Deep Reinforcement Learning

Sanket Lokegaonkar

Advanced Computer Vision (ECE 6554)

Outline

- The Why?
- Gliding Over All : An Introduction
  - Classical RL
  - DQN-Era
- Playing Atari with Deep Reinforcement Learning [2013]
  - 7 Atari Games
- Human-level control through deep reinforcement learning. [2015]
  - 49 Atari Games
- Brave New World

The Why? : Task

- Learning to behave optimally in a changing world
- Characteristics of the Task:
  - No Supervisor (Only Rewards)
  - Delayed Feedback
  - Non I.I.D data
  - Previous action affects the next state
- RL:
  - Learning by Interaction and your choices
  - Time-travelling back to the moment: Only to Live Again
- Reminds you of something:
  - Life - (Time Travelling Part)

The Why?: Hardness

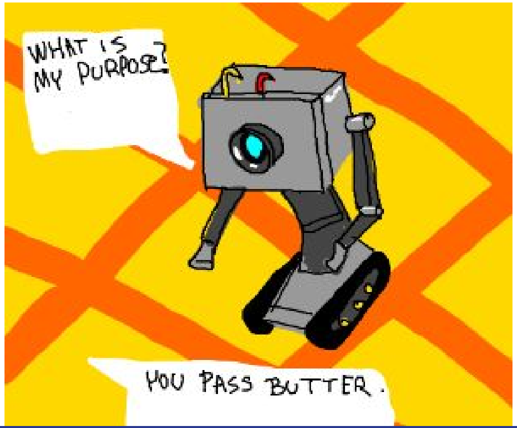
Imagine playing a new game whose rules you don’t know; after a hundred hundred or so moves, your opponent announces, “You lose”.

-Russell and Norvig

Introduction to Artificial Intelligence

The Why?: Why do RL?

- So that we could move away from rule-based systems!!
- So that we could train quadcopters to fly and move !
- So that you could sip coffee all day and your bot will do the trading for you!
- Or Create intelligent robots for dumb tasks

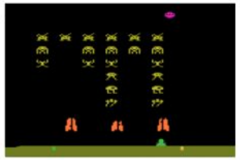


Gliding Over All: An Introduction

ML	Supervised Learning	Target label for each training example
	Reinforcement Learning	Sparse and time-delayed labels
	Unsupervised Learning	No label at all

Pong

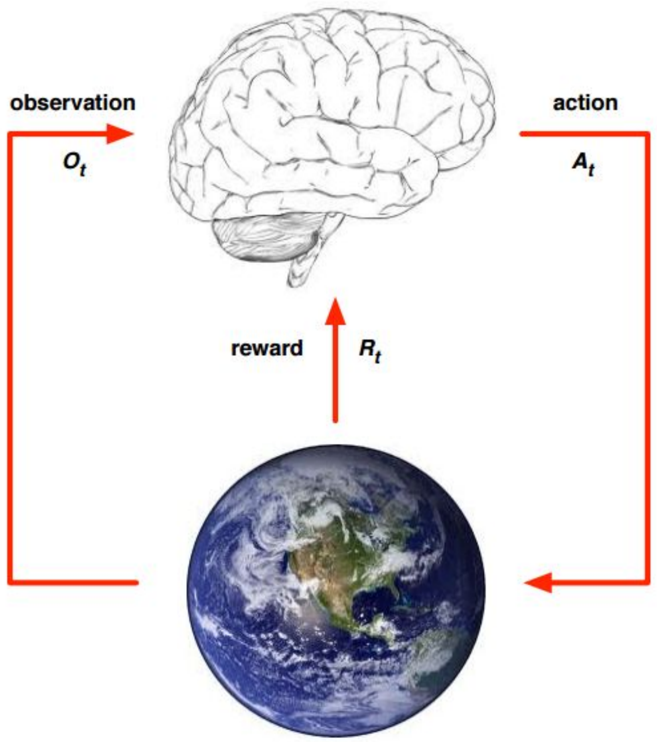
Breakout


Space Invaders

Seaquest

Beam Rider

Credits: David Silver

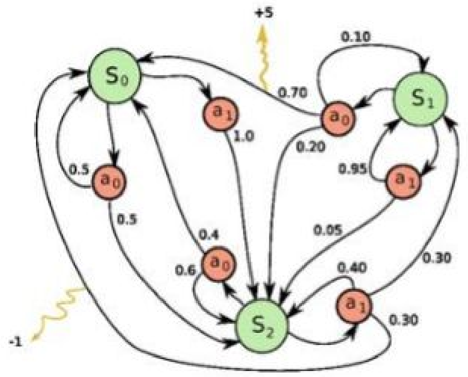


Environment and Agent

- At each step t, the agent
  - Executes action A_t
  - Receives observation O_t
  - Receives scalar reward R_t
- The environment,
  - Receives action A_t
  - Emits observation O_{t+1}
  - Emits scalar reward R_{t+1}
- t increments in every step

Credits: David Silver

Markov Decision Process



$$s_0, a_0, r_1, s_1, a_1, r_2, \ldots, s_{n-1}, a_{n-1}, r_n, s_n$$

state

action

reward

Terminal state

Credits: Andrej Karpathy’s blog

Markov Decision Process (MDP)

- Set of states S
- Set of actions A
- State transition probabilities p(s’ | s, a). This is the probability distribution over the state space given we take action a in state s
- Discount factor γ in [0, 1]
- Reward function R: S x A -> set of real numbers
- For simplicity, assume discrete rewards
- Finite MDP if both S and A are finite

