Advanced Machine Learning Summer 2019

Part 6 – Deep Reinforcement Learning 2

17.04.2019

Jonathon Luiten

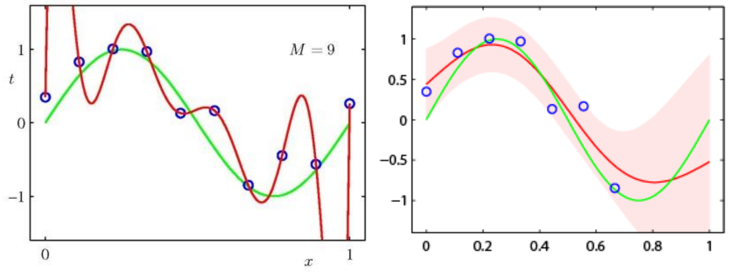
Prof. Dr. Bastian Leibe

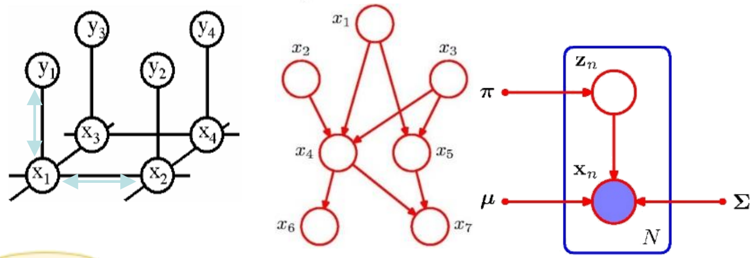
RWTH Aachen University, Computer Vision Group
http://www.vision.rwth-aachen.de

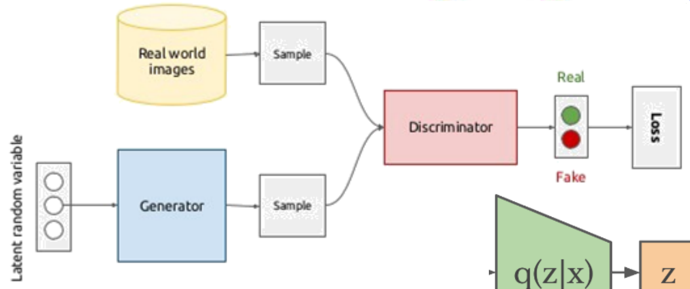
Course Outline

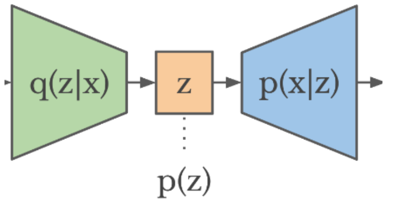
- Regression Techniques
  - Linear Regression
  - Regularization (Ridge, Lasso)
  - Kernels (Kernel Ridge Regression)
- Deep Reinforcement Learning
- Probabilistic Graphical Models
  - Bayesian Networks
  - Markov Random Fields
  - Inference (exact & approximate)
- Deep Generative Models
  - Generative Adversarial Networks
  - Variational Autoencoders

$$f : \mathcal{X} \rightarrow \mathbb{R}$$









Recap: Topics of the Last Lecture

- Reinforcement Learning
  - Introduction
  - Key Concepts
  - Optimal policies
  - Exploration-exploitation trade-off
- Temporal Difference Learning
  - SARSA
  - Q-Learning
- Deep Reinforcement Learning
  - Value based Deep RL
  - Policy based Deep RL

Note on Variables

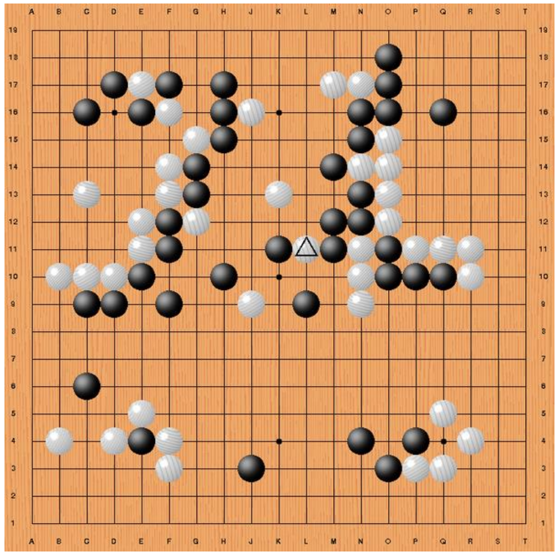
- Capital letters define a variable.
- Lower case letters define a value of a variable.
- 'Fancy' case letters define the set of possible values for a variable.
- Examples

$$p(s'|s,a) = \Pr\{S_{t+1} = s' | S_t = s, A_t = a\} = \sum_{r \in \mathcal{R}} p(s', r | s, a)$$

$$R_{t+1} \in \mathcal{R} \qquad\qquad S_t \in \mathcal{S}$$

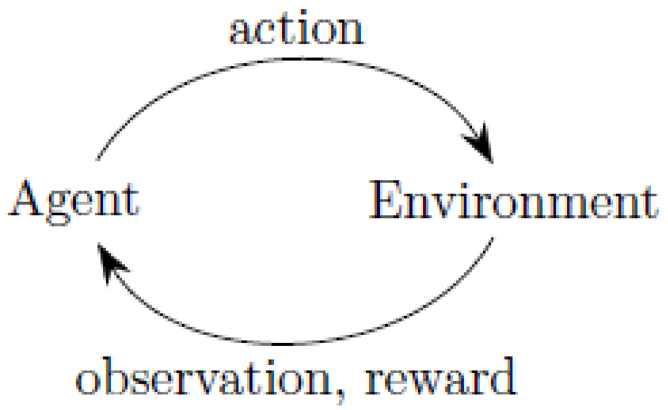
Recap: What is Reinforcement Learning?

- Learning how to act from a reinforcement signal.
- Humans do this too.
- And it works: Atari games, Alpha Go, Dota2/Starcraft, Drone Control, Robot Arm Manipulation, etc.



Recap: Reinforcement Learning

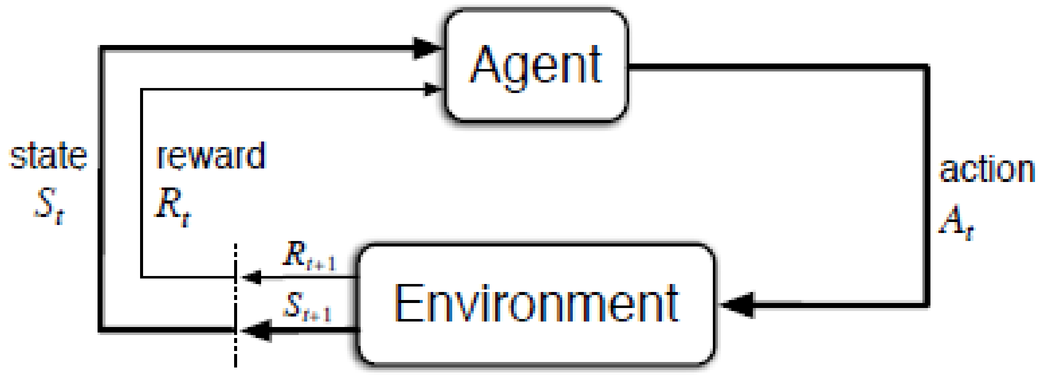
- Motivation
  - General purpose framework for decision making.
  - Basis: Agent with the capability to interact with its environment
  - Each action influences the agent's future state.
  - Success is measured by a scalar reward signal.
  - Goal: select actions to maximize future rewards.



  - Formalized as a partially observable Markov decision process (POMDP)

Slide adapted from: David Silver, Sergey Levine

Recap: The Agent–Environment Interface



- Let's formalize this
  - Agent and environment interact at discrete time steps $t = 0, 1, 2, \ldots$
  - Agent observes state at time t: $S_t \in \mathcal{S}$
  - Produces an action at time t: $A_t \in \mathcal{A}(S_t)$
  - Gets a resulting reward $R_{t+1} \in \mathcal{R} \subset \mathbb{R}$
  - And a resulting next state: S_{t+1}

Slide adapted from: Sutton & Barto

Recap: Reward vs. Return

- Objective of learning
  - We seek to maximize the expected return G_t as some function of the reward sequence $R_{t+1}, R_{t+2}, R_{t+3}, \ldots$
  - Standard choice: expected discounted return

$$G_t = R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \ldots = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1}$$

where $0 \leq \gamma \leq 1$ is called the discount rate.

- Difficulty
  - We don't know which past actions caused the reward.

$\Rightarrow$ Temporal credit assignment problem

Recap: Markov Decision Process (MDP)

- Markov Decision Processes
  - We consider decision processes that fulfill the Markov property.
  - I.e., where the environments response at time t depends only on the state and action representation at t.

- To define an MDP, we need to specify
  - State and action sets
  - One-step dynamics defined by state transition probabilities

$$p(s'|s,a) = \Pr\{S_{t+1} = s' | S_t = s, A_t = a\} = \sum_{r \in \mathcal{R}} p(s', r | s, a)$$

  - Expected rewards for next state-action-next-state triplets

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

Recap: Policy

- Definition
  - A policy determines the agent’s behavior
  - Map from state to action $\pi : \mathcal{S} \rightarrow \mathcal{A}$

- Two types of policies
  - Deterministic policy: $a = \pi(s)$
  - Stochastic policy: $\pi(a|s) = \Pr\{A_t = a | S_t = s\}$

- Note
  - $\pi(a|s)$ denotes the probability of taking action a when in state s.

