Charu C. Aggarwal
IBM T J Watson Research Center
Yorktown Heights, NY

Basic Principles of Reinforcement Learning [Motivating Deep Learning]

Neural Networks and Deep Learning, Springer, 2018
Chapter 8.1–8.3

The Complexity of Human Intelligence is Quite Simple!

- Herbert Simon's ant hypothesis:

  "*Human beings, viewed as behaving systems, are quite simple. The apparent complexity of our behavior over time is largely a reflection of the complexity of the environment in which we find ourselves.*"

  - Humans are simple, because they are reward-driven entities.
  - All of biological intelligence is owed to this simple fact.

- Reinforcement learning attempts to simplify the learning of complex behaviors by using *reward-driven trial and error*.

When to Use Reinforcement Learning?

- Systems that are simple to judge but hard to specify.

- Easy to use trial-and-error to generate data.
  - Video games (e.g., Atari)
  - Board and card games (e.g., chess, Go, Poker)
  - Robot locomotion and visuomotor skills
  - Self-driving cars

- *Reinforcement learning is the gateway to general forms of artificial intelligence!*

Why Don't We have General Forms of Artificial Intelligence Yet?

- Reinforcement learning requires large amounts of data (generated by trial and error).
  - Possible to generate lots of data in some game-centric settings, but not other real-life settings.
- Biological reinforcement learning settings include some unsupervised learning.
  - The number of synapses in our brain is larger than the number of seconds we live!
  - There must be some unsupervised learning going on continuously $\Rightarrow$ We haven't mastered that art yet.
- Recent results do show promise for the future.

Simplest Reinforcement Learning Setting: Multi-armed Bandits

- Imagine a gambler in a casino faced with 2 slot machines.

- Each trial costs the gambler \$1, but pays \$100 with some unknown (low) probability.

- The gambler suspects that one slot machine is better than the other.

- What would be the optimal strategy to play the slot machines, assuming that the gambler's suspicion is correct?

- **Stateless Model:** Environment at every time-stamp is identical (although knowledge of *agent* improves).

Observations

- Playing both slot machines alternately helps the gambler learn about their payoff (over time).
  - However, it is wasteful *exploration*!
  - Gambler wants to *exploit* winner as soon as possible.

- Trade-off between exploration and exploitation $\Rightarrow$ Hallmark of reinforcement learning

Naïve Algorithm

- **Exploration:** Play each slot machine for a fixed number of trials.

- **Exploitation:** Play the winner forever.
  - Might require a large number of trials to robustly estimate the winner.
  - If we use too few trials, we might actually play the poorer slot machine forever.

ϵ-Greedy Strategy

- Probabilistically merge exploration and exploitation.

- Play a random machine with probability ϵ, and play the machine with highest current payoff with probability $1 - \epsilon$.

- Main challenge in picking the proper value of $\epsilon \Rightarrow$ Decides trade-off between exploration and exploitation.

- **Annealing:** Start with large values of ϵ and reduce slowly.

Upper Bounding: The Optimistic Gambler!

- Upper-bounding represents optimism towards unseen machines $\Rightarrow$ Encourages exploration.

- Empirically estimate mean μ_i and standard deviation σ_i of payoff of the ith machine using its n_i trials.

- Pick the slot machine with largest value of mean plus confidence interval $= \mu_i + K \cdot \sigma_i / \sqrt{n_i}$
  - Note the $\sqrt{n_i}$ in the denominator, because it is *sample* standard deviation.
  - Rarely played slot machines more likely to be picked because of optimism.
  - Value of K decides trade-off between exploration and exploitation.

Multi-Armed Bandits versus Classical Reinforcement Learning

- Multi-armed bandits is the simplest form of reinforcement learning.

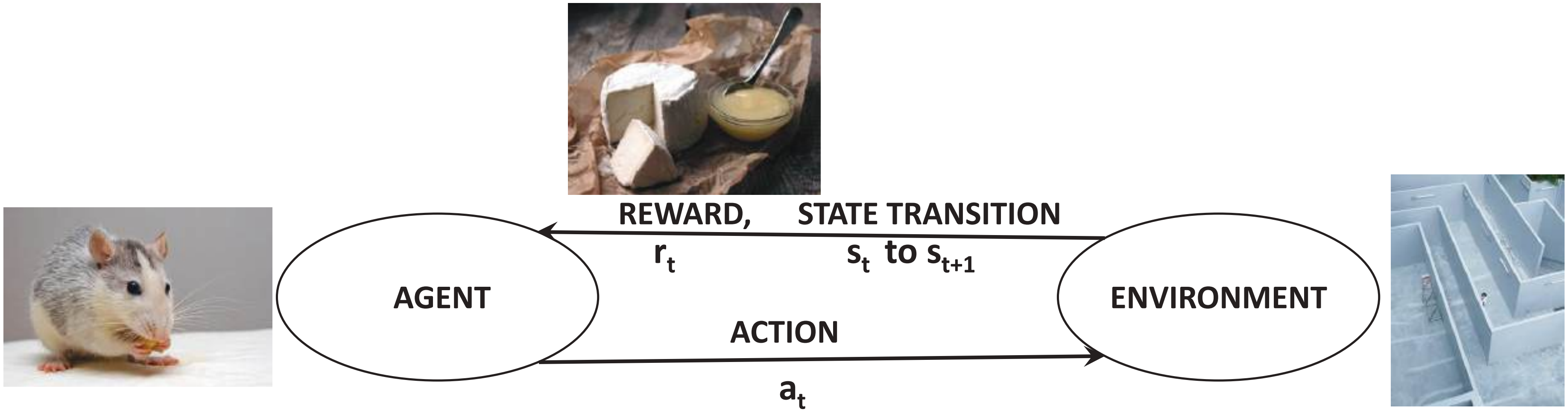
- The model is stateless, because the environment is identical at each time-stamp.