State Transition Probabilities

- Suppose the reward function is discrete and maps from S x A to W
- The state transition probability or probability of transitioning to state s’ given current state s and action a in that state is given by:

$$p(s'|s,a) = \sum_{r \in W} p(s', r|s,a)$$

Expected Rewards

- The expected reward for a given state-action pair is given by:

$$r(s,a) = \mathbb{E}[R_{t+1} | S_t = s, A_t = a] = \sum_{r \in W} r \sum_{s' \in S} p(s', r | s, a)$$

Policy and Value Function

- Policy
  - Behavior Function
  - Mapping from state to action
  - Deterministic Policy : a=\Pi (s)
  - Stochastic Policy: $\pi(a \mid s) = \Pr [A_t = a \mid S_t = s]$
- Value Function
  - Prediction of total future Reward under a policy
  - Captures goodness/badness of states
  - Can be used to greedily select among the actions
  - $v_\pi(s) = E_\pi[R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} \dots \mid S_t = s]$

Credits: ICML DRL Tutorial

Action-Value Function (Q -function)

- Expected Return starting from state s, taking action A and then following policy with γ as the discounting factor.

$$Q^{\pi}(s, a) = \mathbb{E}\left[r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + \ldots \mid s, a\right]$$

- Goodness of state given an action a

Optimal Action-Value Function

- Action -Value Function Equivalent Representation (Bellman Equation):
  - $Q^{\pi}(s,a) = E[R_s + \gamma Q^{\pi}(s',a')]$
- Optimal Value Function:
  - $Q^{*}(s,a) = \max_{\pi} Q^{\pi}(s,a)$
- If we know the optimal action-value function, the optimal policy would be:
  - $\pi^{*}(s) = \text{argmax}_{a} Q^{*}(s,a)$
- Optimal value maximizes over all decisions:
  - $Q^{*}(s,a) = E[R_s + \gamma \max_{a'} Q^{\pi}(s',a')]$

Credits: ICML DRL Tutorial

Bellman Equation (1)

- The equation expresses the relationship between the value of a state s and the values of its successor states
- The value of the next state must equal the discounted value of the expected next state, plus the reward expected along the way

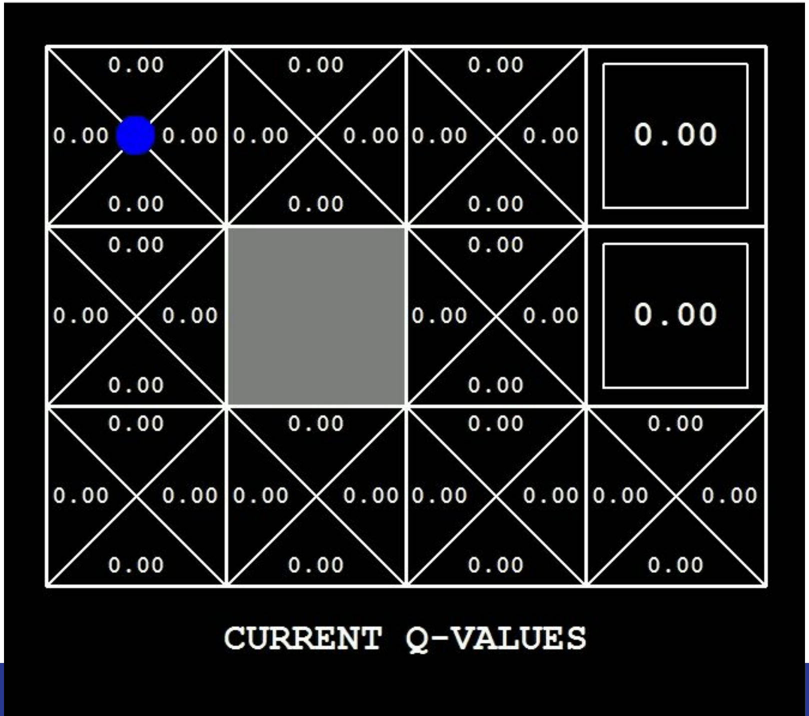
Bellman Equation (2)

- The value of state s is the expected value of the sum of time-discounted rewards (starting at current state) given current state s
- This is expected value of r plus the sum of time-discounted rewards (starting at successor state) over all successor states s' and all next rewards r and over all possible actions a in current state s

$$\forall s \in S : v_\pi(s) = \sum_a \pi(a|s) \sum_{s',r} p(s',r|s,a)[r + \gamma v_\pi(s')]$$

DEMO

http://cs.stanford.edu/people/karpathy/reinforcejs/index.html



Credits: [3]

Approaches in RL

- Value -Based RL (Value Iteration) (Model Free)
  - Estimate the value of optimal value function Q* (s,a)
- Policy -Based RL (Policy Iteration) (Model Free)
  - Search for the optimal policy π^*(s)
- Model-Based RL
  - Learn a model of the environment
  - Plan using the model
- “Deep” networks are used to approximate the functions

Model-free versus Model-based

- A model of the environment allows inferences to be made about how the environment will behave
- Example: Given a state and an action to be taken while in that state, the model could predict the next state and the next reward
- Models are used for planning, which means deciding on a course of action by considering possible future situations before they are experienced
- Model-based methods use models and planning. Think of this as modelling the dynamics $p(s' \mid s, a)$
- Model-free methods learn exclusively from trial-and-error (i.e. no modelling of the environment)
- We focus on model-free methods today:

On-policy versus Off-policy

- An on-policy agent learns only about the policy that it is executing
- An off-policy agent learns about a policy or policies different from the one that it is executing

What is TD learning?

