewards.

- **Agent:** The robot/system/program interacting with the environment/state, choosing actions that manipulate the environment/state to maximize rewards.

- **State:** signal conveyed to the Agent about the current environment.

- **Reward and Return:** $Expected$ return, where the return, denoted R_t, is defined as some specific function of the reward sequence.

  $$R_t = r_{t+1} + r_{t+2} + r_{t+3} + ... = \sum_{k=0}^{\infty} r_{t+k+1}$$

  Used when the agent-environment interactions breaks naturally into subsequences called $Episodes$. Where each episode ends at the terminal state, followed by a reset to the starting state

  Discounted Rewards: The cumulative future discounted award

  $$R_t = r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + ... = \sum_{k=0}^{\infty} \gamma^k r_{t+k+1}$$

  Where γ is a discount rate, $0 < \gamma < 1$. As γ approaches 1, the return objective takes future rewards into account more strongly; the agent becomes more farsighted. As γ approaches 0, the return objective takes immediate rewards into account more strongly; the agent becomes more nearsighted.

- **Episodes:** Tasks that always terminate. [Otherwise return would go on to ∞]
- **Episodic Tasks:** The agent-environment interaction naturally breaks down into a sequence of separate episodes (episodic tasks)

- **Policy:** Strategy of the agent. Mapping from states to probabilities of each possible action. If the agent is following policy π at time t then $\pi(s, a)$ is the probability that $a_t = a$ if $s_t = s$. That is - the probability of taking a certain action in a specific state.

- **Stochastic:** Randomly Determined; having random probability distribution or pattern

- **Value Function:** Functions of states or state-action pairs that estimate how "good" it is to be in a given state or perform a given action. The return is much like a mean; it is literally what you expect to see. The reason we use an expectation is that there is some randomness in what happens after you arrive at a state. (Stochastic policy)

- **State-Value Function:** Tells us the value of a state, s, under policy π. [the expected return starting from state s, at time, t, and following policy π].

  Mathematically:
  $$V_\pi(s) = E\left[\sum_{k=0}^{\infty} \gamma^k r_{t+k+1} | s_t = s\right]$$
  Or simply

  $$V_\pi(s) = E\left[R_t | s_t = s\right]$$

- **Action-Value Function:** Tells us the value of an action, a, in a given state s, under policy π. [the expected return starting from state s, at time, t, taking action a, and following policy π].

  Mathematically:
  $$Q_\pi(s,a) = E\left[\sum_{k=0}^{\infty} \gamma^k r_{t+k+1} | s_t = s, a_t = a\right]$$
  Or simply
  $$Q_\pi(s,a) = E\left[R_t | s_t = s, a_t = a\right]$$

  If you maintain estimates of the action values, then at any time there is at least one action whose estimated value is greatest; call this the greedy action.

- **Bellman Equations:**

  Note $\mathcal{P}_a(s,s') = Pr(s_{t+1} = s' | s_t = s, a_t = a)$ is the transition probability if we start at state s and take action a we end up in state s' with probability $\mathcal{P}_a(s,s')$.

  Then $\mathcal{R}_a(s,s') = E\left[r_{t+1} | s_t = s, s_{t+1} = s', a_t = a\right]$ is the expected (or mean) reward that we receive when starting in state s, taking action a, and moving into state s'.

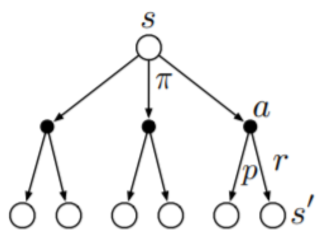
  Finally, the the Bellman Equation for the State-Value Function is:

  $$V_\pi(s) = \sum_a \pi(s,a) \sum_{s'} \mathcal{P}_a(s,s') \left[\mathcal{R}_a(s,s') + \gamma V_\pi(s')\right]$$

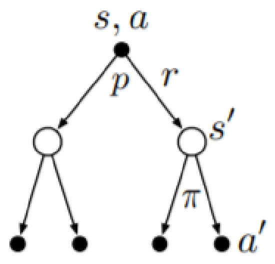
  For the Action-Value Function:

  $$Q_\pi(s,a) = \sum_{s'} \mathcal{P}_a(s,s') \left[\mathcal{R}_a(s,s') + \gamma \sum_{a'} \pi(s',a') Q_\pi(s',a')\right]$$

  The Bellman Equations let us express values of the states as values of other states. This means that if we know the value of s_{t+1}, we can very easily calculate the value of state s_t. Which allows us to calculate the value of each state iteratively.