- Same action is optimal for each time-stamp.
  - Not true for classical reinforcement learning like Go, chess, robot locomotion, or video games.
  - *State* of the environment matters!

Markov Decision Process (MDP): Examples from Four Settings

- *Agent:* Mouse, chess player, gambler, robot
- *Environment:* maze, chess rules, slot machines, virtual test bed for robot
- *State:* Position in maze, chess board position, unchanged, robot joints
- *Actions:* Turn in maze, move in chess, pulling a slot machine, robot making step
- *Rewards:* cheese for mouse, winning chess game, payoff of slot machine, virtual robot reward

The Basic Framework of Reinforcement Learning



1. AGENT (MOUSE) TAKES AN ACTION a_t (LEFT TURN IN MAZE) FROM STATE (POSITION) s_t
2. ENVIRONMENT GIVES MOUSE REWARD r_t (CHEESE/NO CHEESE)
3. THE STATE OF AGENT IS CHANGED TO s_{t+1}
4. MOUSE'S NEURONS UPDATE SYNAPTIC WEIGHTS BASED ON WHETHER ACTION EARNED CHEESE

OVERALL: AGENT LEARNS OVER TIME TO TAKE STATE-SENSITIVE ACTIONS THAT EARN REWARDS

- The biological and AI frameworks are similar.

- MDP represented as $s_0 a_0 r_0 s_1 a_1 r_1 \ldots s_n a_n r_n$

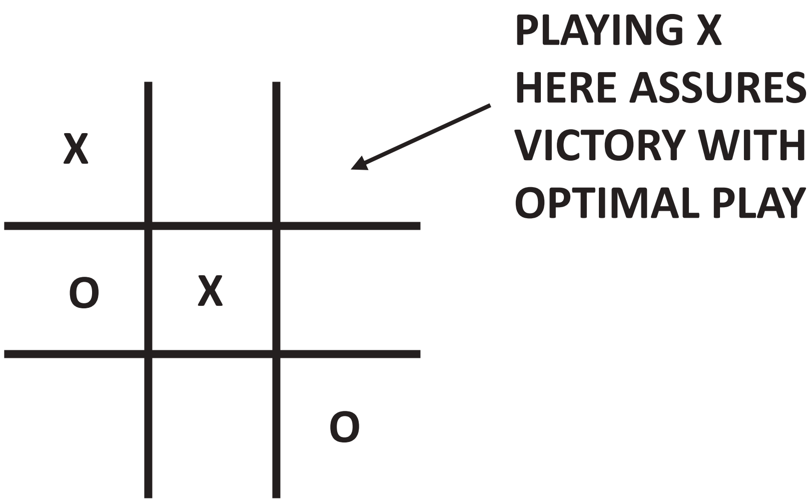
Examples of Markov Decision Process

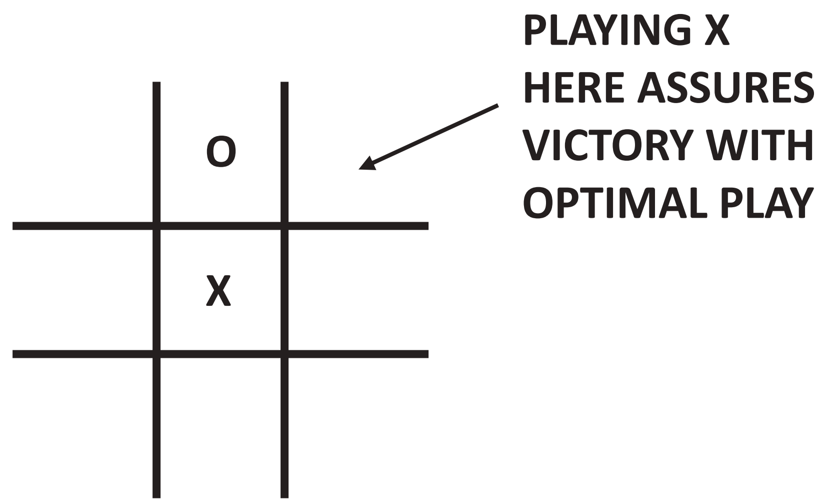
- *Game of tic-tac-toe, chess, or Go:* The state is the position of the board at any point, and the actions correspond to the moves made by the agent. The reward is $+1$, 0, or -1 (depending on win, draw, or loss), *which is received at the end of the game*.

- *Robot locomotion:* The state corresponds to the current configuration of robot joints and its position. The actions correspond to the torques applied to robot joints. The reward at each time stamp is a function of whether the robot stays upright and the amount of forward movement.

- *Self-driving car:* The states correspond to the sensor inputs from the car, and the actions correspond to the steering, acceleration, and braking choices. The reward is a function of car progress and safety.

Role of Traditional Reinforcement Learning





- **Traditional reinforcement learning:** Learn through trial-and-error the *long-term* value of each state.

- Long-term values are not the same as rewards.
  - Rewards not realized immediately because of stochasticity (e.g., slot machine) or delay (board game).

Reinforcement Learning for Tic-Tac-Toe

- Main difference from multi-armed bandits is that we need to learn the long-term rewards for each action in each *state*.

- An eventual victory earns a reward from $\{+1, 0, -1\}$ with delay.