- Temporal-Difference learning = TD learning
- The prediction problem is that of estimating the value function for a policy π
- The control problem is the problem of finding an optimal policy π_*
- Given some experience following a policy π, update estimate v of v_π for non-terminal states occurring in that experience
- Given current step t, TD methods wait until the next time step to update $V(S_t)$
- Learn from partial returns

Epsilon-greedy Policy

- At each time step, the agent selects an action
- The agent follows the greedy strategy with probability 1 – epsilon
- The agent selects a random action with probability epsilon
- With Q-learning, the greedy strategy is the action a that maximises Q given S_{t+1}

Q-learning – Off-policy TD Control

- Similar to SARSA but off-policy updates
- The learned action-value function Q directly approximates the optimal action-value function q_* independent of the policy being followed
- In update rule, choose action a that maximises Q given S_{t+1} and use the resulting Q-value (i.e. estimated value given by optimal action-value function) plus the observed reward as the target
- This method is off-policy because we do not have a fixed policy that maps from states to actions. This is why A_{t+1} is not used in the update rule

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha[R_{t+1} + \gamma \max_a(Q(S_{t+1}, a)) - Q(S_t, A_t)]$$

Deep Q-Networks (DQN)

- Introduced deep reinforcement learning
- It is common to use a function approximator $Q(s, a; \theta)$ to approximate the action-value function in Q-learning
- Deep Q-Networks is Q-learning with a deep neural network function approximator called the Q-network
- Discrete and finite set of actions A
- Example: Breakout has 3 actions – move left, move right, no movement
- Uses epsilon-greedy policy to select actions

Q-Networks

- Core idea: We want the neural network to learn a non-linear hierarchy of features or feature representation that gives accurate Q-value estimates
- The neural network has a separate output unit for each possible action, which gives the Q-value estimate for that action given the input state
- The neural network is trained using mini-batch stochastic gradient updates and experience replay

State representation

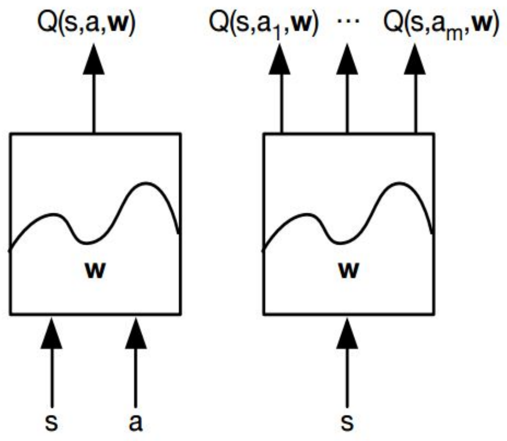
- It is difficult to give the neural network a sequence of arbitrary length as input
- Use fixed length representation of sequence/history produced by a function $\phi(s_t)$
- Example: The last 4 image frames in the sequence of Breakout gameplay

Q-Network Training

- Sample random mini-batch of experience tuples uniformly at random from D
- Similar to Q-learning update rule but:
  - Use mini-batch stochastic gradient updates
  - The gradient of the loss function for a given iteration with respect to the parameter θ_i is the difference between the target value and the actual value is multiplied by the gradient of the Q function approximator Q(s, a; θ) with respect to that specific parameter
- Use the gradient of the loss function to update the Q function approximator

Value-based RL

- Q-function is represented as Q-network with weights :
- $Q(s,a,w) = Q^*(s,a)$



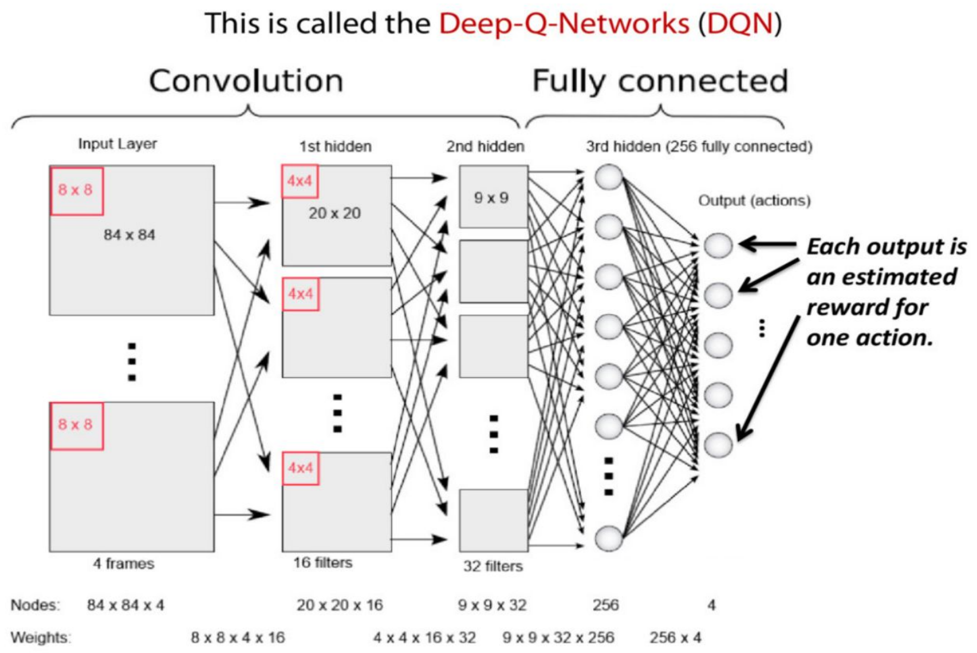
Playing Atari with Deep Reinforcement Learning