Recap: Value Function

- Idea
  - Value function is a prediction of future reward
  - Used to evaluate the goodness/badness of states
  - And thus to select between actions

- Definition
  - The value of a state s under a policy π, denoted $v_\pi(s)$, is the expected return when starting in s and following π thereafter.

$$v_\pi(s) = \mathbb{E}_\pi[G_t | S_t = s] = \mathbb{E}_\pi[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1} | S_t = s]$$

  - The value of taking action a in state s under a policy π, denoted $q_\pi(s, a)$, is the expected return starting from s, taking action a, and following π thereafter.

$$q_\pi(s, a) = \mathbb{E}_\pi[G_t | S_t = s, A_t = a] = \mathbb{E}_\pi[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1} | S_t = s, A_t = a]$$

Recap: Bellman Equation

- Recursive Relationship
  - For any policy π and any state s, the following consistency holds

$$
\begin{aligned}
v_\pi(s) &= \mathbb{E}_\pi[G_t | S_t = s] \\
&= \mathbb{E}_\pi\left[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \middle| S_t = s\right] \\
&= \mathbb{E}_\pi\left[R_{t+1} + \gamma \sum_{k=0}^{\infty} \gamma^k R_{t+k+2} \middle| S_t = s\right] \\
&= \sum_a \pi(a|s) \sum_{s'} \sum_r p(s', r | s, a)\left[r + \gamma \mathbb{E}_\pi\left[\sum_{k=0}^{\infty} \gamma^k R_{t+k+2} \middle| S_{t+1} = s'\right]\right] \\
&= \sum_a \pi(a|s) \sum_{s',r} p(s', r | s, a)[r + \gamma v_\pi(s')], \qquad \forall s \in \mathcal{S}
\end{aligned}
$$

  - This is the Bellman equation for $v_\pi(s)$.

Recap: Optimal Value Functions

- For finite MDPs, policies can be partially ordered
  - There will always be at least one optimal policy π_*.
  - The optimal state-value function is defined as

$$v_*(s) = \max_\pi \mathrm{v}_\pi(\mathrm{s})$$

  - The optimal action-value function is defined as

$$q_*(s,a) = \max_\pi q_\pi(s,a)$$

Recap: Optimal Value Functions

- Bellman optimality equations
  - For the optimal state-value function v_*:

$$v_*(s) = \max_{a \in \mathcal{A}(s)} q_{\pi_*}(s,a)$$

$$= \max_{a \in \mathcal{A}(s)} \sum_{s',r} p(s',r|s,a)[r + \gamma v_*(s')]$$

  - v_* is the unique solution to this system of nonlinear equations.
  - For the optimal action-value function q_*:

$$q_*(s,a) = \sum_{s',r} p(s',r|s,a)\left[r + \gamma \max_{a'} q_*(s',a')\right]$$

  - q_* is the unique solution to this system of nonlinear equations.

$\Rightarrow$ If the dynamics of the environment $p(s',r|s,a)$ are known, then in principle one can solve those equation systems.

Recap: Optimal Policies

- Why optimal state-value functions are useful
  - *Any policy that is greedy w.r.t. v_* is an optimal policy.*

    $\Rightarrow$ Given v_*, one-step-ahead search produces the long-term optimal results.

    $\Rightarrow$ Given q_*, we do not even have to do one-step-ahead search

$$\pi_*(s) = \underset{a \in \mathcal{A}(s)}{\operatorname{argmax}}\, q_*(s,a)$$

- Challenge
  - Many interesting problems have too many states for solving v_*.
  - Many Reinforcement Learning methods can be understood as approximately solving the Bellman optimality equations, using actually observed transitions instead of the ideal ones.

Recap: Tabular vs. Approximate methods

- For problems with small discrete state and action spaces:
  - Value function or Policy function can be expressed as a table of values.
- If we cannot enumerate our states or actions we use function approximation.
  - Kernel methods
  - Deep Learning / Neural Networks
- Want to solve large problems with huge state spaces, e.g. chess: 10^{120} states.
- Tabular methods don't scale well - they're a lookup table
  - Too many states to store in memory
  - Too slow to learn value function for every state/state-action.

Recap: Model-based vs Model-free

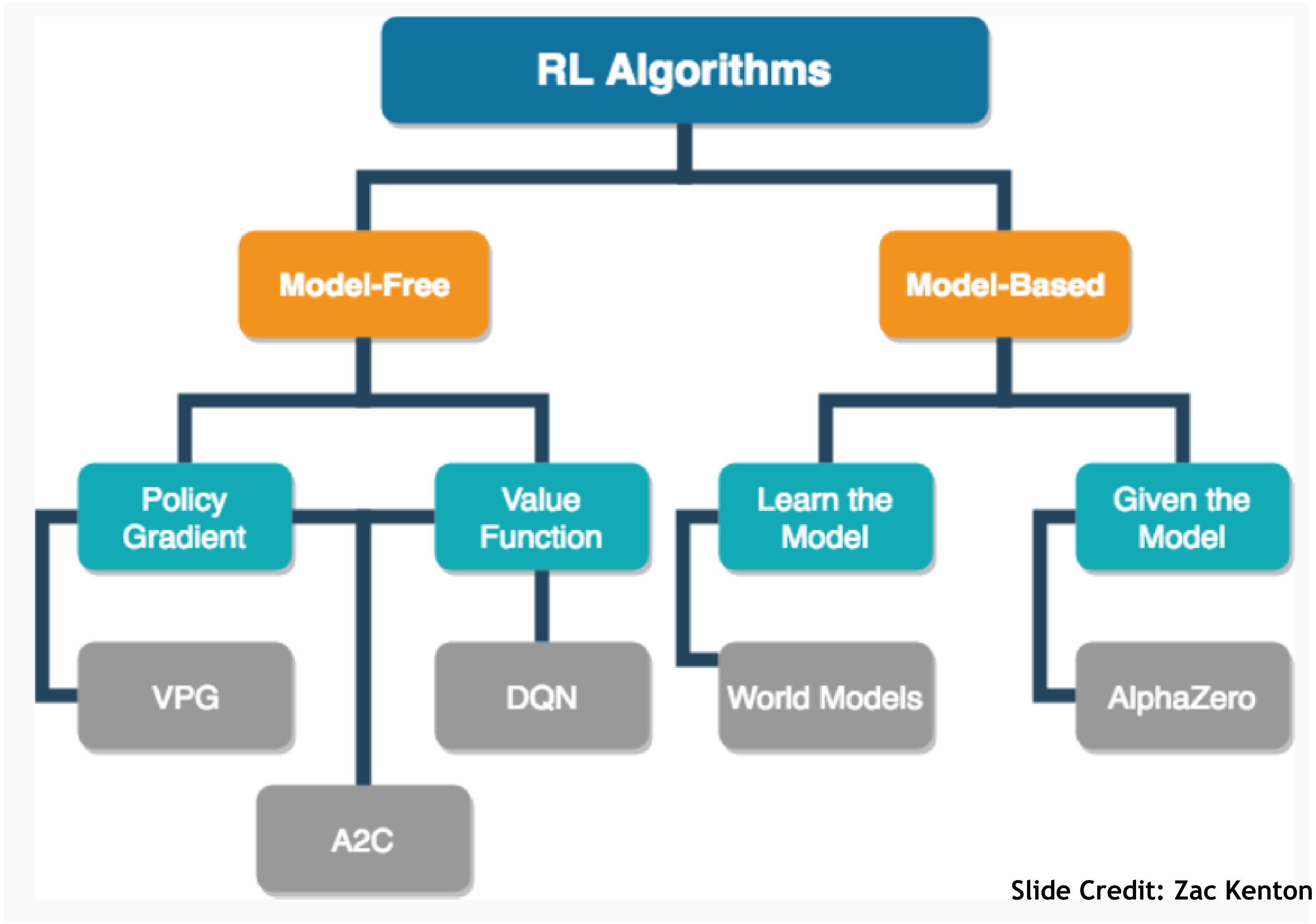
- Model-based
  - Has a model of the environment dynamics and reward
  - Allows agent to plan: predict state and reward before taking action
  - Pro: Better sample efficiency
  - Con: Agent only as good as the environment - Model-bias

- Model-free
  - No explicit model of the environment dynamics and reward
  - Less structured. More popular and further developed and tested.
  - Pro: Can be easier to implement and tune
  - Cons: Very sample inefficient

Recap: Value-based RL vs Policy-based RL

- RL methods can directly estimate a policy: Policy Based
  - A direct mapping of what action to take in each state.
  - $\pi(a|s) = \mathrm{P}(\mathrm{a}|\mathrm{s}, \theta)$
- RL methods can estimate a value function and derive a policy from that: Value Based
  - Either a state-value function
    - $\hat{V}(s;\theta) \approx V^{\pi}(\mathrm{s})$
  - Or an action-state value function (q function)
    - $\hat{Q}(s,a;\theta) \approx Q^{\pi}(\mathrm{s},\mathrm{a})$
- Or both simultaneously: Actor-Critic
  - Actor-Critic methods learn both a policy (actor) and a value function (critic)

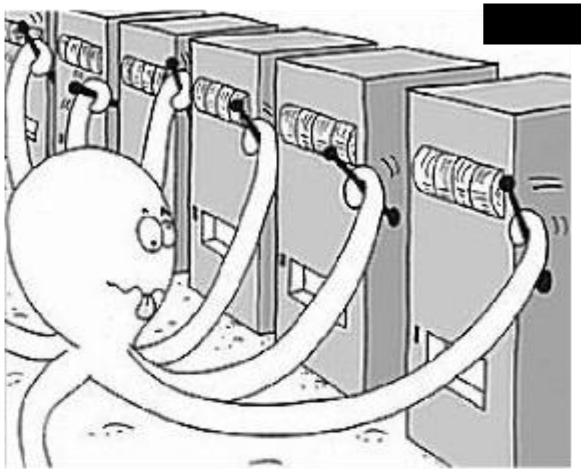
Recap: Taxonomy of RL methods



Slide Credit: Zac Kenton

Recap: Exploration-Exploitation Trade-off

- Example: N-armed bandit problem
  - Suppose we have the choice between N actions $a_1, \ldots, a_N$.
  - If we knew their value functions $q_*(s, a_i)$, it would be trivial to choose the best.
  - However, we only have estimates based on our previous actions and their returns.