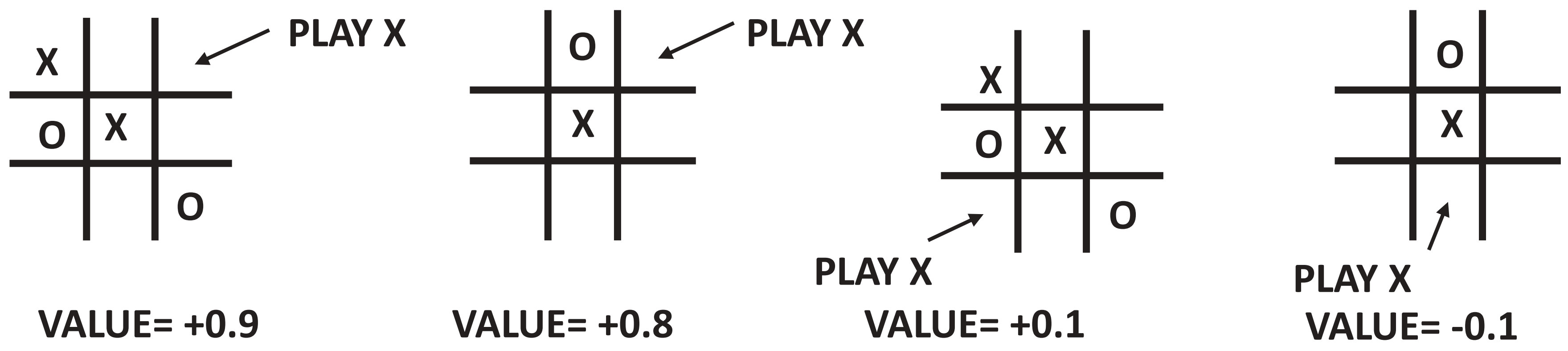
- A move occurring r moves earlier than the game termination earns *discounted* rewards of $\{\gamma^{r-1}, 0, -\gamma^{r-1}\}$.
  - Future rewards would be less certain in a replay.

- Assume that a fixed pool of humans is available as opponents to train the system (self-play possible).

Generalizing ϵ-Greedy to Tic-Tac-Toe

- Maintain table of values of state-action pairs (initialize to small random values).
  - In multi-armed bandits, we only had values on actions.
  - Table contains unnormalized total reward for each state-action pair $\Rightarrow$ Normalize to average reward.
- Use ϵ-greedy algorithm with *normalized* table values to simulate moves and create a game.
- **After game:** Increment at most 9 entries in the unnormalized table with values from $\{\gamma^{r-1}, 0, -\gamma^{r-1}\}$ for r moves to termination and win/loss.
- Repeat the steps above.

At the End of Training



- Typical examples of normalized values of moves
- ϵ-greedy will learn the strategic values of moves.
- Rather than state-action-value triplets, we can equivalently learn state-value pairs.

Where Does Deep Learning Fit In?

- The tic-tac-toe approach is a glorified "learning by rote" algorithm.

- Works only for toy settings with few states.
  - Number of board positions in chess is huge.
  - Need to be able to *generalize* to unseen states.
  - **Function Approximator:** Rather than a table of state-value pairs, we can have a *neural network* that maps states to values.
  - The *parameters* in the neural network substitute for the table.

Strawman ϵ-Greedy Algorithm with Deep Learning for Chess [Primitive: Don't Try It!]

- Convolutional neural network takes board position as input and produces position value as output.

- Use ϵ-greedy algorithm on output values to simulate a full game.

- **After game of X moves:** Create X training points with board position as input feature map and targets from $\{\gamma^{r-1}, 0, -\gamma^{r-1}\}$ depending on move number and win/loss.

- Update neural network with these X training points.

- Repeat the steps above.

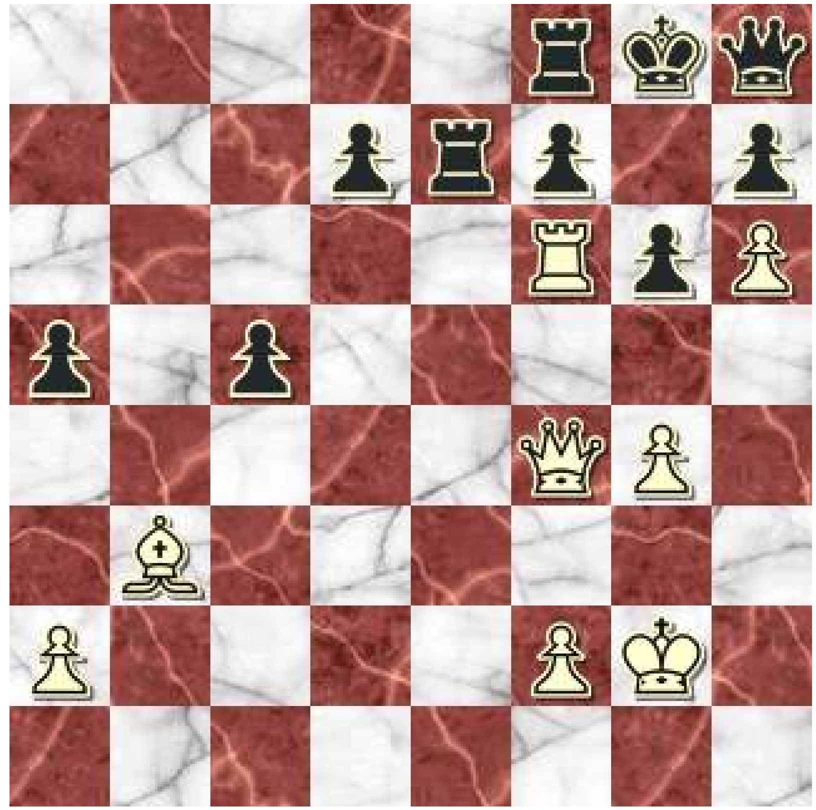
Reinforcement Learning in Chess and Go

- The reinforcement learning systems, *AlphaGo* and *Alpha Zero*, have been designed for chess, Go, and shogi.