Problem Statement:

- Input:
  - Raw Atari Frames 210 x160 pixel with 128-color pallete
- Output:
  - Possible Action to be taken
- Goal /Objective:
  - Optimal Policy with Maximal Reward

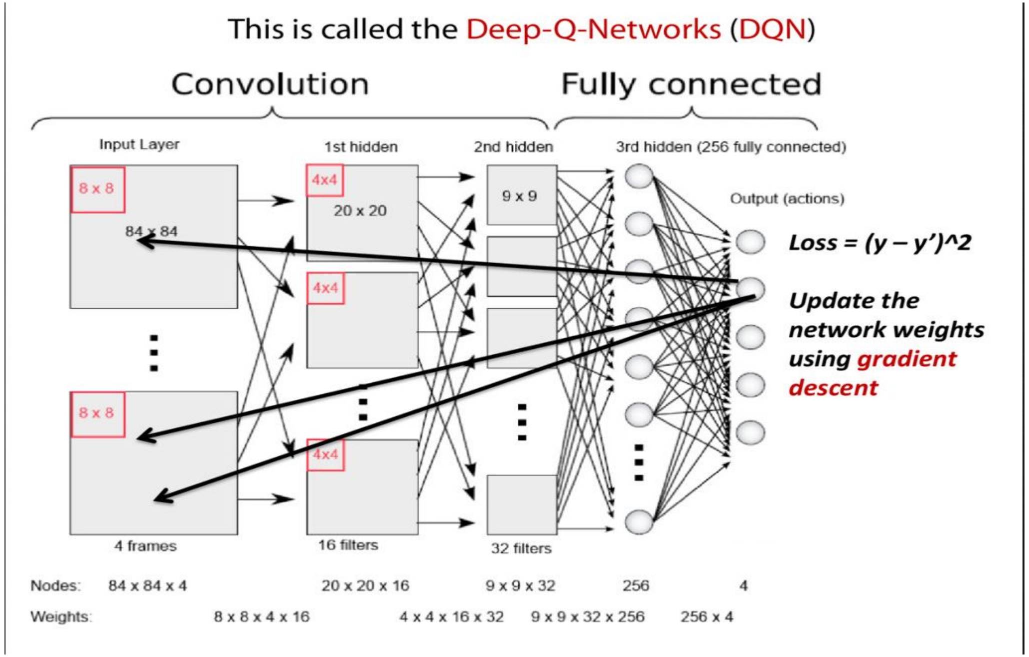
Playing Atari with Deep Reinforcement Learning

Architecture:

- 

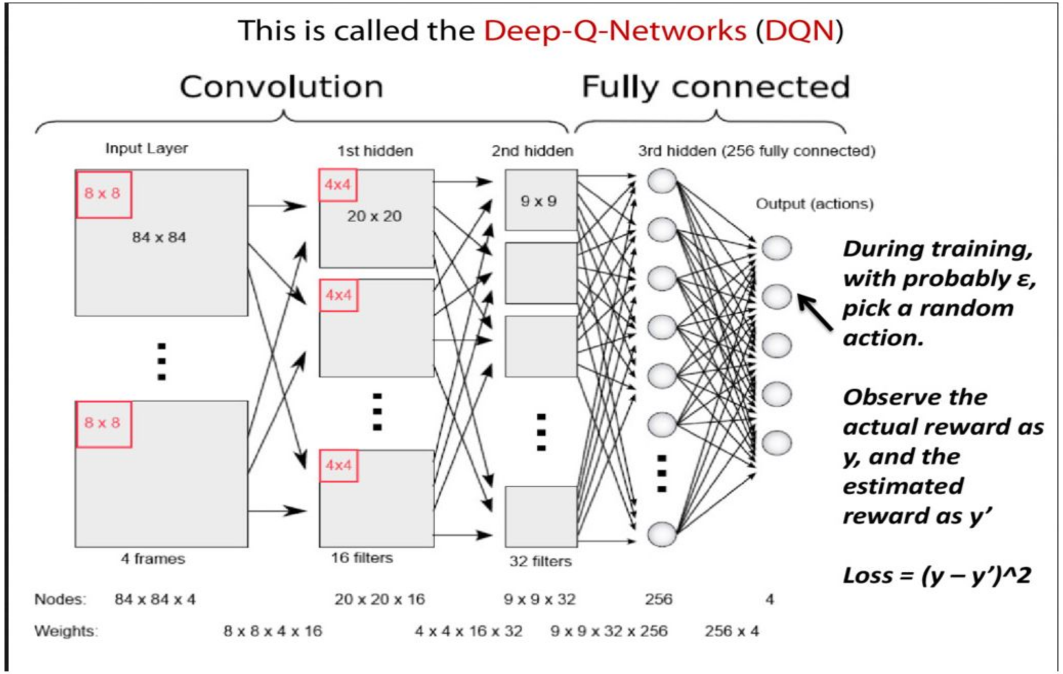
Playing Atari with Deep Reinforcement Learning