[redacted]

- We can now
  - Exploit our current knowledge
    - And choose the greedy action that has the highest value based on our current estimate.
  - Explore to gain additional knowledge
    - And choose a non-greedy action to improve our estimate of that action's value.

Image source: research.microsoft.com

Recap: Simple Action Selection Strategies

- ϵ-greedy
  - Select the greedy action with probability $(1-\epsilon)$ and a random one in the remaining cases.
  - $\Rightarrow$ In the limit, every action will be sampled infinitely often.
  - $\Rightarrow$ Probability of selecting the optimal action becomes $> (1-\epsilon)$.
  - But: many bad actions are chosen along the way.

- Softmax
  - Choose action a_i at time t according to the softmax function

$$\frac{e^{q_t(a_i)/\tau}}{\sum_{j=1}^{N} e^{q_t(a_j)/\tau}}$$

    where τ is a temperature parameter (start high, then lower it).
  - Generalization: replace q_t by a preference function H_t that is learned by stochastic gradient ascent (“gradient bandit”).

Recap: On-Policy vs. Off-Policy

- On-policy methods
  - Attempt to evaluate or improve the policy used to make decisions.
  - “Learn while on the job”

- Off-policy methods
  - Policy used to generate behavior (behavior policy) is unrelated to the policy that is evaluated and improved (estimation policy)
  - Can we learn the value function of a policy given only experience “off” the policy?
  - “Learn while looking over someone else’s shoulder”

Recap: Policy Evaluation

- Policy evaluation (the prediction problem)
  - How good is a given policy?
  - For a given policy π, compute the state-value function v_π.
  - Once we know how good a policy is, we can use this information to improve the policy

- If we know the model:
  - $V_{k+1}^{\pi}(s_t) =$

$$\sum_{a_t} \pi(a_t \mid s_t) \sum_{s_{t+1}} p(s_{t+1} \mid s_t, a_t) \left(r(s_t, a_t, s_{t+1}) + \gamma V_k^{\pi}(s_{t+1})\right)$$

  - This can be shown to converge to the actual V^π as $K \to \infty$

Recap: Policy Evaluation

- If we do not know the model, then we have to approximate it using observations
- One option: Monte-Carlo methods
  - Play through a sequence of actions until a reward is reached, then backpropagate it to the states on the path.
  - Update after whole sequence (episodic)

$$V(S_t) \leftarrow V(S_t) + \alpha[G_t - V(S_t)]$$ Target: the actual return after time t

- Or: Temporal Difference Learning (TD Learning) – TD(λ)
  - Directly perform an update using the estimate $V(S_{t+\lambda+1})$.
  - Bootstraps the current estimate of the value function
  - Can update every step

$$V(S_t) \leftarrow V(S_t) + \alpha[\underbrace{R_{t+1} + \gamma V(S_{t+1})}_{\text{Target: an estimate of the return (here: TD(0))}} - V(S_t)]$$

Recap: SARSA: On-Policy TD Control

- Idea
  - Turn the TD idea into a control method by always updating the policy to be greedy w.r.t. the current estimate

- Procedure
  - Estimate $q_\pi(s,a)$ for the current policy π and for all states s and actions a.
  - TD(0) update equation

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

  - This rule is applied after every transition from a nonterminal state S_t.
  - It uses every element of the quintuple $(S_t, A_t, R_{t+1}, S_{t+1}, A_{t+1})$.

    $\Rightarrow$ *Hence, the name SARSA*.

Image source: Sutton & Barto

Recap: Q-Learning: Off-Policy TD Control

- Idea
  - Directly approximate the optimal action-value function q_*, independent of the policy being followed.

- Procedure
  - TD(0) update equation

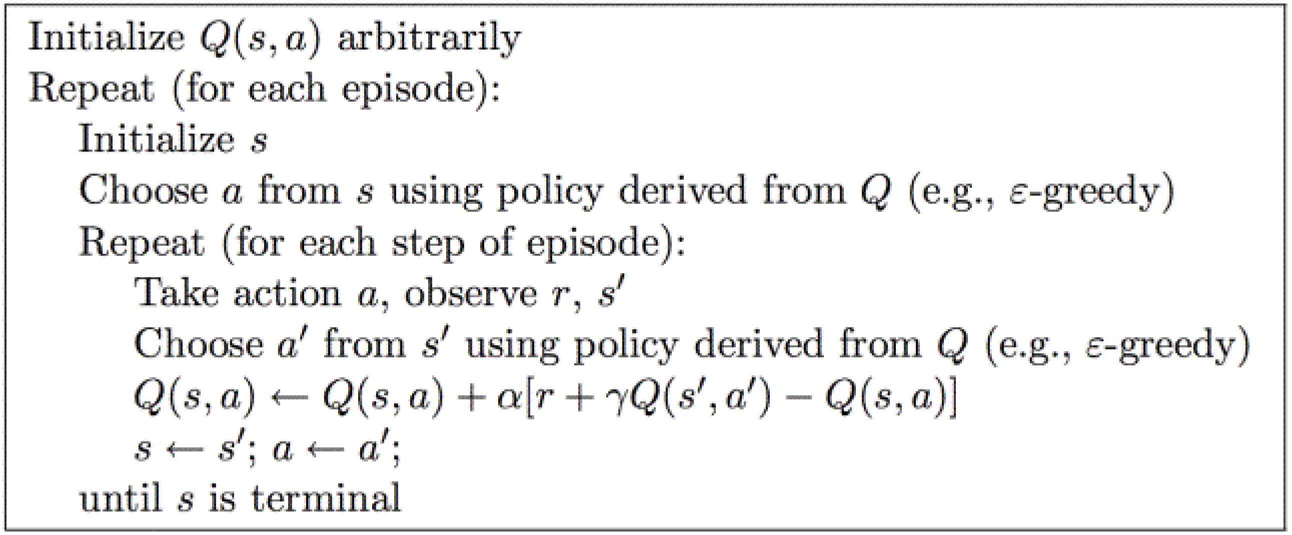
$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha \left[R_{t+1} + \gamma \max_a Q(S_{t+1}, a) - Q(S_t, A_t) \right]$$

  - Dramatically simplifies the analysis of the algorithm.
  - All that is required for correct convergence is that all pairs continue to be updated.

Image source: Sutton & Barto

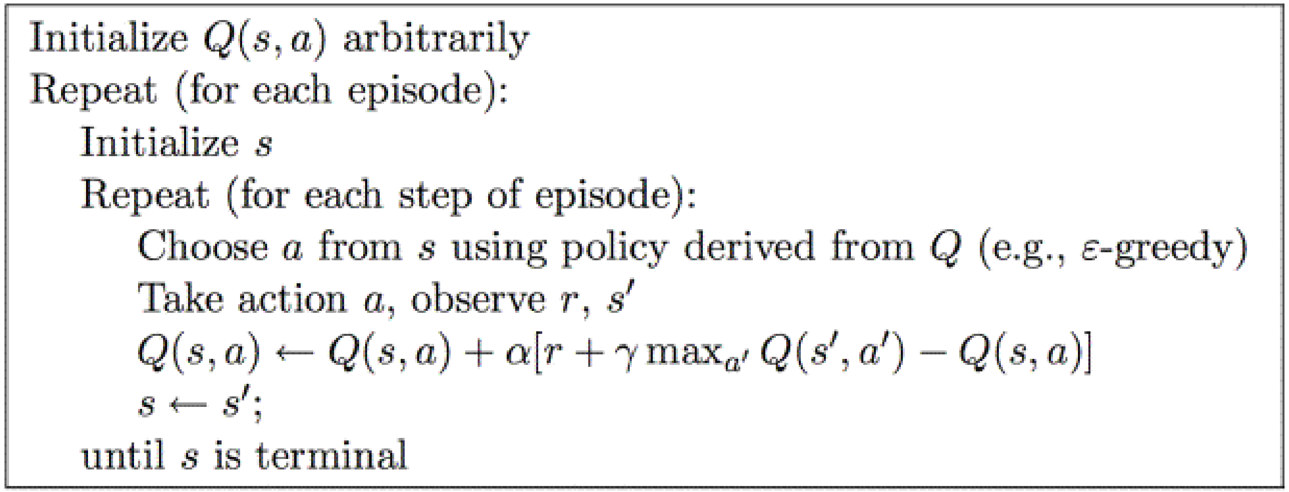
Recap: SARSA vs Q-Learning

- SARSA



```
Initialize $Q(s,a)$ arbitrarily
Repeat (for each episode):
    Initialize $s$
    Choose $a$ from $s$ using policy derived from $Q$ (e.g., $\varepsilon$-greedy)
    Repeat (for each step of episode):
        Take action $a$, observe $r$, $s'$
        Choose $a'$ from $s'$ using policy derived from $Q$ (e.g., $\varepsilon$-greedy)
        $Q(s,a) \leftarrow Q(s,a) + \alpha[r + \gamma Q(s',a') - Q(s,a)]$
        $s \leftarrow s'$; $a \leftarrow a'$;
    until $s$ is terminal
```