- Combines various advanced deep learning methods and Monte Carlo tree search.

- Plays positionally and sometimes makes sacrifices (much like a human).
  - Neural network encodes evaluation function learned from trial and error.
  - More complex and subtle than hand-crafted evaluation functions by conventional chess software.

Examples of Two Positions from Alpha Zero Games vs Stockfish



- Generalize to unseen states in training.

- Deep learner can recognize subtle positional factors because of trial-and-error experience with feature engineering.

Other Challenges

- Chess and tic-tac-toe are *episodic*, with a maximum length to the game (9 for tic-tac-toe and $\approx$6000 for chess).

- The ϵ-greedy algorithm updates episode by episode.

- What about infinite Markov decision processes like robots or long episodes?
  - Rewards received continuously.
  - Not optimal to update episode-by-episode.

- Value function and Q-function learning can update after each *step* with *Bellman's equations*.

Charu C. Aggarwal
IBM T J Watson Research Center
Yorktown Heights, NY

Value Function Learning and Q-Learning

Neural Networks and Deep Learning, Springer, 2018
Chapter 8.4

Challenges with Long and Infinite Markov Decision Processes

- Previous lecture discusses how value functions can be learned for shorter episodes.
  - Update state-action-value table for each episode with Monte Carlo simulation.

- Effective for games like tic-tac-toe with small episodes.

- What to do with continuous Markov decision processes?

An Infinite Markov Decision Process

- Sequence below is of infinite length (continuous process)

$$s_0 a_0 r_0 s_1 a_1 r_1 \dots s_t a_t r_t \dots$$

- The cumulative reward R_t at time t is given by the discounted sum of the immediate rewards for $\gamma \in (0, 1)$:

$$R_t = r_t + \gamma \cdot r_{t+1} + \gamma^2 \cdot r_{t+2} + \gamma^3 \cdot r_{t+3} \dots = \sum_{i=0}^{\infty} \gamma^i r_{t+i} \quad (1)$$

- Future rewards worth less than immediate rewards ($\gamma < 1$).

- Choosing $\gamma < 1$ is not essential for episodic processes but critical for long MDPs.

Recap of Episodic ϵ-Greedy for Tic-Tac-Toe

- Maintain table of average values of state-action pairs (initialize to small random values).

- Use ϵ-greedy algorithm with table values to simulate moves and create a game.

- **After game:** Update at most 9 entries in the table with new averages, based on the outcomes from $\{\gamma^{r-1}, 0, -\gamma^{r-1}\}$ depending on move number and win/loss.

- Repeat the steps above.

The Bootstrapping Intuition

- Consider a Markov decision process in which we are predicting values (e.g., long-term rewards) at each time-stamp.

  – A partial simulation of the future can improve the prediction at the current time-stamp.

  – This improved prediction can be used as the ground-truth at the current time stamp.

- **Tic-tac-toe:** Parameterized evaluation function for board.

  – After our opponent plays the next move, and board evaluation changes unexpectedly, we go back and correct parameters.

- **Temporal difference learning:** Use difference in prediction caused by partial lookaheads (treated as error for updates).

Example of Chess

- Why is the minimax evaluation of a chess program at 10-ply stronger than that using the 1-ply board evaluation?
  - Because evaluation functions are imperfect (can be strengthened by "cheating" with data from future)!
  - If chess were solved (like checkers today), the evaluation function at any ply would be the same.
  - The minimax evaluation at 10 ply can be used as a "ground truth" for updating a parameterized evaluation function at current position!
- Samuel's checkers program was the pioneer (called *TD-Leaf* today)
- Variant of idea used by TD-Gammon, *Alpha Zero*.

Q-Learning

- Instead of minimax over a tree, we can use one-step lookahead

- Let $Q(s_t, a_t)$ be a table containing optimal values of state-action pairs (best value of action a_t in state s_t).

- Assume we play tic-tac-toe with ϵ-greedy and $Q(s_t, a_t)$ initialized to random values.

- Instead of Monte Carlo, make following update:

$$Q(s_t, a_t) = r_t + \gamma \mathrm{max}_a Q(s_{t+1}, a) \tag{2}$$

- Update: $Q(s_t, a_t) = Q(s_t, a_t)(1 - \alpha) + \alpha(r_t + \gamma \mathrm{max}_a Q(s_{t+1}, a))$

Why Does this Work?

- Most of the updates we initially make are not meaningful in tic-tac-toe.
  - We started off with random values.
- However, the update of the value of a next-to-terminal state is informative.
- The next time the next-to-terminal state occurs on RHS of Bellman, the update of the next-to-penultimate state will be informative.
- Over time, we will converge to the proper values of all state-action pairs.

SARSA: ϵ-greedy Evaluation

- Let $Q(s_t, a_t)$ be the value of action a_t in state s_t when following the ϵ-greedy policy.

- **An improved estimate** of $Q(s_t, a_t)$ via bootstrapping is $r_t + \gamma Q(s_{t+1}, a_{t+1})$

- Follows from $R_t = \sum_{i=0}^{\infty} \gamma^i r_{t+i} = r_t + \gamma R_{t+1}$

- SARSA: Instead of episodic update, we can update the table containing $Q(s_t, a_t)$ after performing a_t by ϵ-greedy, observing s_{t+1} and then computing a_{t+1} again using ϵ-greedy:
$$Q(s_t, a_t) \Leftarrow r_t + \gamma Q(s_{t+1}, a_{t+1}) \quad (3)$$