Architecture:

- 

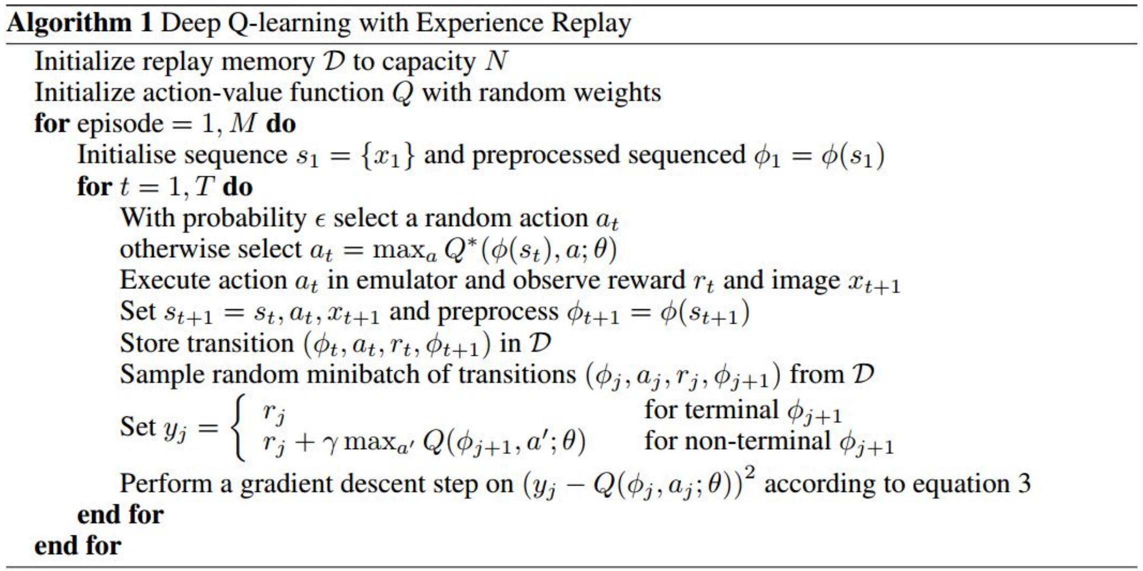
Playing Atari with Deep Reinforcement Learning

Architecture:

- 

Playing Atari with Deep Reinforcement Learning

Algorithm:



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 $\epsilon$ 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
```

Playing Atari with Deep Reinforcement Learning

MSE Loss with DQN:

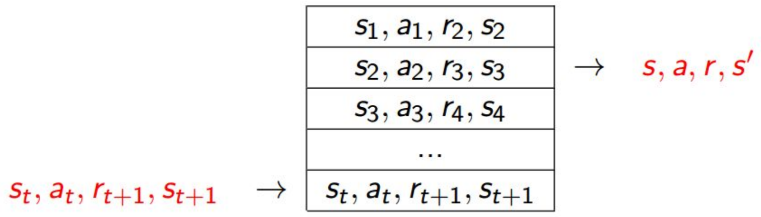
- $$I = \left(r + \gamma \max_{a} Q(s', a', \mathbf{w}) - Q(s, a, \mathbf{w})\right)^2$$

Issues with Convergence of DQN:

- Non I.I.D data
- Oscillating policies with slight variations of Q-values
- Gradient descent can be large and unstable

Solutions to the Issues: Non-IID data

- Experience Replay (for handling non-iid data)
  - Build a memory storing N pairs of agent's experience i.e (s_t,a_t,r_{t+1},s_{t+1})
  - Sample random minibatch experience from the memory
  - Breaks correlation ~ brings back to iid domain


s_1, a_1, r_2, s_2
s_2, a_2, r_3, s_3
s_3, a_3, r_4, s_4
$\dots$
$s_t, a_t, r_{t+1}, s_{t+1}$
$\rightarrow s, a, r, s'$
$s_t, a_t, r_{t+1}, s_{t+1} \rightarrow$

Solution to the Issues: How to prevent oscillations

- Fix parameters used in Q-learning target
- Compute Q-learning targets wrt old fixed parameters θ^-
- Loss Minimization Equation between Q-network and Q-learning targets
  - $L(\theta) = E_{s,a,r,s' \sim D} [r + \gamma \max_{a'} Q(s',a',\theta)]$
- Periodically update fixed parameters θ^-

Solution to the Issues: Unstable Gradient

- Clipping Rewards between [-1,1]
- Limits the scale of the derivatives
- Easier to use same learning rate over different Atari Games
- Disadvantage:
    - Invariant to different magnitudes of rewards

Evaluations

Atari Games Tried:

- Beam Rider
- Breakout
- Pong
- Q*bert
- Seaquest
- Space Invaders

Evaluations

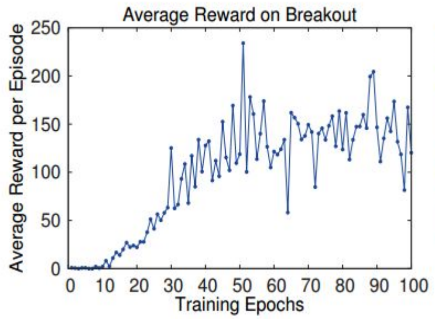
Metrics:

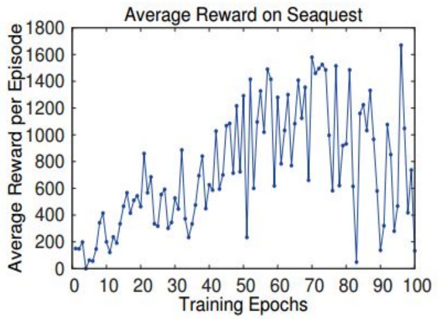
- Average Total Reward Metric
- Average action-value
    - Collect a fixed set of states by running a random policy before training starts
    - Average of the maximum predicted Q for these states

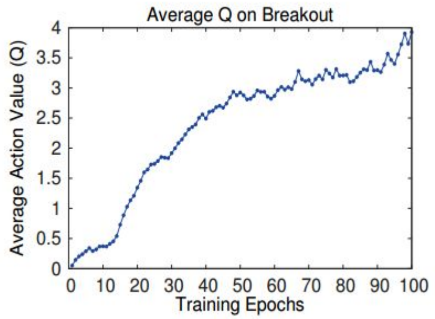
Optimization:

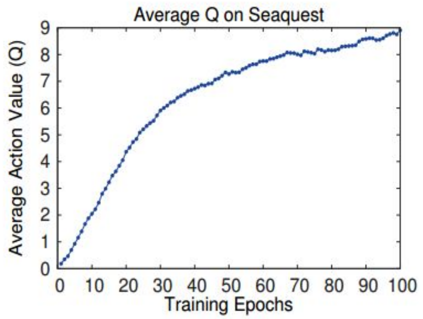
- RMSProp with minibatches of size 32
- Behavior policy ϵ greedy (Annealed from 1 to 0.1 and fixed to 0.1 later)

Evaluations



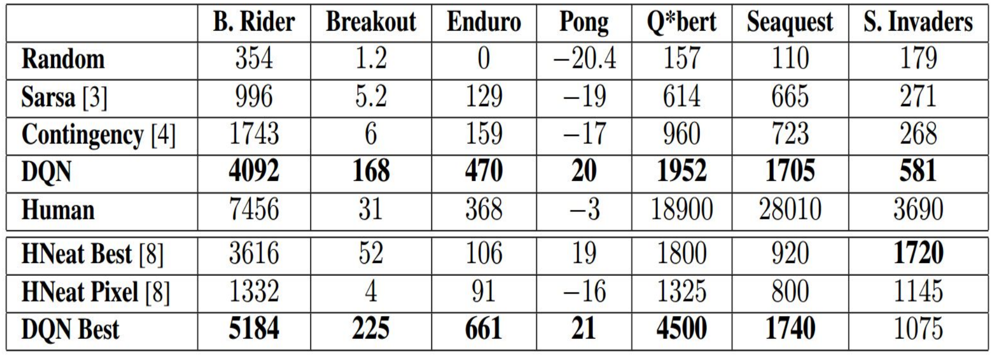






Evaluations

- Average Total Reward
- Best Performing Episode



	B. Rider	Breakout	Enduro	Pong	Q*bert	Seaquest	S. Invaders
Random	354	1.2	0	−20.4	157	110	179
Sarsa [3]	996	5.2	129	−19	614	665	271
Contingency [4]	1743	6	159	−17	960	723	268
DQN	**4092**	**168**	**470**	**20**	**1952**	**1705**	**581**
Human	7456	31	368	−3	18900	28010	3690
HNeat Best [8]	3616	52	106	19	1800	920	**1720**
HNeat Pixel [8]	1332	4	91	−16	1325	800	1145
DQN Best	**5184**	**225**	**661**	**21**	**4500**	**1740**	1075

Table 1: The upper table compares average total reward for various learning methods by running an ϵ-greedy policy with $\epsilon = 0.05$ for a fixed number of steps. The lower table reports results of the single best performing episode for HNeat and DQN. HNeat produces deterministic policies that always get the same score while DQN used an ϵ-greedy policy with $\epsilon = 0.05$.

Analysis

Strengths:

- Q-learning over non-linear function approximators