V_π



Q_π

- **Markov Decision Process (MDP):** classical formalization of sequential decision making, where actions influence not just immediate rewards, but also subsequent situations, or states, and through those future rewards. Thus involve a delayed reward and the need to tradeoff immediate and delayed reward

  5-tuple $(S, A, \mathcal{P}_a, \mathcal{R}_a, \gamma)$ where
  - S is a finite set of states
  - A is a finite set of actions (A_s is the finite set of actions available at state s)
  - $\mathcal{P}_a(s, s') = Pr(s_{t+1} = s' | s_t = s, a_t = a)$ is the transition probability if we start at state s and take action a we end up in state s' with probability $\mathcal{P}_a(s, s')$.
  - $\mathcal{R}_a(s, s')$ is the expected (or mean) reward that we receive when starting in state $\mathcal{R}_a(s, s')$ is the expected (or mean) reward that we receive when starting in state s, taking action a, and moving into state s'.
  - γ is the discount factor that controls the importance of future rewards

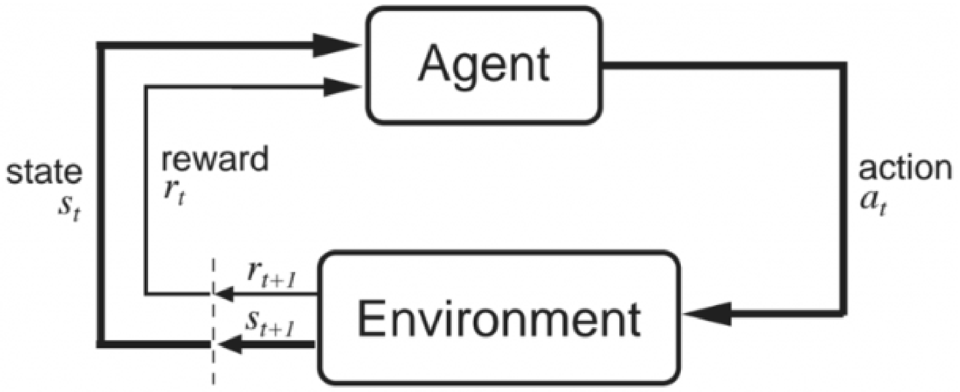


Image from Sutton and Barto – Reinforcement Learning: an Introduction

Markov Property: The state must include information about all aspects of the past agent-environment interaction that make a difference for the future.

- **Optimal Policy and Optimal Value Functions:** A policy π is defined to be better than or equal to a policy π' if its expected return is greater than or equal to that of π' for all states. In other words, $\pi \geq \pi'$ if and only if $V_\pi(s) \geq V_{\pi'}(s)$ for all states. There will always be at least one optimal policy. If there is more than one we denote them all by π_*.

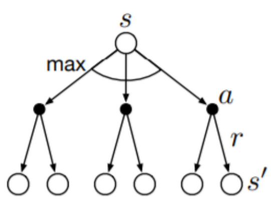
  They all share the same optimal state-value function denoted by V_* and defined as
  $V_*(s) = \max_\pi V_\pi(s)$ for all states.

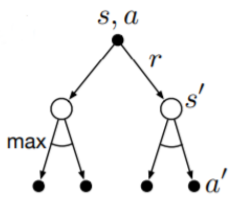
  Also share the same optimal action-value function denoted Q_* and defined as
  $Q_*(s,a) = \max_\pi Q_\pi(s,a)$ for all states.

  The state-action pair (s,a) gives the expected return for taking action a in state s following an optimal policy. Thus, Q_* can be written in terms of V_* as follows:
  $Q_*(s,a) = E\left[r_{t+1} + \gamma V_*(s_{t+1}) | s_t = s, a_t = a\right]$

- **Bellman Optimality Equations:**
  State-value function:
  $$V_*(s) = \max_a \sum_{s'} \mathcal{P}_a(s,s') \left[\mathcal{R}_a(s,s') + V_*(s')\right]$$

  Action-value function:
  $$Q_*(s,a) = \sum_{s'} \mathcal{P}_a(s,s') \left[\mathcal{R}_a(s,s') + \gamma \max_a Q_*(s',a')\right]$$





V_* Q_*

- **Off Policy**: the learned action-value function, Q, directly approximates Q_*, the optimal action-value function, independent of the policy being followed.