- Q-Learning



```
Initialize $Q(s,a)$ arbitrarily
Repeat (for each episode):
    Initialize $s$
    Repeat (for each step of episode):
        Choose $a$ from $s$ using policy derived from $Q$ (e.g., $\varepsilon$-greedy)
        Take action $a$, observe $r$, $s'$
        $Q(s,a) \leftarrow Q(s,a) + \alpha[r + \gamma \max_{a'} Q(s',a') - Q(s,a)]$
        $s \leftarrow s'$;
    until $s$ is terminal
```

Image source: Sutton & Barto

Topics of This Lecture

- Reinforcement Learning
  - Introduction
  - Key Concepts
  - Optimal policies
  - Exploration-exploitation trade-off
- Temporal Difference Learning
  - SARSA
  - Q-Learning
- Deep Reinforcement Learning
  - Value based Deep RL
  - Policy based Deep RL

Deep Reinforcement Learning

- RL using deep neural networks to approximate functions
  - Value functions
    - Measure goodness of states or state-action pairs
  - Policies
    - Select next action
  - Dynamics Models
    - Predict next states and rewards

Slide credit: Sergej Levine

Deep Reinforcement Learning

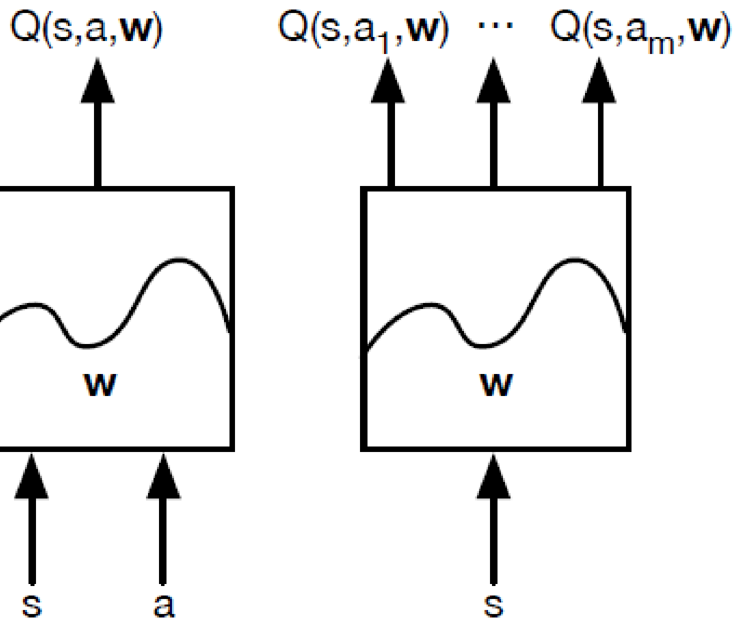
- Use deep neural networks to represent
  - Value function
  - Policy
  - Model
- Optimize loss function by stochastic gradient descent

Slide credit: David Silver

Q-Networks

- Represent value function by Q-Network with weights $\mathbf{w}$

$$Q(s, a, \mathbf{w}) = Q_*(s, a)$$



Slide credit: David Silver

Deep Q-Learning

- Idea
  - Optimal Q-values should obey Bellman equation

$$Q_*(s,a) = \mathbb{E}\left[r + \gamma \max_{a'} Q(s',a') \,|s,a\right]$$

  - Treat the right-hand side $r + \gamma \max_{a'} Q(s',a',\mathbf{w})$ as a target
  - Minimize MSE loss by stochastic gradient descent

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

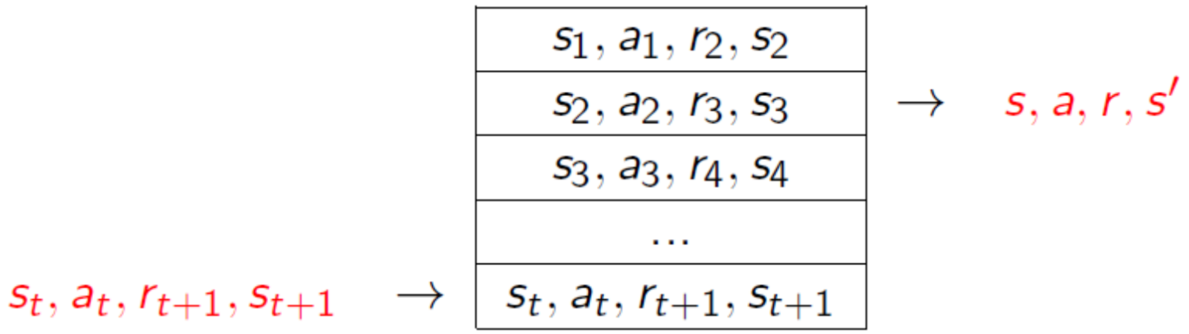
  - This converges to Q_* using a lookup table representation.

  - Unfortunately, it diverges using neural networks due to
    - Correlations between samples
    - Non-stationary targets

Slide adapted from David Silver

Deep Q-Networks (DQN): Experience Replay

- Adaptations
  - To remove correlations, build a dataset from agent’s own experience


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'
$s_t, a_t, r_{t+1}, s_{t+1}$ $\rightarrow$

  - Perform minibatch updates to samples of experience drawn at random from the pool of stored samples
    - $(s, a, r, s') \sim U(D)$ where $D = \{(s_t, a_t, r_{t+1}, s_{t+1})\}$ is the dataset
  - Advantages
    - Each experience sample is used in many updates (more efficient)
    - Avoids correlation effects when learning from consecutive samples
    - Avoids feedback loops from on-policy learning

Slide adapted from David Silver

Deep Q-Networks (DQN): Experience Replay

- Adaptations
  - Sample from the dataset and apply an update

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

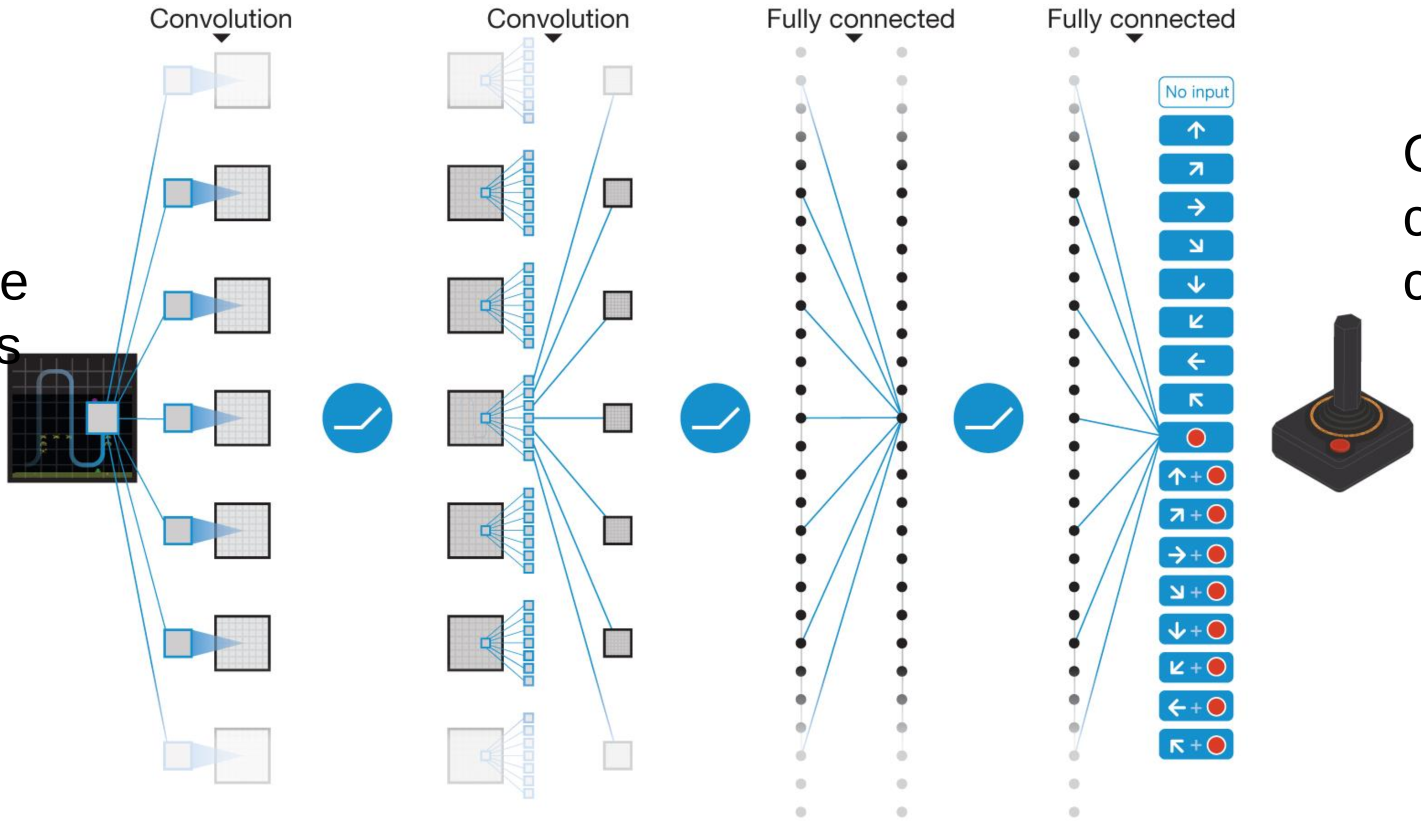
  - To deal with non-stationary parameters $\mathbf{w}^-$, are held fixed.
    - Only update the target network parameters every C steps.
    - I.e., clone the network Q to generate a target network $\hat{Q}$.