- End-to-End Learning over multiple games
- SGD Training
- Seminal paper

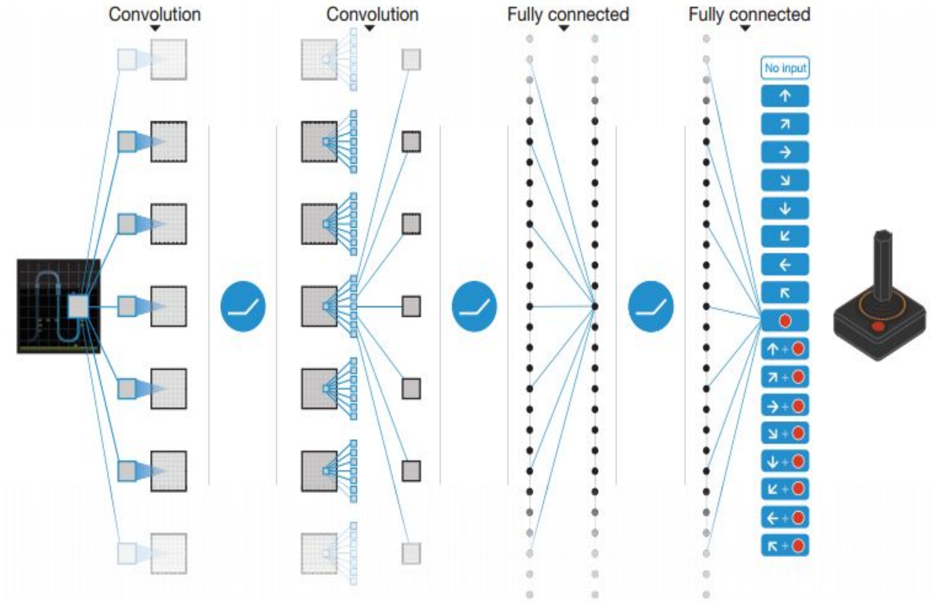
Weakness:

- DQN limited to finite discrete actions
- Long Training
- Reward Clipping

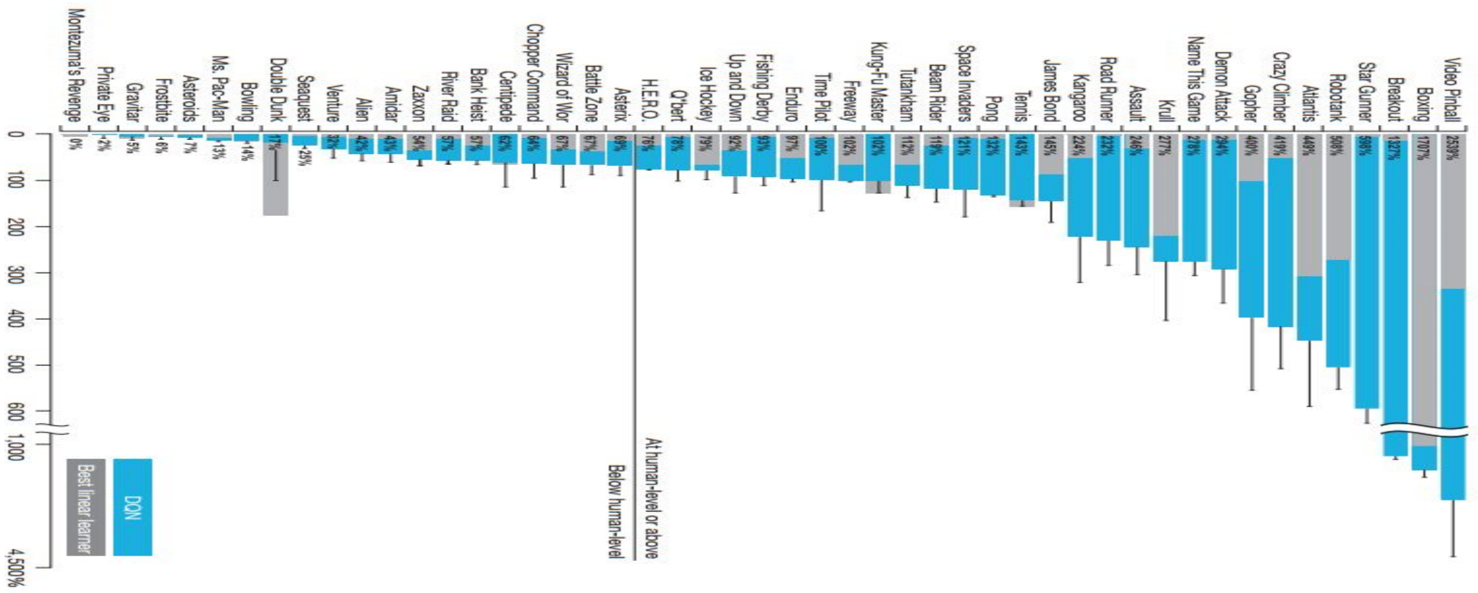
Human-level control through deep reinforcement learning

- Extending the DQN architecture to play 49 Atari 2600 arcade games
- No pretraining
- No game-specific training
- State: Transitions from 4 frames : Experience Replay
- Actions -18:
    - 9 directions of joystick
    - 9 directions + button
- Reward - Game Score

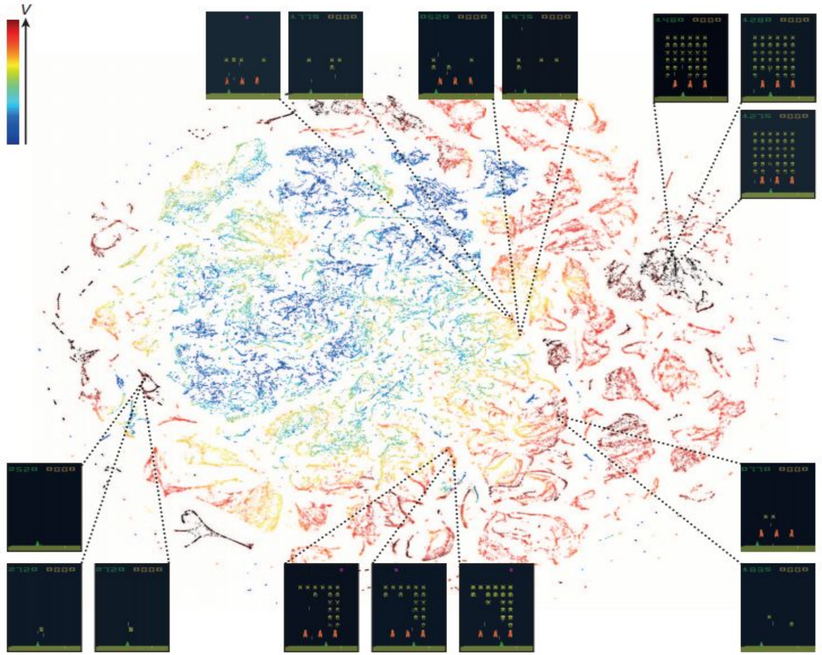
Architecture



Stats over 49 games



T-SNE Embedding (Last Hidden Layer)



Analysis

Strengths:

- End-to-End Learning over multiple games
- Beats human performance on most of the games
- Richer Rewards

Weakness:

- Long Training
- DQN limited to finite discrete actions

Brave New World

Optimizations on DQN Since Then

- Double DQN: Remove Upward bias caused by $\max_a Q(s,a, w)$
  - Current Q-network w: Used to select actions
  - Older Q-network w-: Used to evaluate actions
- Prioritized replay : Weight experience according to TD-error (suprise)
  - Store experience in priority queue according to DQN error
- Asynchronous RL
  - Joint Training using Parameter Sharing on Distributed Scale
  - GORILA

What are Policy Gradient Methods?

- Before: Learn the values of actions and then select actions based on their estimated action-values. The policy was generated directly from the value function
- We want to learn a parameterised policy that can select actions without consulting a value function. The parameters of the policy are called policy weights
- A value function may be used to learn the policy weights but this is not required for action selection
- Policy gradient methods are methods for learning the policy weights using the gradient of some performance measure with respect to the policy weights
- Policy gradient methods seek to maximise performance and so the policy weights are updated using gradient ascent

Policy-based Reinforcement Learning

- Search directly for the optimal policy π^*
- Can use any parametric supervised machine learning model to learn policies $\pi(a \mid s; \boldsymbol{\theta})$ where $\boldsymbol{\theta}$ represents the learned parameters
- Recall that the optimal policy is the policy that achieves maximum future return

Policy-Based RL

- Represent policies by deep networks instead of Q-function
  - $a = \pi(a \mid s, u)$ Stochastic Policies
  - $A = \pi(s, u)$ Deterministic Policies
  - where u is the parameters of the deep network
- Objective function for the network
  - $L(u) = E [r_1 + \gamma r_2 + \gamma^2 r_3 \mid \pi(. , u)]$
- SGD Optimization
- Allows Continuous and Discrete Control
- Known to get stuck in Local minima

Algorithms in Policy-Based RL

- REINFORCE
  - Episodic updates
  - Maximize the loss of expected reward under the objective
  - while (true) run_episode(policy) update(policy) end;
- Actor-Critic
  - Updates at each step
  - Critic approximates the value function
  - Actor approximates the policy
- Asynchronous Advantage Function Actor-Critic