- **Q-Learning**: Off-policy Temporal-Difference (TD) control algorithm. The learned action-value function Q, directly approximates Q_*, the optimal action-value function independent from the policy being followed. One-step Q-learning defined by:

$$Q(s_t, a_t) \leftarrow \underbrace{Q(s_t, a_t)}_{\text{old value}} + \underbrace{\alpha}_{\text{learning rate}} \left[r_{t+1} + \gamma \underbrace{\max_a Q(s_{t+1}, a_t)}_{\text{estimate of optimal future value}} - \underbrace{Q(s_t, a_t)}_{\text{old value}} \right]$$

$$Q(s_t, a_t) \leftarrow \underbrace{Q(s_t, a_t)}_{\text{old value}} + \underbrace{\alpha}_{\text{learning rate}} \left[r_{t+1} + \gamma \underbrace{\max_a Q(s_{t+1}, a_t)}_{\text{estimate of optimal future value}} - \underbrace{Q(s_t, a_t)}_{\text{old value}} \right]$$

Algorithm:

```
Loop for each episode:
    Initialize $s$
    Loop for each step of episode:
        Choose an action, $a$, using policy derived from $Q$
        Take action $a$, receive $r$
        Observe new state $s'$
        Update $Q(s,a)$
        Update $s \leftarrow s'$
    Until $s$ is terminal
```

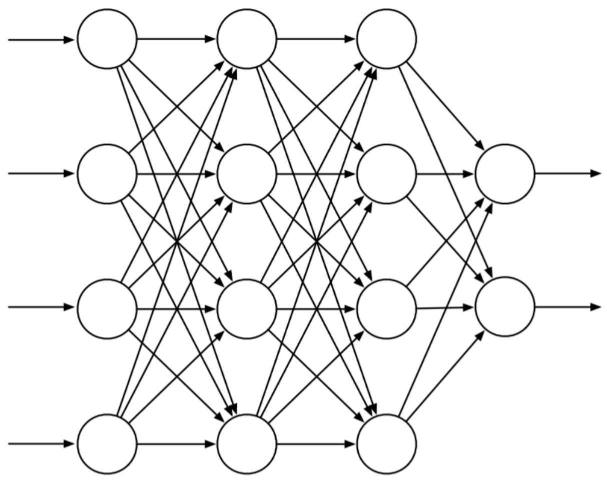
Results: Given sufficient exploration, Q-Learning converges to optimal policy even if you're acting sub-optimally (due to not using the optimal policy)

Q-Learning is generally represented as a neural network and the function is estimated through back propagation.

- **Exploration vs Exploitation:** Trade off between exploiting the known actions and their results vs exploring new actions.

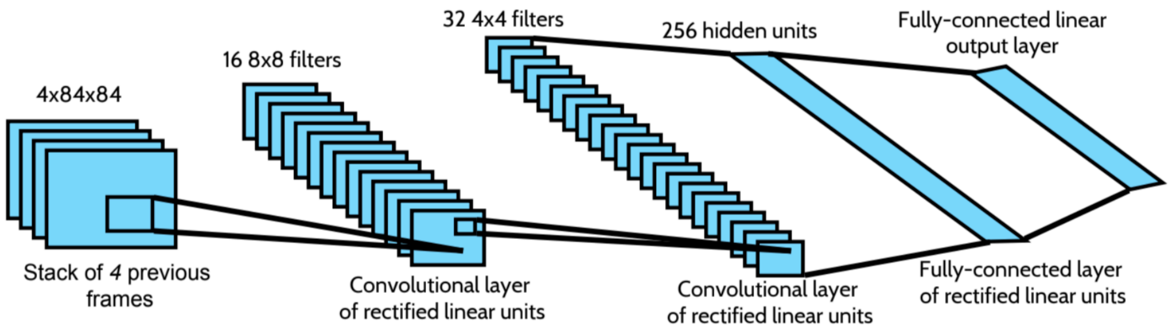
- **Function Approximation:** expect to receive examples of the desired input-output behavior of the function they are trying to approximate. We use these methods for value prediction simply by passing to them the $s \mapsto u$ of each update as a training example. We then interpret the approximate function they produce as an estimated value function. In reinforcement learning, however, it is important that learning be able to occur online, while the agent interacts with its environment or with a model of its environment. To do this requires methods that are able to learn eciently from incrementally acquired data. In addition, reinforcement learning generally requires function approximation methods able to handle non stationary target functions (target functions that change over time).