      $\Rightarrow$ Again, this reduces oscillations to make learning more stable.

Deep Reinforcement Learning

- Application: Learning to play Atari games

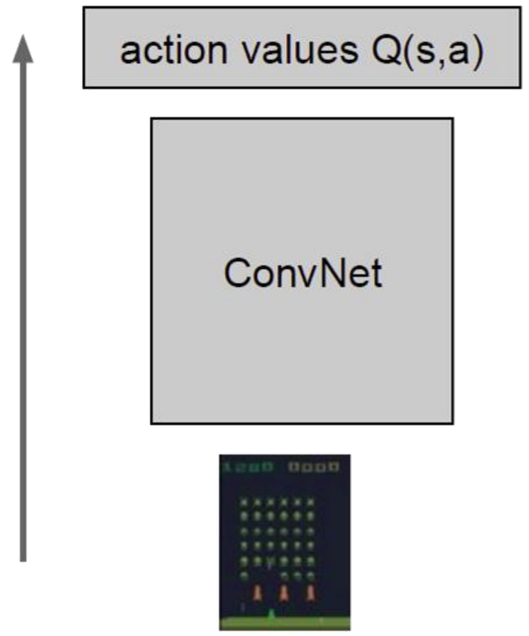
Input: pixels +game scores



Output: control commands

V. Mnih et al., Human-level control through deep reinforcement learning, Nature Vol. 518, pp. 529-533, 2015

Idea Behind the Model



- Interpretation
  - Assume finite number of actions
  - Each number here is a real-valued quantity that represents the Q function in Reinforcement Learning
- Collect experience dataset:
  - Set of tuples {(s,a,s',r), … }
  - (State, Action taken, New state, Reward received

L2 Regression Loss

$$L_i(\theta_i) = \mathbb{E}_{(s,a,r,s')\sim \mathrm{U}(D)}\left[\left(\underbrace{\boxed{r+\gamma \max_{a'} Q(s',a';\theta_i^-)}}_{\text{target value}} - \underbrace{\boxed{Q(s,a;\theta_i)}}_{\text{predicted value}}\right)^2\right]$$

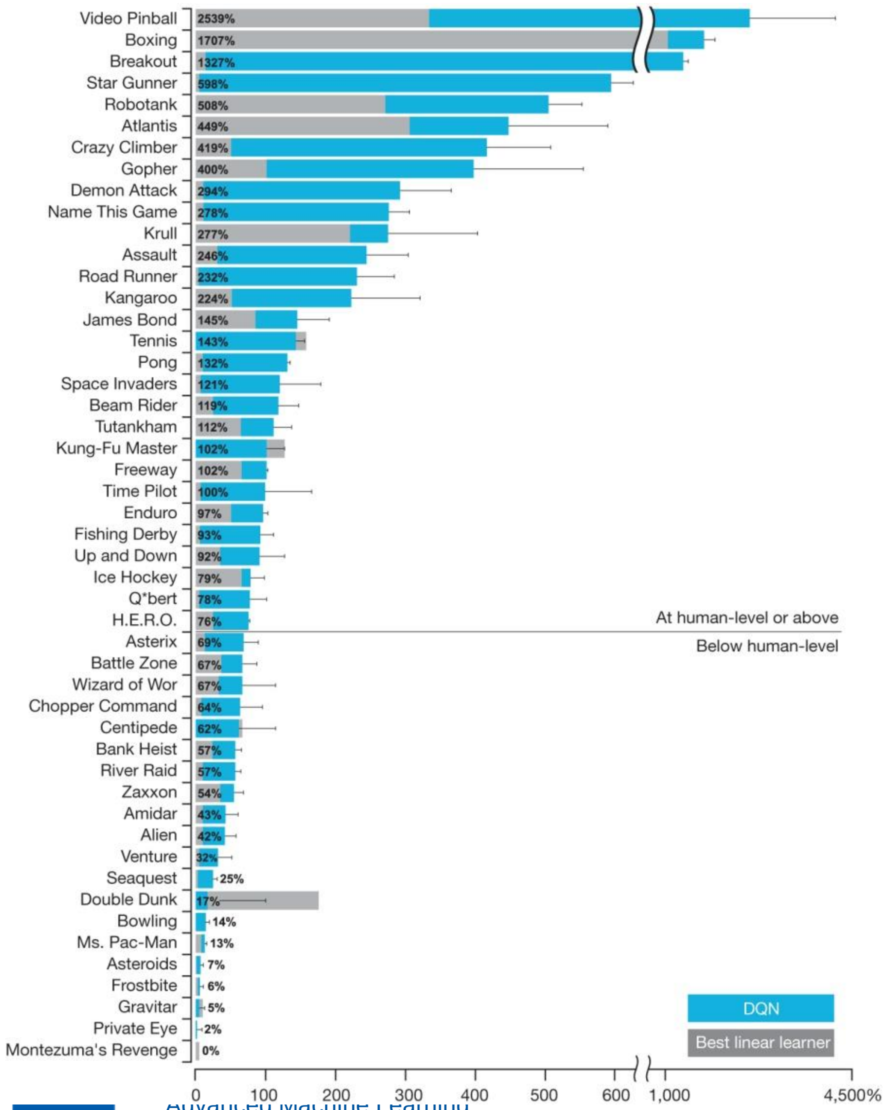
Current reward + estimate of future reward, discounted by γ