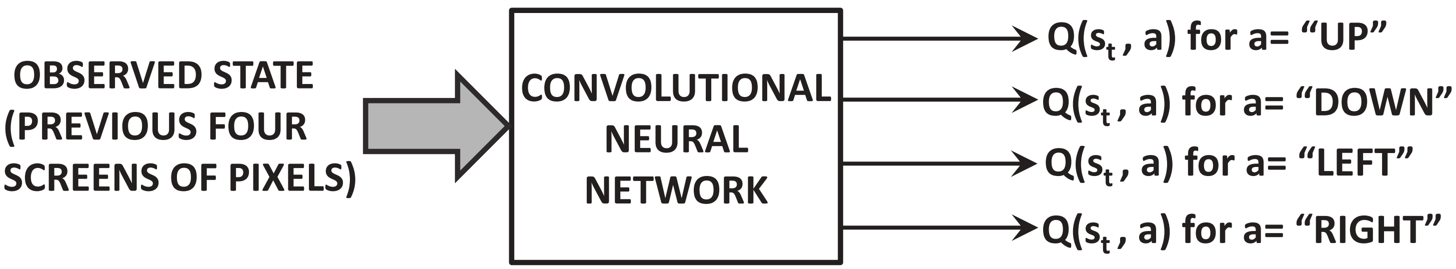
- Gentler and stable variation: $Q(s_t, a_t) \Leftarrow Q(s_t, a_t)(1 - \alpha) + \alpha(r_t + \gamma Q(s_{t+1}, a_{t+1}))$

On-Policy vs Off-Policy Learning

- SARSA: On-policy learning is useful when learning and inference cannot be separated.
  - A robot who continuously learns from the environment.
  - The robot must be cognizant that exploratory actions have a cost (e.g., walking at edge of cliff).

- Q-learning: Off-policy learning is useful when we don't need to perform exploratory component during inference time (have non-zero ϵ during training but set to 0 during inference).
  - Tic-tac-toe can be learned once using Q-learning, and then the model is fixed.

Using Deep Learning

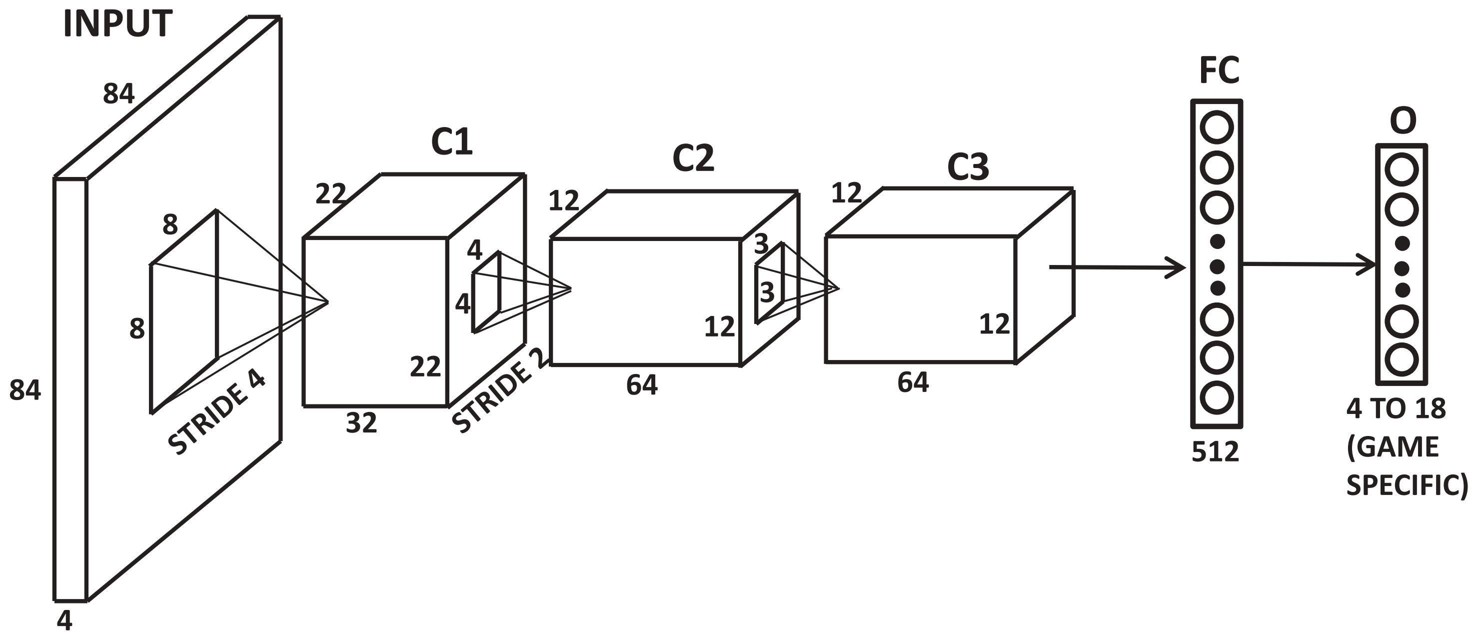


- When the number of states is large, the values $Q(s_t, a_t)$ are predicted from state s_t representation $\overline{X}_t$ rather than tabulated.

$$F(\overline{X}_t, \overline{W}, a) = \hat{Q}(s_t, a) \tag{4}$$

- $\overline{X}_t$: Previous four screens of pixels in Atari

Specific Details of Convolutional Network



- Same architecture with minor variations was used for all Atari games.

Neural Network Updates for Q-Learning

- The neural network outputs $F(\overline{X}_t, \overline{W}, a_t)$.