- **Artificial Neural Networks (ANN):** network of interconnected units that have some of the properties of neurons, the main components of nervous systems. The network in the figure has an output layer consisting of two output units, an input layer with four input units, and two “hidden layers”: layers that are neither input nor output layers. A real-valued weight is associated with each link.



The units are typically semi-linear units, meaning that they compute a weighted sum of their input signals and then apply to the result a nonlinear function, called the activation function, to produce the unit’s output, or activation. The activation of each output unit of a feedforward ANN is a nonlinear function of the activation patterns over the network’s input units. The functions are parameterized by the network’s connection weights.

Convolutional Neural Network (CNN): Artificial Neural Network. Has hidden convolutional layers, that receives and transforms input. Each layer has a filter that detects patterns from the input.



32 4x4 filters
256 hidden units
Fully-connected linear output layer
16 8x8 filters
4x84x84
Stack of *4* previous frames
Convolutional layer of rectified linear units
Convolutional layer of rectified linear units
Fully-connected layer of rectified linear units

Deep Q-Networks (DQN): This type of network is specialized for processing high-dimensional data arranged in spatial arrays, such as images. Because of its special architecture, a deep convolutional network can be trained by backpropagation.

- **Gradient Descent:** Iterative optimization algorithm for finding local minimum of a differentiable function.
  Let $f(x)$ be some multivariable, differentiable, function defined at point a. $f(x)$ decreases fastest if one goes from a in the direction of the negative gradient of f at a $-\nabla f(a)$.
  If $a_{n+1} = a_n - \gamma \nabla f(a_n)$ for $\gamma \in \mathbb{R}^+$ then $f(a_n) \geq f(a_{n+1})$.
  The sequence will either time out or converge at the desired local minimum.
  Algorithm:

```
Initialize $x$ to start the descent
Initialize $\gamma$, the learning rate
Until convergence (or time out):
    Calculate $x_{n+1} \leftarrow x_n - \gamma \nabla f(x_n)$
    Calculate gradient at $f(x_{n+1})$
```

  Loss Function: Computes the error for a single iteration. Goal is to minimize this. Loss function will not improve when hit convergence. (Many different loss functions: sum of the squared residuals, etc;)

  This is a computationally expensive process and does not scale for large data. This is where stochastic gradient descent comes in

  Stochastic Gradient Descent (SGD): Replace the gradient with an estimate thereof. That is, pick a data point, a batch of data points, to reduce the amount of computations.

- **Experience Replay/Replay Memory:** Off-policy learning is inherently of greater variance than on-policy learning. This is not surprising; if you receive data less closely related to a policy, you should expect to learn less about the policy’s values. Experience Replay’s are the previous agent’s experience. At some time-step the replay memory is accessed to perform the weight updates. It overall reduces the variance of the system.

Main Points:

1. Introduction:

- Long-Standing challenge of RL is learning how to control agents directly from *high-dimensional sensory inputs*. (vision, speech, etc)

- All past successful applications relied on hand-crafted features/linear value functions/policy representations.

- DL CHALLENGES:

1. All to date have required hand-labelled training data (**RL algorithms must be able to learn from from scalar reward signals** where the **delay between actions and resulting rewards** can be thousands of time steps long)
2. Most assume data samples to be independent (RL encounters sequences of **highly correlated states** and the **data distribution changes** as the algo learns new behaviors)

- This paper demonstrates a CNN that can overcome these challenges to **learn control policies from raw video data.** Using:

1. **Variant of the Q-Learning Algorithm**
2. With **Stochastic Gradient Descent** to update weights
3. Uses **experience replay mechanism** which randomly samples previous transitions (smooths training distribution and alleviates problems of correlated data and non-stationary distributions).

- **High-Dimensional Visual Input: 210 X 160 RGB video at 60 HZ**

- Goal of paper: Create a single neural network agent that can learn to play as many of the atari games as possible.

- Network not provided with any game-specific information or hand-designed (labelled) visual features.

- Learned just as a human does with:

1. Video Input
2. Reward and Terminal Signals
3. Set of Possible Actions

- Network Architecture and all hyperparameters kept constant.