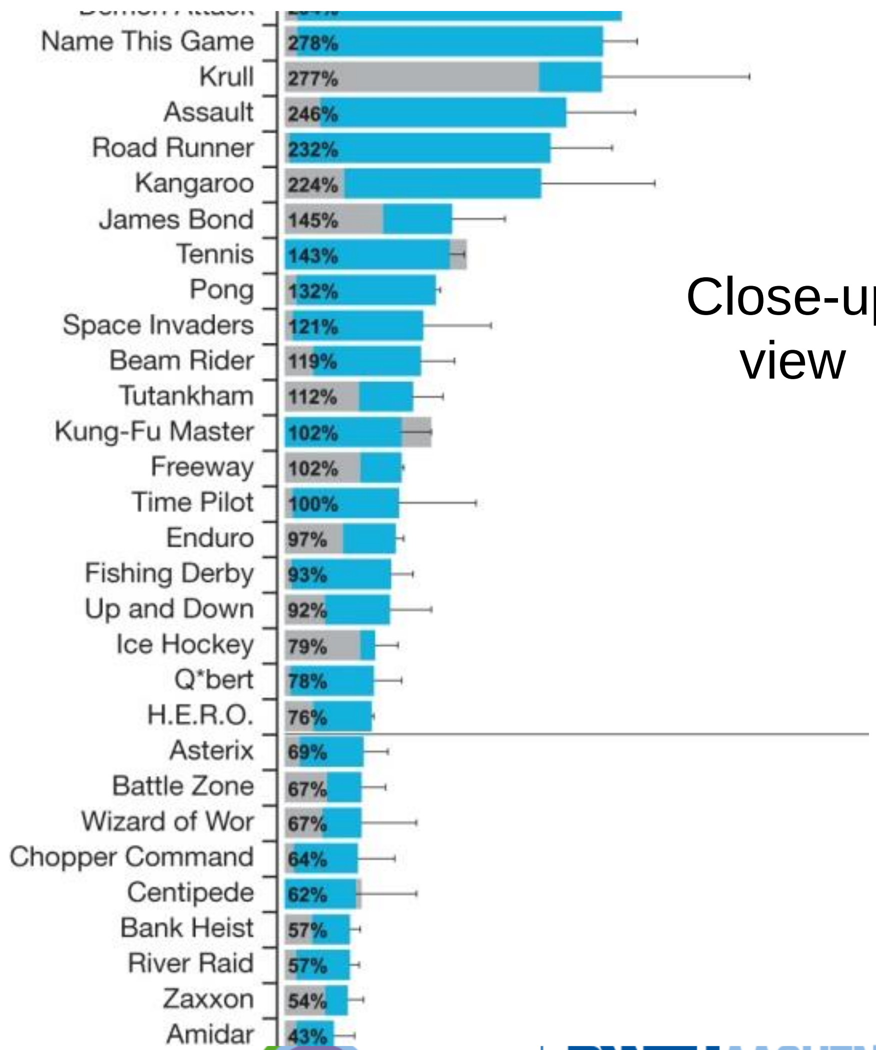
Slide credit: Andrej Karpaty

Results: Space Invaders

Results: Breakout

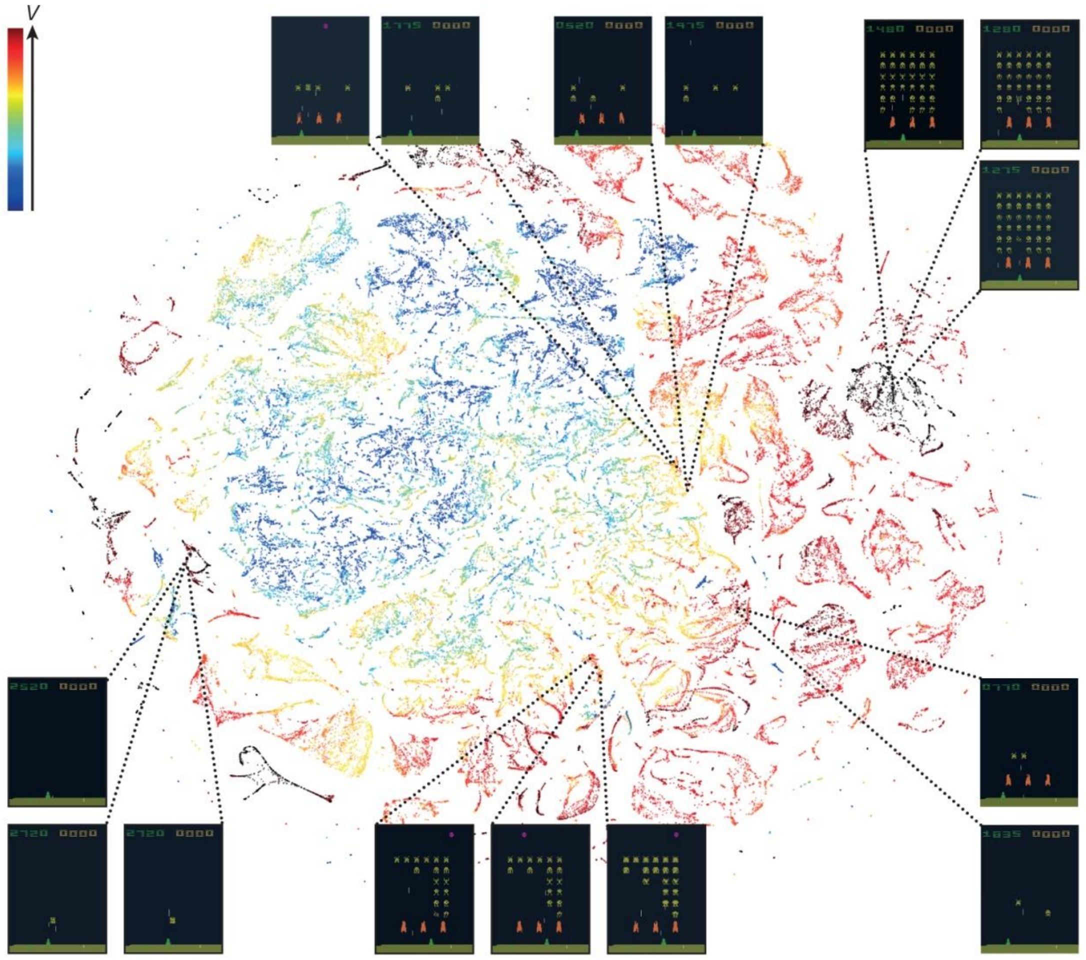
Comparison with Human Performance





Close-up view

Learned Representation



- t-SNE embedding of DQN last hidden layer (Space Inv.)

Improvements since Nature DQN

- Double DQN
  - Remove upward bias caused by $\max_a Q(s, a, \mathbf{w})$
  - One Q-network w is used to select actions
  - Another Q-network w^- is used to evaluate actions

$$L(\mathbf{w}) = \left(r + \gamma Q\left(s', \operatorname*{argmax}_a Q(s', a', \mathbf{w}), \mathbf{w}^-\right) - Q(s, a, \mathbf{w})\right)^2$$

- Prioritised replay
  - Weight experience according to surprise
  - Store experience in priority queue according to DQN error

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

$\Rightarrow$ Emphasize state transitions from which one can learn the most.

Improvements since Nature DQN (2)

- Duelling network
  - Split Q-network into two channels
  - Action-independent value function $V(s, v)$
  - Action-dependent advantage function $A(s, a, \mathbf{w})$

$$Q(s, a) = V(s, v) + A(s, a, \mathbf{w})$$

  - Intuition: network can learn which states are valuable without having to learn the effect of each action for each state.

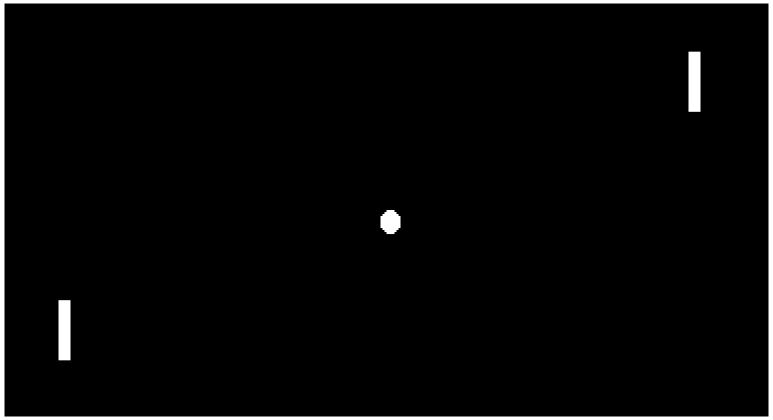
- Combined Algorithm
  - 3× mean Atari score vs. Nature DQN

Slide adapted from David Silver

Topics of This Lecture

- Reinforcement Learning
  - Introduction
  - Key Concepts
  - Optimal policies
  - Exploration-exploitation trade-off
- Temporal Difference Learning
  - SARSA
  - Q-Learning
- Deep Reinforcement Learning
  - Value based Deep RL
  - Policy based Deep RL

Policy Gradients: Learning to Play Pong from Pixels



MDP for Pong

State space:

All possible combinations of pixel values for a 210x60x3 image with each being an integer between 0 and 255

Action space:

Move paddle UP or DOWN (binary)

Transition Model:

The ball bounces around, folowing the 'laws of the game‘

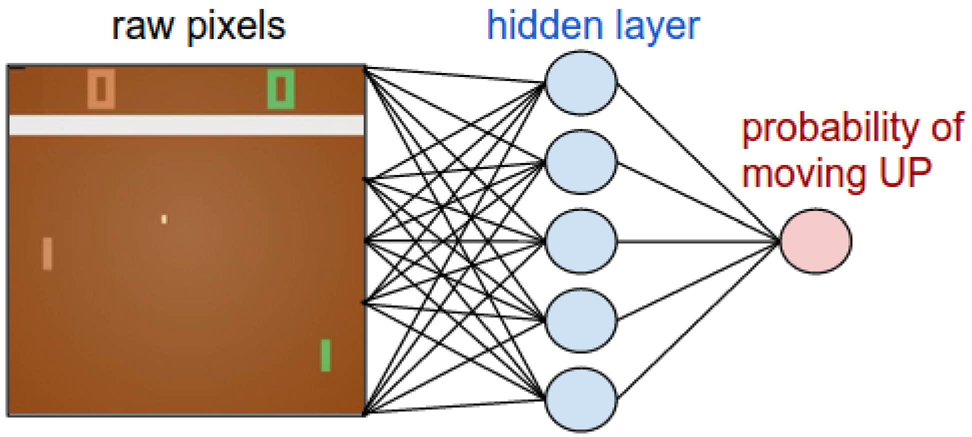
Reward Model:

+1 if ball goes behind other players paddle, -1 behind ours

Slide credit: Andrej Karpaty

Policy Network Example

Input: Pixel values (100,800 in total: 210x160x3)
Output: Probability of going UP (single number)
Stochastic: Only outputs probability which we use to sample
Example Network: Two fully-connected layers with weights W1,W2 with Sigmoid function on the output.
Preprocessing: Ideally we want to feed in multiple frames pixel values into the network so that it can learn to detect motion. Here we will feed *difference frames* to the network (current-previous).



Slide credit: Andrej Karpaty

This is hard

- We get 100,800 numbers as input.
  - Feed through our network with millions of parameters
  - Our network will decide UP or DOWN.
  - We repeat for hundreds of timesteps before anything happens.
  - Then let’s say eventually we get a +1 reward.
  - How can we know which actions caused this to happen?

- Credit Assignment Problem
  - The cause of the +ve reward: ball bounced from our paddle on a good trajectory.
  - But this was maybe 20 frames ago, and since then every action we took has zero effect.
  - How can we learn to do that more often: Ooooooof Hard

Slide credit: Andrej Karpaty

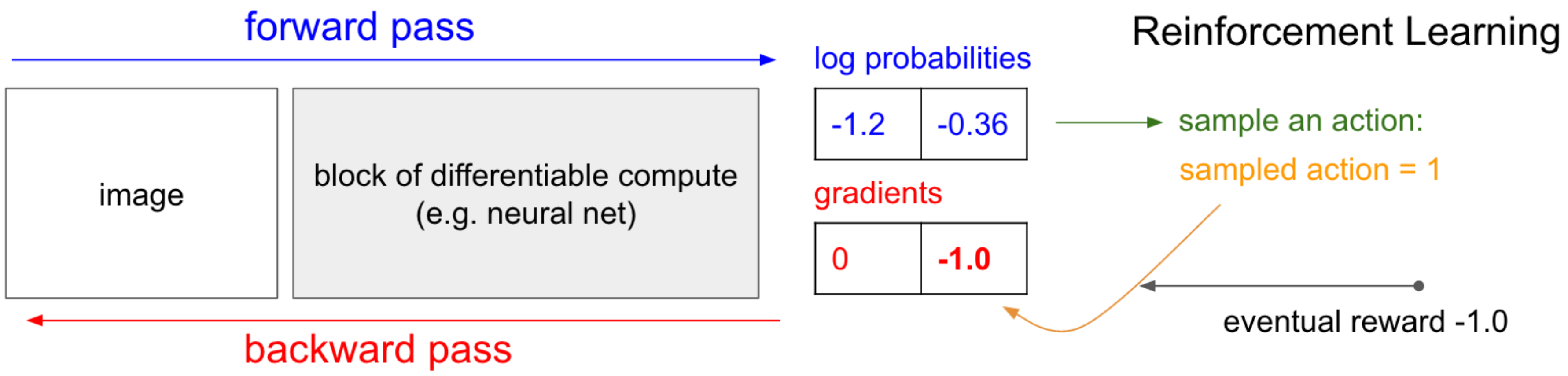
Simple Policy Gradient Solution

- Supervised Learning:
  - Predict output -> Make correct output more likely
- Reinforcement Learning:
  - Predict action -> Make action taken more likely if resulted in good reward.
- We can take our predicted action, UP or DOWN, and make this more or less likely by assigning a positive or negative score to this action.