- We must wait to observe state $\overline{X}_{t+1}$ and then set up a "ground-truth" value for the output using Bellman's equations:

$$\text{Bootstrapped Ground-Truth} = r_t + \gamma \text{max}_a F(\overline{X}_{t+1}, \overline{W}, a) \tag{5}$$

- Loss: $L_t = \left\{ \underbrace{[r_t + \gamma \text{max}_a F(\overline{X}_{t+1}, \overline{W}, a)]}_{\text{Treat as constant ground-truth}} - F(\overline{X}_t, \overline{W}, a_t) \right\}^2$

$$\overline{W} \Leftarrow \overline{W} + \alpha \left\{ \underbrace{[r_t + \gamma \text{max}_a F(\overline{X}_{t+1}, \overline{W}, a)]}_{\text{Constant ground-truth}} - F(\overline{X}_t, \overline{W}, a_t) \right\} \frac{\partial F(\overline{X}_t, \overline{W}, a_t)}{\partial \overline{W}} \tag{6}$$

Neural Network Updates for SARSA

- The neural network outputs $F(\overline{X}_t, \overline{W}, a_t)$.

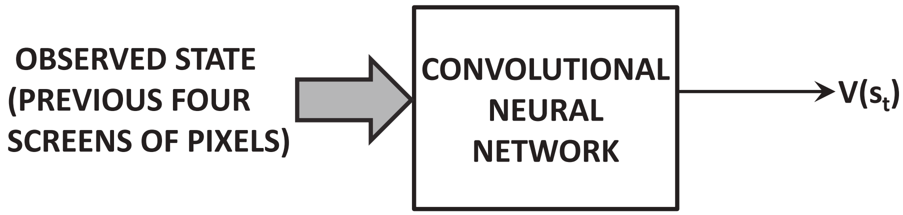
- We must wait to observe state $\overline{X}_{t+1}$, simulate a_{t+1} with ϵ-greedy and then set up a "ground-truth" value:

$$\text{Bootstrapped Ground-Truth} = r_t + \gamma F(\overline{X}_{t+1}, \overline{W}, a_{t+1}) \quad (7)$$

- Loss: $L_t = \left\{ \underbrace{[r_t + \gamma F(\overline{X}_{t+1}, \overline{W}, a_{t+1})]}_{\text{Treat as constant ground-truth}} - F(\overline{X}_t, \overline{W}, a_t) \right\}^2$

$$\overline{W} \Leftarrow \overline{W} + \alpha \left\{ \underbrace{[r_t + \gamma F(\overline{X}_{t+1}, \overline{W}, a_{t+1})]}_{\text{Constant ground-truth}} - F(\overline{X}_t, \overline{W}, a_t) \right\} \frac{\partial F(\overline{X}_t, \overline{W}, a_t)}{\partial \overline{W}} \quad (8)$$

Value Function Learning



- Instead of outputting values of state-action pairs we can output just values.

- Q-Learning and SARSA can be implemented with this architecture as well.
  - General class of temporal difference learning $\Rightarrow$ Multi-step bootstrapping
  - Can explore a forward-looking tree for arbitrary bootstrapping.

Temporal Difference Learning $TD(0)$

- Value network produces $G(\overline{X}_t, \overline{W})$ and bootstrapped ground truth $= r_t + \gamma G(\overline{X}_{t+1}, \overline{W})$

- Same as SARSA: Observe next state by executing a_t according to current policy

- Loss: $L_t = \left\{ \underbrace{r_t + \gamma G(\overline{X}_{t+1}, \overline{W})}_{\text{“Observed” value}} - G(\overline{X}_t, \overline{W}) \right\}^2$

$$\overline{W} = \overline{W} + \alpha \left\{ \underbrace{[r_t + \gamma G(\overline{X}_{t+1}, \overline{W})]}_{\text{“Observed” value}} - G(\overline{X}_t, \overline{W}) \right\} \frac{\partial G(\overline{X}_t, \overline{W})}{\partial \overline{W}} \tag{9}$$

- Short notation: $\overline{W} \Leftarrow \overline{W} + \alpha \delta_t (\nabla G(\overline{X}_t, \overline{W}))$

Bootstrapping over Multiple Steps

- Temporal difference bootstraps only over one time-step.
  - A strategically wrong move will not show up immediately.
  - Can look at n-steps instead of one.
- On-policy looks at single sequence greedily (too weak)
- Off-policy (like Bellman) picks optimal over entire minimax tree (Samuel's checkers program).
- Any optimization heuristic for lookahead-based inference can be exploited.
  - Monte Carlo tree search explores *multiple branches* with upper-bounding strategy $\Rightarrow$ Statistically robust target.

Fixed Window vs Smooth Decay: Temporal Difference Learning $TD(\lambda)$

- Refer to one-step temporal difference learning as $TD(0)$

- Fixed Window n: $\overline{W} \Leftarrow \overline{W} + \alpha\delta_t \sum_{k=t-n+1}^{t} \gamma^{t-k}(\nabla G(\overline{X}_k, \overline{W}))$

- $TD(\lambda)$ corrects past mistakes with discount factor λ when new information is received.

$$\overline{W} \Leftarrow \overline{W} + \alpha\delta_t \sum_{k=0}^{t} (\lambda\gamma)^{t-k}(\nabla G(\overline{X}_k, \overline{W})) \tag{10}$$

- Setting $\lambda = 1$ or $n = \infty$ is equivalent to Monte Carlo methods.
  - Details in book.

Monte Carlo vs Temporal Differences

- Not true that greater lookahead always helps!
  - The value of λ in $TD(\lambda)$ regulates the trade-off between bias and variance.
  - Using small values of λ is particularly advisable if data is limited.
- A temporal difference method places a different value on each position in a single chess game (that depends on the merits of the position).
  - Monte Carlo places a value that depends only on time discounting and final outcome.