2. Background:

Notation:

$\varepsilon = environment$ (The Atari Emulator)

$a_t = action$ from the Set of legal actions $\mathcal{A} = \{1, ..., K\}$

$x_t \in \mathbb{R}^d$ image from the emulator (vector of raw pixel values)

r_t reward representing the change in game score

- Agent only observes image of the current screen, impossible to fully understand current situation just from screen x_t So consider the actions and observations $s_t = x_1, a_1, x_2, ..., a_{t-1}, x_t$ and learn strategies that depend on the sequences. All sequences assumed to terminate in a finite number of time-steps. Results in a large but finite **Markov Decision Process (MDP)** in which each sequence is a distinct state.

- By using the complete sequence s_t as the state at time t: Standard reinforcement learning methods for MDP's were applied.

- Goal of the agent: Interact with the emulator by selecting actions to maximize future rewards. Future rewards discounted by a factor γ per time-step.
- The future discounted return at time t is defined as
  $$R_t = \sum_{t'=t}^{T} \gamma^{t'-t} r_{t'} .$$
  Where T is the time-step at which the game terminates.
- The **Optimal action-value function** is defined as $Q_*(s, a)$ as the maximum expected return achievable.

After seeing some sequence s and taking some action a, $Q_*(s,a) = \mathbb{E}\left[R_t | s_t = s, a_t = a, \pi\right]$, Where π is a policy mapping sequences to actions.

- This function obeys the **Bellman equation** identity. That is, if the optimal value $Q_*(s', s')$ of the sequence s' at the next time-step was known for every possible action a', then the optimal strategy to select is a'. Maximizing the expected return $r + \gamma Q_*(s', a')$,
  . $Q_*(s,a) = \mathbb{E}_{s' \sim \epsilon}\left[r + \gamma \mathrm{max}_{a'} Q_*(s', a') | s, a\right]$

- Basic Idea: **Estimate the action-value function** by using the Bellman Equation as an iterative update, $Q_{i+1}(s,a) = \mathbb{E}\left[r + \gamma \mathrm{max}_{a'} Q_i(s', a') | s, a\right]$. Such value iteration converge to the optimal action-value function $Q_i \rightarrow Q_*$ as $i \rightarrow \infty$ In practice - impractical because the action-value function is estimated separately for each sequence.

- Use a function approximator to estimate the action-value function, $Q(s,a;\theta) \approx Q_*(s,a)$. This is typically a linear function approximator, but can be non-linear as well.

- [We] refer to a **neural network approximator** with weights θ as a Q-network. Which can be trained by minimizing a sequence **loss function** $L_i(\theta_i)$ that changes every iteration.

- $L_i(\theta_i) = \mathbb{E}_{s,a \sim \rho(.)}\left[(y_i - Q(s,a;\theta_i))^2\right]$, where $y_i = \mathbb{E}_{s' \sim \epsilon}\left[r + \gamma \mathrm{max}_{a'} Q(s', a', \theta_i) | s, a\right]$ is the target iteration i and $\rho(s,a)$ is a probability distribution over sequences s and actions a referred to as the behaviour distribution.

- Differentiating the loss function with respect to the weights give you the following gradient:

$$\nabla_{\theta_i} L_i(\theta_i) = \mathbb{E}_{s,a \sim \rho(.); s' \sim \epsilon}\left[(r + \gamma_{a'} Q(s', a', \theta_{i-1}) - Q_(s, a; \theta_i))\nabla_{\theta_i} Q(s, a, ; \theta_i)\right]$$

  Use a **Stochastic Gradient Descent** to optimise.

- We arrive at the familiar Q-Learning algorithm if the weights are updated after every time step and the expectations are replaced by single samples from the behavior distribution.

- This algorithm is model-free; It learns about the greedy strategy while following the behavior distribution that ensures sufficient exploration of the state space. In practice the behavior distribution is selected by the greedy strategy with a probability of $1-\epsilon$ and selects a random action with probability ϵ.

4. Deep Reinforcement Learning:

- Our goal is to connect a reinforcement learning algorithm to a **deep neural network** which operates directly on RGB images and efficiently process training data by using stochastic gradient updates.

- We utilize a technique known as **experience replay**, where we store the agent's experiences at each time-step $e_t = (s_t, a_t, r_t, s_{t+1})$ in a data set $\mathcal{D} = e_1, \ldots, e_N$, pooled over many episodes into a **replay memory**. After performing experience replay, the agent selects and executes an action according to an $\epsilon - \text{greedy}$ policy. our Q-function instead works on fixed length representation of histories produced by a function ϕ.
- We apply mini batch updates to samples of experience $e \sim \mathcal{D}$ drawn at random.