- E.g. we can use the Reward.
  - After finishing one round of pong, we can see if we got +1 or -1 reward at the end of the game. And use these scores to score every decision we made in that round.
  - +1 will encourage the network to make that action in the future when presented with that state. -1 will discourage it.

Slide credit: Andrej Karpaty

Simple Policy Gradient Solution



- Very simple. And works.
- After playing MANY games. Network will learn good moves.

Slide credit: Andrej Karpaty

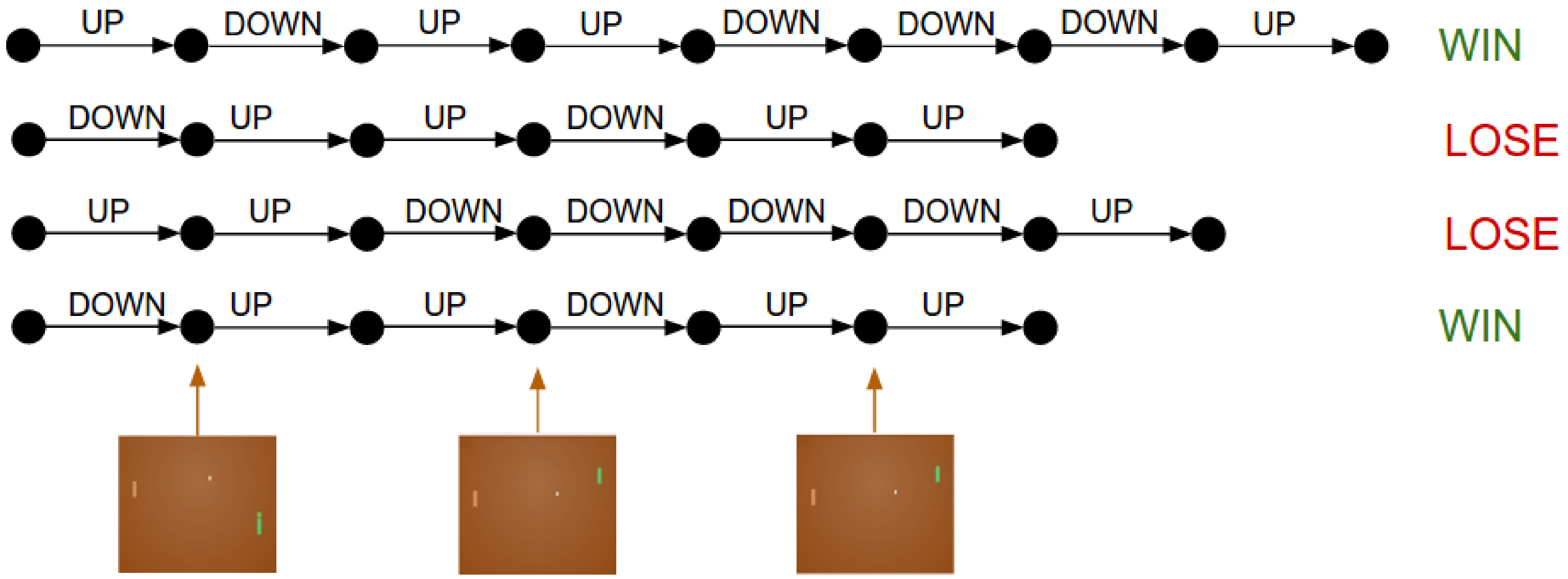
Training protocol

- Initialize network with random W1 and W2.
- Play 100 games of Pong (policy rollouts).
  - At the end of each game (approx. 200 frames) we get reward +1 or -1 if we won or lost.

- Let's say we won 12 games, and lost 88.
  - Now we take all decisions in the games we won (~200*12) and use +1 to weight the gradients.
  - Use -1 to weight the gradients in the other (~200*88) games.
  - Use Gradient Ascent (or another optimizer) to optimizer to optimize the weights of our NN to make the decisions from the won games more likely, and those from the lost games less likely.

Slide credit: Andrej Karpaty

Training protocol



Slide credit: Andrej Karpaty

Making this work better

- In general using the reward directly doesn't work so well.
- Instead we use Return. The average discounted future reward as we have seen earlier.

$$G_t = R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \ \dots = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1}$$

  - This is equivalent to using the Q-value to weight our gradients.
  - Q-value is the return (future discounted reward) for being in a particular state and performing a particular action.
- Thus policy gradients can be seen as directly learning a policy that maximizes the Q-value.

Slide credit: Andrej Karpaty

Making this work even better

- We can actually do something better than optimizing the Q-value. We can optimize the advantage function.

$$Adv(s,a) = Q(s,a) - V(s)$$

- The advantage function indicates the ‘Advantage’ of performing action a in state s, compared to the expected outcome from being in state s.
  - In practice, this results in more stable gradients and faster learning.
  - This can be computed here by standardizing the returns (subtract mean, divide by standard deviation), over a running average.

Slide credit: Andrej Karpaty

Deriving Policy Gradients

- Policy gradients are a special case of the more general *function gradient estimator*.
- We have a function in the form $E_{x\sim p(x|\theta)}[f(x)]$
- We are interested in how we can shift the distribution (defined through parameters θ) to increase it's score as judged by f

$$
\begin{aligned}
\nabla_\theta E_x[f(x)] &= \nabla_\theta \sum_x p(x) f(x) && \text{definition of expectation} \\
&= \sum_x \nabla_\theta p(x) f(x) && \text{swap sum and gradient} \\
&= \sum_x p(x) \frac{\nabla_\theta p(x)}{p(x)} f(x) && \text{both multiply and divide by } p(x) \\
&= \sum_x p(x) \nabla_\theta \log p(x) f(x) && \text{use the fact that } \nabla_\theta \log(z) = \frac{1}{z}\nabla_\theta z \\
&= E_x[f(x) \nabla_\theta \log p(x)] && \text{definition of expectation}
\end{aligned}
$$

Slide credit: Andrej Karpaty

What does this mean?

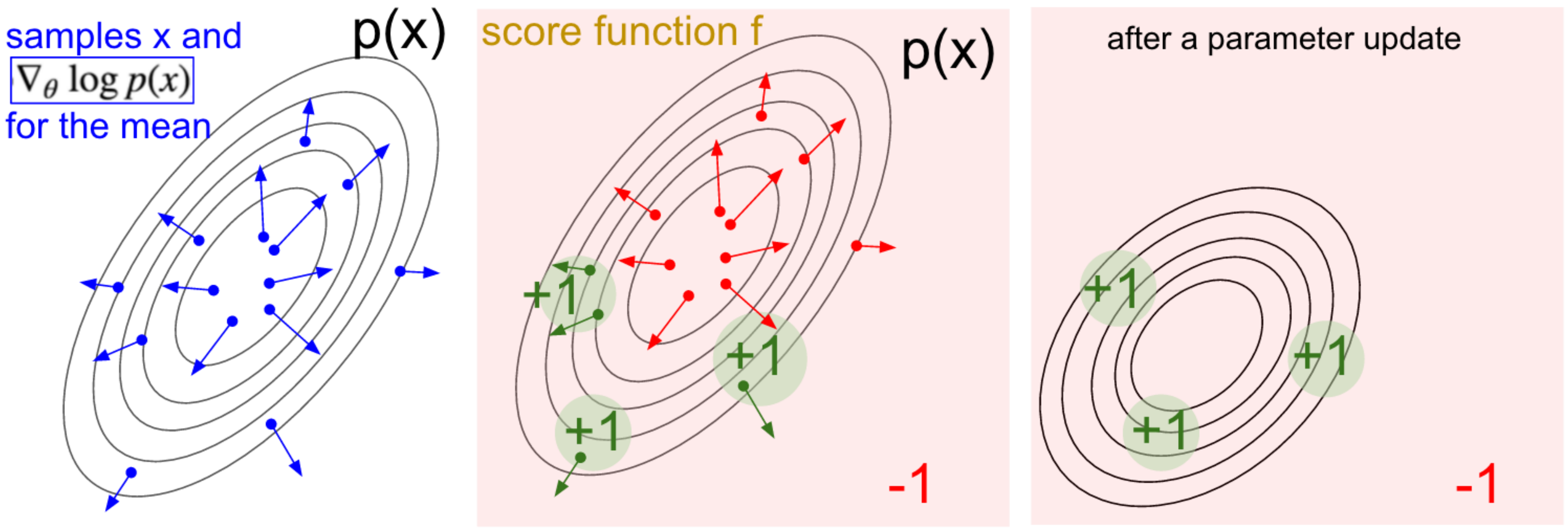
$$E_{x\sim p(x|\theta)}[f(x)] = E_x[f(x)\nabla_\theta \log p(x)]$$

$$\nabla_\theta \log p(x;\theta)$$

- This part is a vector which gives us the direction in the parameter space which would result in an increased probability of x
- By multiplying this by a scaler valued f we can define how we need to change the parameters to result in actions which have a high f.