Monte Carlo vs Temporal Differences: Chess Example

- Imagine a Monte Carlo rollout of chess game between two agents Alice and Bob.
  - Alice and Bob each made two mistakes but Alice won.
  - Monte Carlo training data does not differentiate between mistakes and good moves.
  - Using temporal differences might see an error after each mistake because an additional ply has *differential* insight about the effect of the move (bootstrapping).
- More data is needed in Monte Carlo rollouts to remove the effect of noise.
- On the other hand, TD(0) might favor learning of end games over openings.

Implications for Other Methods

- Policy gradients often use Monte Carlo rollouts.
  - Deciding the *advantage* of an action is often difficult in games without continuous rewards.

- Value networks are often used in combination with *policy-gradients* in order to design *actor-critic* methods.
  - Temporal differences are used to evaluate the advantage of an action.

Charu C. Aggarwal
IBM T J Watson Research Center
Yorktown Heights, NY

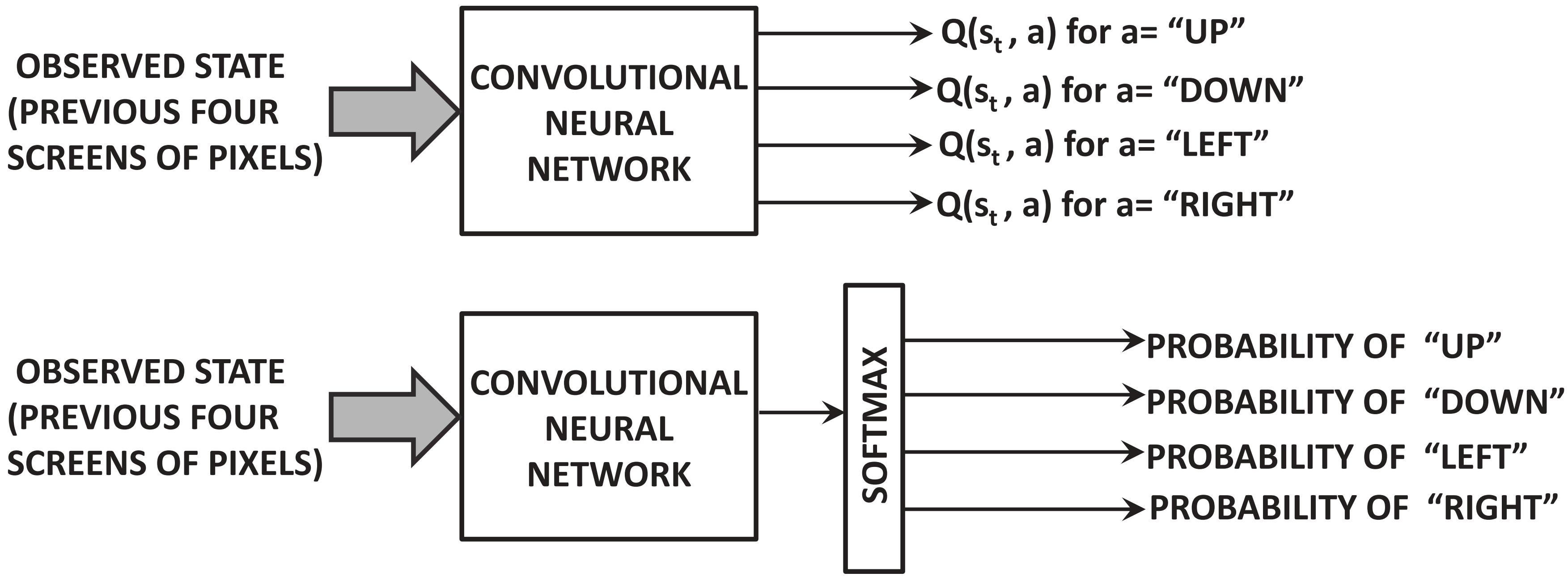
Policy Gradients

Neural Networks and Deep Learning, Springer, 2018
Chapter 8.5

Difference from Value-Based Methods

- Value-based methods like Q-learning attempt to predict the value of an action with a *parameterized value function*.
  - Often coupled with a generic policy (like ϵ-greedy).
- Policy gradient methods estimate the *probability* of each action at each step with the goal of maximizing the overall reward.
- Policy is itself parameterized.

Policy Network vs Q-Network for Atari Game



- Output is *probability* of each action in policy network rather than *value* of each action.

Overview of Approach

- We want to update network to maximize *expected* future rewards $\Rightarrow$ We need to collect samples of long-term rewards for each simulated action.

  - **Method 1:** Use Monte Carlo policy rollouts to estimate the *simulated* long-term reward after each action.

  - **Method 2:** Use another value network to *model* long-term reward after each action (actor-critic methods).

- Main problem is in setting up a loss function that uses the simulated or modeled rewards to update the parameterized probabilities.

Example: Generating Training Data

- Training chess agent Alice (using pool of human opponents) with reward in $\{+1, 0, -1\}$.

- Consider a Monte Carlo simulation with win for agent Alice.

- Create training points for each board position faced by Alice and each action output a with long-term reward of 1.
  - If discount factor of γ then long-term reward is γ^{r-1}.

- **Backpropagated stochastic gradient ascent:** Somehow need to update neural network from samples of rewards to maximize expected rewards (non-obvious).

Nature of Training Data

- We have board positions together with output action *samples* and long-term rewards of each *sampled* action.
  - We do not have ground-truth *probabilities*.

- So we want to maximize *expected* long-term rewards from *samples* of the probabilistic output.

- How does one compute the gradient of an expectation from samples?

Log Probability Trick

- Let $Q^p(s_t, a)$ be the long-term reward of action a and policy p.