  Advantages:
  1. Allows for greater data efficiency
  2. learning directly from consecutive samples is inefficient, due to the strong correlations between the samples; randomizing the samples breaks these correlations and therefore reduces the variance of the updates.
  3. Avoids unwanted feedback loops that may arise if the parameters get stuck in a poor local minimum or diverge catastrophically.

By using experience replay the behavior distribution is averaged over many of its previous states, smoothing out learning and avoiding oscillations or divergence in the parameters.

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**
    Initialize sequence $s_1 = \{x_1\}$ and preprocessed sequence $\phi_1 = \phi(s_1)$
    **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 the emulator
        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 transition $(\phi_j, a_j, r_j, \phi_{j+1})$ from $\mathcal{D}$
        Set $y_j = \begin{cases} r_j & \text{for terminal } \phi_{j+1} \\ r_j + \gamma \max_{a'} Q(\phi_{j+1}, a'; \theta) & \text{for non-terminal } \phi_{j+1} \end{cases}$
```

$$y_j = \begin{cases} r_j & \text{for terminal } \phi_{j+1} \\ r_j + \gamma \max_{a'} Q(\phi_{j+1}, a'; \theta) & \text{for non-terminal } \phi_{j+1} \end{cases}$$

```
        Perform (stochastic) gradient descent step on $(y_j - Q(\phi_j, a_j; \theta))^2$
    **end for**
**end for**
```

4.1 Preprocessing and Model Architecture:

- Working with raw Atari frames is demanding so apply a basic preprocessing step aimed at reducing the input dimensionality.
- The raw frames are preprocessed by converting their RGB representation to gray-scale and down-sampling it to a 110X84 image. Then by performing a final 84X84 cropping region of the image that captures the playing area.

- We use an architecture in which there is a separate output unit for each possible action, and only the state representation is an input to the neural network. The outputs correspond to the predicted Q-values of the individual action for the input state.

- The **input of the neural net** is an $84X84X4$ image produced by ϕ.
  The first hidden layer convoles 16 $8X8$ filters with stride 4 with the input image and applied a rectifier nonlinearity[10,18].
  The second hidden layer convoled 32 $4X4$ filters with stride 2, again followed by a rectifier nonlinearity.
  The final hidden layer is fully-connected and consists of 256 rectifier units.
  The output layer is a fully-connected linear layer with single output for each valid action. (varied between the different games)

<u>5. Experiments:</u>

- Performed experiments on 7 ATARI games
    1. Beam Rider
    2. Breakout
    3. Enduro
    4. Pong
    5. Q*bert
    6. Seaquest
    7. Space Invaders

- Same network architecture, learning algorithm, and hyperparameters.
- Made one change to the reward structure of the games during training only. Since the scale of scores varies greatly from game to game, we fixed all positive rewards to be 1 and all negative rewards to be −1, leaving 0 rewards unchanged.

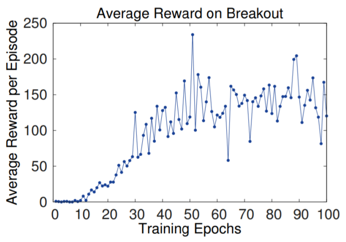
- Trained for a total of 10 million frames and used replay memory of one million most recent frames

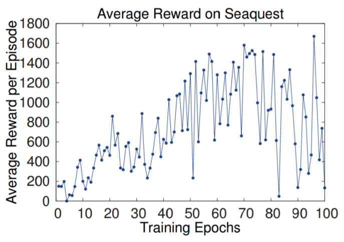
- Used simpel frame-skipping-technique: The agent sees and selects actions on every k^th frame instead of every frame. It's last action is repeated on skipped frames. Allows agent to play k more times without increasing runtime. Used $k = 4$ in all games except Space Invaders as it made the lasers invisible ($k = 3$ for SI).