Slide credit: Andrej Karpaty

Visual Example



Slide credit: Andrej Karpaty

The math for Policy Gradients

– Optimize the value function of the policy

$$J(\theta) = V^{\pi_\theta}(s_0)$$

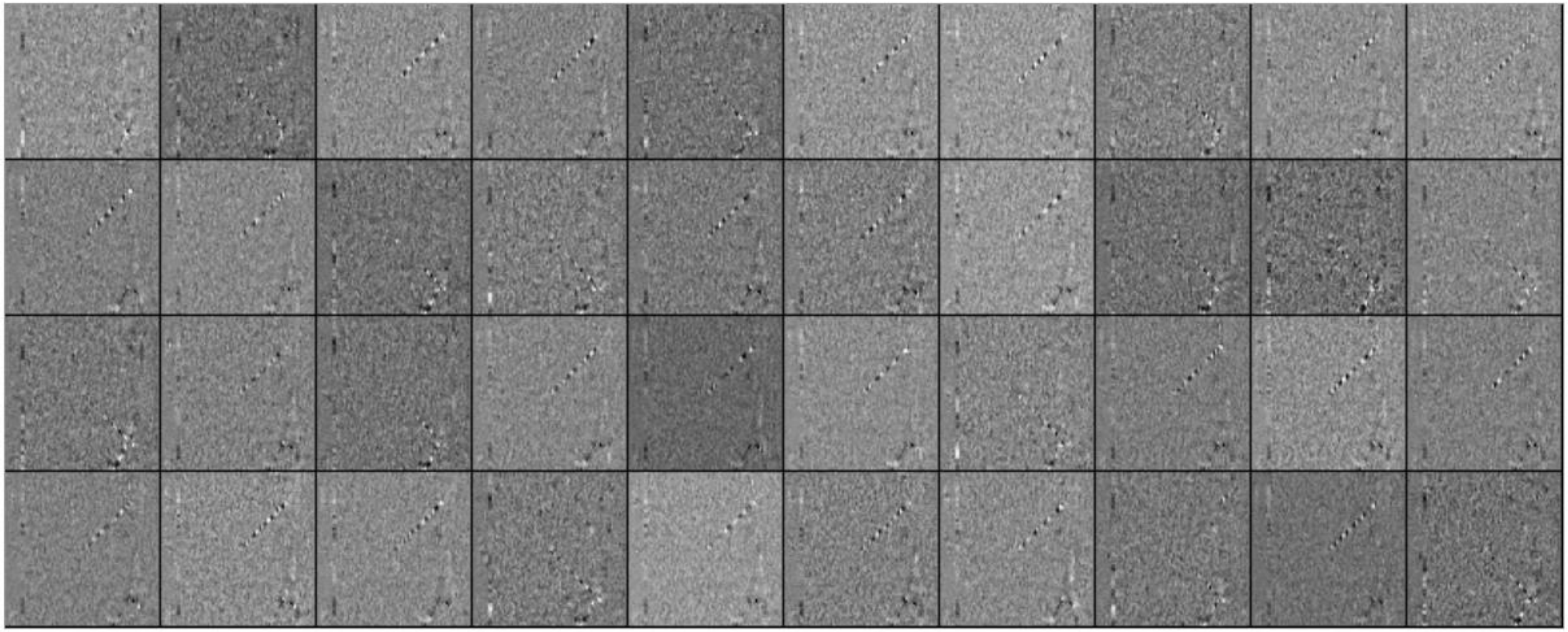
– For any differentiable policy, the policy gradient is

$$\nabla_\theta J(\theta) = \mathbb{E}_{\pi_\theta}\left(\nabla_\theta \log \pi_\theta(a_t \mid s_t)\, Q^{\pi_\theta}(s_t, a_t)\right))$$

(policy gradient theorem)

What is learnt

- Learnt weights of 40 (from 200) neurons in W1.



Slide credit: Andrej Karpaty

Algorithm – Monte-Carlo Policy Gradient

- Execute policy to obtain sample episodes
- Update parameters by stochastic gradient ascent using the policy gradient theorem

- REINFORCE algorithm
  - Initialize parameters arbitrarily
  - Repeat:
    - Sample episode $(s_0, a_0, r_1, s_1, a_1, \ldots, r_T, s_T)$ using current policy
    - For each $t \in \{0, \ldots, T-1\}$
      - Update policy:

$$\theta \leftarrow \theta + \eta \nabla_\theta \log \pi_\theta(a_t \mid s_t) \left(\sum_{k=0}^{T-t-1} r_{t+k+1} \right)$$

Actor-Critic Algorithms

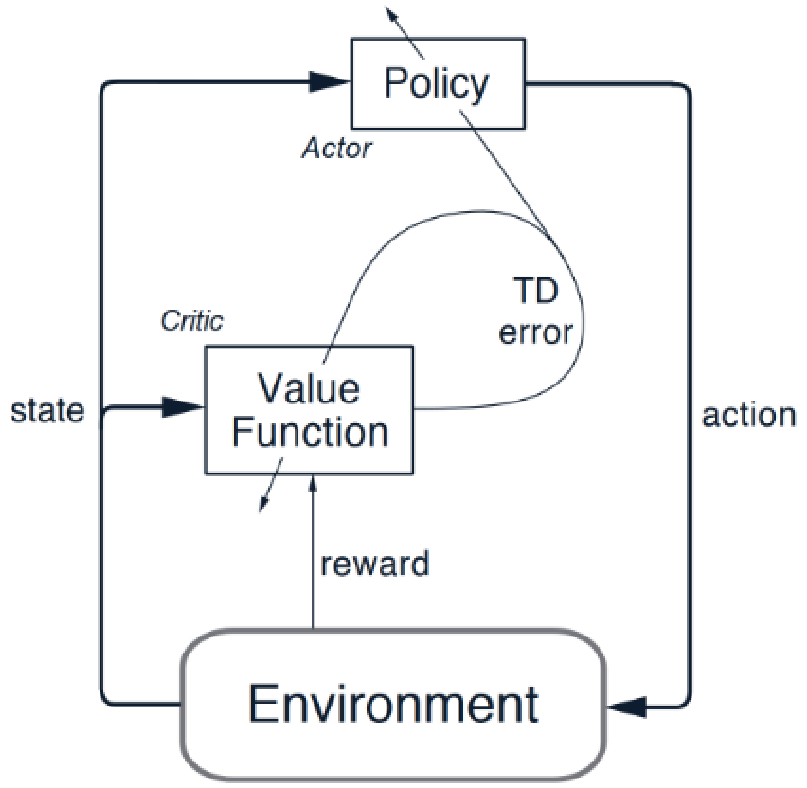
- Recap: REINFORCE plays full episodes to sample returns for policy improvement through policy gradient ascent

- Generalized Policy Iteration:
  - Critic: Q-function of the current policy learned on the fly (on-policy)

$$Q^{\pi}(s, a, w)$$

  - Actor: parametrized policy, improved based on its Q-function estimate

$$\nabla J_{\theta} = \nabla_{\theta} \log \pi_{\theta}(a \mid s)\, Q^{\pi}(s, a, w)$$



Advantage Actor-Critic Algorithm (A2C)

- Learning both the Value-function and the Policy together with 2 separate neural networks.
- Training to maximize the advantage $Adv(s,a) = Q(s,a) - V(s)$

– **Advantage**: use difference in return and value function estimate to update policy and value function, up to n-step expected return

n-step expected return: $Q_t = r_{t+1} + \ldots + \gamma^{n-1} r_{t+n} + \gamma^n V(s_{t+n}, v)$

policy gradient: $\nabla_\theta J(\theta) = \nabla_\theta \pi_\theta(a \mid s)\,(Q_t - V(s_t, v))$

value function objective: $L(v) = (Q_t - V(s_t, v))^2$

Intuition: updates make mismatch between predicted return and estimated return smaller (reduce wrong expectations)

$\Rightarrow$ *Effect: 4$\times$ mean Atari score vs. Nature DQN*

Deep Policy Gradients (DPG)

- DPG is the continuous analogue of DQN
  - Experience replay: build data-set from agent's experience
  - Critic estimates value of current policy by DQN

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

  - To deal with non-stationarity, targets $\mathbf{u}^-, \mathbf{w}^-$ are held fixed
  - Actor updates policy in direction that improves Q

$$\frac{\partial L_{\mathbf{u}}(\mathbf{u})}{\partial \mathbf{u}} = \frac{\partial Q(s, a, \mathbf{w})}{\partial a} \frac{\partial a}{\partial \mathbf{u}}$$

  - In other words critic provides loss function for actor.

Slide credit: David Silver

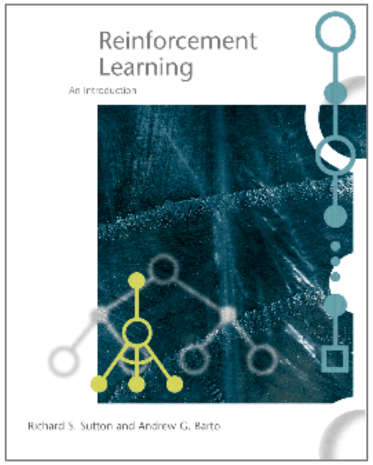
Summary

- The future looks bright!
  - Soon, you won’t have to play video games anymore…
  - Your computer can do it for you (and beat you at it)
- Reinforcement Learning is a very promising field
  - Currently limited by the need for data
  - At the moment, mainly restricted to simulation settings

References and Further Reading

- More information on Reinforcement Learning can be found in the following book

Richard S. Sutton, Andrew G. Barto
Reinforcement Learning: An Introduction
MIT Press, 1998



- The complete text is also freely available online

https://webdocs.cs.ualberta.ca/~sutton/book/ebook/the-book.html

References and Further Reading

- DQN paper
  - www.nature.com/articles/nature14236

- AlphaGo paper
  - www.nature.com/articles/nature16961

nature
At last — a computer program that can beat a champion Go player
ALL SYSTEMS GO