- The log probability trick of REINFORCE relates gradient of expectation to expectation of gradient:

$$\nabla E[Q^p(s_t, a)] = E[Q^p(s_t, a) \nabla \log(p(a))] \tag{11}$$

- $Q^p(s_t, a)$ is estimated by Monte-Carlo roll-out and $\nabla \log(p(a))$ is the log-likelihood gradient from backpropagation in the policy network for sampled action a.

$$\overline{W} \Leftarrow \overline{W} + \alpha Q^p(s_t, a) \nabla \log(p(a)) \tag{12}$$

Baseline Adjustments

- Baseline adjustments change the reward into an "advantage" with the use of state-specific adjustments.
  - Subtract some quantity b from each $Q(s_t, a)$
  - Reduces variance of result without affecting bias.

- **State-independent:** One can choose b to be some long-term average of $Q^p(s_t, a)$.

- **State-specific:** One can choose b to be the value $V^p(s_t)$ of state $s_t \Rightarrow$ Advantage is same as temporal difference!

Why Does a State-Specific Baseline Adjustment Make Sense?

- Imagine a self-play chess game in which both sides make mistakes but one side wins.
  - Without baseline adjustments all training points from the game will have long-term rewards that depend only on final results.
  - Temporal difference will capture the *differential* impact of the error made in each action.
  - Gives more refined idea of the specific effect of that move $\Rightarrow$ Its advantage!

Problems with Monte Carlo Policy Gradients

- Full Monte Carlo simulation is best for episodic processes.

- Actor-critic methods allow online updating by combining ideas in policy gradients and value networks:

  - **Value-based:** The policy (e.g., ϵ-greedy) of the actor is subservient to the critic.

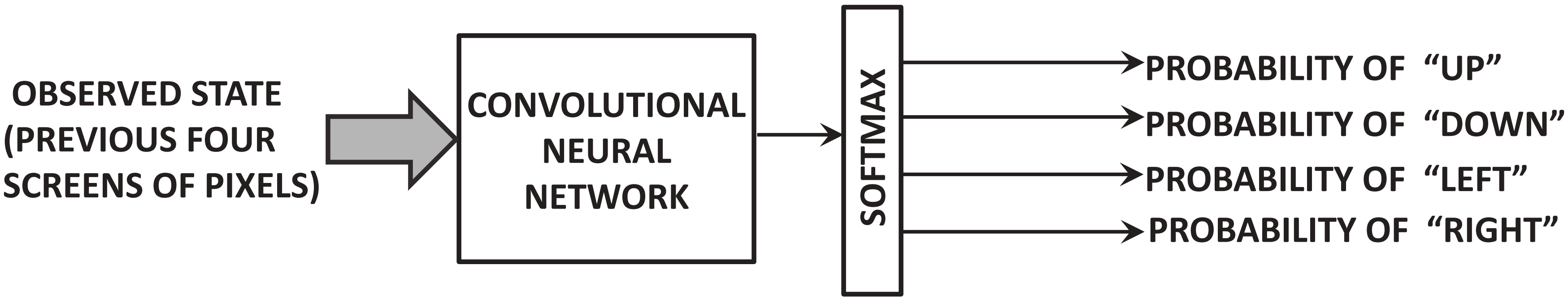
  - **Policy-based:** No notion of critic for value estimation (typical approach is Monte Carlo)

- **Solution:** Create separate neural network for estimating value/advantage.

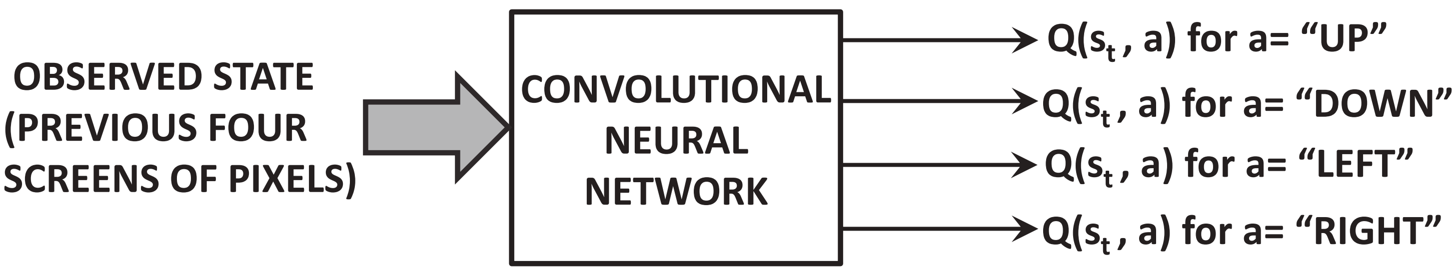
Actor-Critic Method

- Two separate neural networks:
  - **Actor:** Policy network with parameters $\overline{\Theta}$ that decides actions.
  - **Critic:** Value network or Q-network with parameter $\overline{W}$ that estimates long-term reward/advantage $\Rightarrow$ Advantage is temporal difference.
- The networks are trained simultaneously within an iterative loop.

Actor and Critic



(a) Actor (Decides actions as a probabilistic policy)



(b) Critic (Evaluates advantage in terms of temporal differences)

Steps in Actor-Critic Methods

- Sample the action a_{t+1} at state s_{t+1} using the policy network.

- Use Q-network to compute temporal difference error δ_t at s_t using bootstrapped target derived from value of s_{t+1}.

- **[Update policy network parameters]:** Update policy network using the Q-value of action a_t as its advantage (use temporal difference error for variance reduction).

- **[Update Q-Network parameters]:** Update the Q-network parameters using the squared temporal difference δ_t^2 as the error.

- Repeat the above updates.

Advantages and Disadvantages of Policy Gradients

- Advantages:
  - Work in continuous action spaces.
  - Can be used with stochastic policies.
  - Stable convergence behavior
- Main disadvantage is that they can reach local optima.