5.1 Training and Stability:

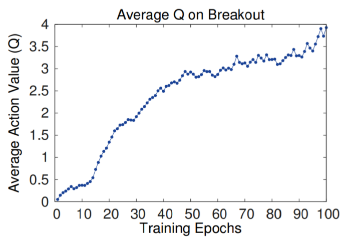
- Periodically computed the total reward the agent collected during training.

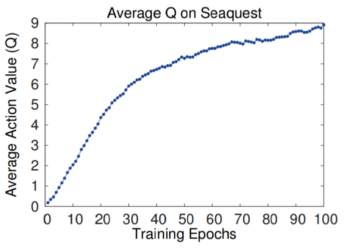
- The average total reward metric tends to be very noisy because small changes to the weights of a policy can lead to large changes in the distribution of states the policy visits .





Average Reward per Episode. Computed running $\epsilon - \text{greedy}$ policy with $\epsilon = 0.05$ for 10000 steps.





Average Maximum Predicted Action-Value of states. One epoch corresponds to 50000 minibatch weight updates. (30 minutes of training time).

5.2 Visualizing the Value Function:

5.3 Main Evaluation:

6 Conclusion:

Summary:

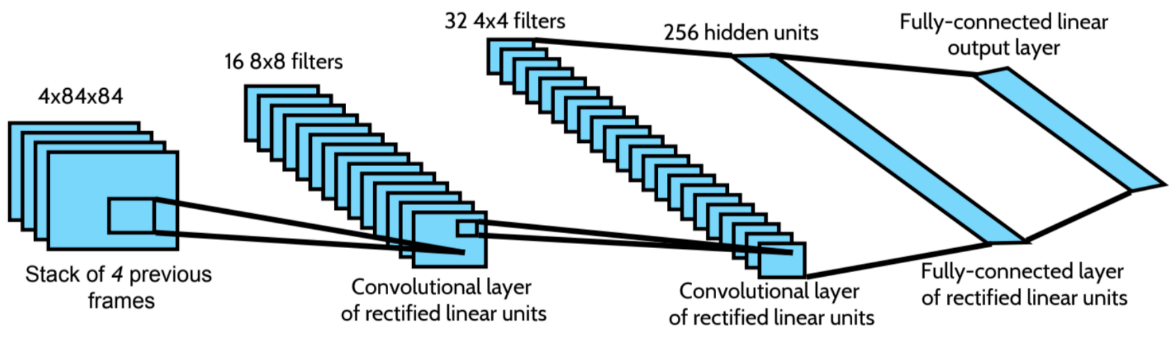
Only raw pixels, 210 X 160 RGB video at 60 HZ, were used as input. [Fed into the DQN]

Specifically 4 consecutive frames were preprocessed into grayscale, resized, and cropped into 84X84 squares.

4 frames were sent in for a couple reasons

1. Reduce unnecessary complexity. If every frame were treated as a specific state you would have 4X the input.
2. The state, within 4 frames, doesn't relatively change. (60 frames a second, 1 frame every 0.06 milliseconds). You would essentially be feeding in the same 4 frames separately.

These 4 frames are sent into the DQN as a single state.


4x84x84
Stack of 4 previous frames
16 8x8 filters
Convolutional layer of rectified linear units
32 4x4 filters
Convolutional layer of rectified linear units
256 hidden units
Fully-connected layer of rectified linear units
Fully-connected linear output layer

The First Hidden Layer: Stride = 4, Rectifier Nonlinearity[10,18].

The Second Hidden Layer: Stride = 2, Rectifier Nonlinearity

The Third Fully Connected Layer: 256 rectifier units.

The Fully Connected Output Layer: Gives out a single output for each valid action.

This DQN Approximates the Optimal Action-Value Q-Function. That is, spits out the Q-Function which tells the agent the best action to choose given the state sent in. The agent will either choose a random action or this optimal action, which advances the state and gives the agent the reward. It also stores this transition into memory for later use.

It gets better overtime as the same states are “experienced” and the weights change in between the layers. The probability of choosing a random action allows the agent to freely explore while gradually getting better by also exploiting its current knowledge.

misc:

Outperformed Humans:
Breakout, Enduro, Pong

Close to Human Performance:
Beam Rider

Far From Human Performance:
Q*bert, Seaquest, Space Invaders
From Paper: these games “are more challenging because they require the network to find a strategy that extends over long time scales.”

$\Pr\{X = x\}$	probability that a random variable X takes on the value x
$X \sim p$	random variable X selected from distribution $p(x) \doteq \Pr\{X = x\}$
$\mathbb{E}[X]$	expectation of a random variable X, i.e., $\mathbb{E}[X] \doteq \sum_x p(x)x$

In these experiments, we used the RMSProp algorithm with mini batches of size 32. The behavior policy during training was -greedy with annealed linearly from 1 to 0.1 over the first million frames, and fixed at 0.1 thereafter. We trained for a total of 10 million frames and used a replay memory of one million most recent frames.