  - Uses Advantage Function Estimate for state-actuib. A(s,a,w) = Q(s,a,w) -V(s)
  - Replacing the need of replay memory by using parallel agents running on CPU
  - Relies on different exploration behavior of the parallel agents
  - Outperforms the conventional method

What is Asynchronous Reinforcement Learning?

- Use asynchronous gradient descent to optimise controllers
- This is useful for deep reinforcement learning where the controllers are deep neural networks, which take a long time to train
- Asynchronous gradient descent speeds up the learning process
- Can use one multi-core CPU to train deep neural networks asynchronously instead of multiple GPUs

Parallelism (1)

- Asynchronously execute multiple agents in parallel on multiple instances of the environment
- This parallelism decorrelates the agents’ data into a more stationary process since at any given time-step, the agents will be experiencing a variety of different states
- This approach enables a larger spectrum of fundamental on-policy and off-policy reinforcement learning algorithms to be applied robustly and effectively using deep neural networks
- Use asynchronous actor-learners (i.e. agents). Think of each actor-learner as a thread
- Run everything on a single multi-core CPU to avoid communication costs of sending gradients and parameters

Parallelism (2)

- Multiple actor-learners running in parallel are likely to be exploring different parts of the environment
- We can explicitly use different exploration policies in each actor-learner to maximise this diversity
- By running different exploration policies in different threads, the overall changes made to the parameters by multiple actor-learners applying updates in parallel are less likely to be correlated in time than a single agent applying online updates

No Experience Replay

- No need for a replay memory. We instead rely on parallel actors employing different exploration policies to perform the stabilising role undertaken by experience replay in the DQN training algorithm
- Since we no longer rely on experience replay for stabilising learning, we are able to use on-policy reinforcement learning methods to train neural networks in a stable way

Video Demo : A3C Labryinth

Video Demo : DQN Doom

Scope / Future

- Multi-agent Deep RL
  - Share Parameters!!! (Naive approach)
- Hierarchical Deep Reinforcement Learning
  - Road to General AI!

Quotes from the Maestro

If intelligence was a cake, unsupervised learning would be the cake, supervised learning would be the icing on the cake, and reinforcement learning would be the cherry on the cake.

-Yann Lecun

Reference Zone

- http://www.cs.princeton.edu/~andyz/pacmanRL
- ICML Deep RL Tutorial : http://icml.cc/2016/tutorials/deep_rl_tutorial.pdf
- Andrej Karpathy’s blog: http://karpathy.github.io/
- Playing Atari with Deep Reinforcement Learning. V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, M. Riedmiller, NIPS workshop 2013
- Human-level control through deep reinforcement learning. Mnih et al. Nature 2015
- https://www.cs.princeton.edu/courses/archive/spring17/cos598F/lectures/RL.pptx