Reinforcement Learning in Digital Finance

Policy-based reinforcement learning

Funded by
the European Union

Lecture agenda

- Policy gradient algorithms
  - Introduction to policy function approximation
  - Examples
  - Derivation of theory
  - Discrete and continuous policies
  - Actor-critic models
  - Advanced policy gradients

Policy gradient methods

Value function approximation [1/2]

- So far, we stayed close to Dynamic Programming paradigm:
  - Replace true value functions V with approximation $\bar{V}$
  - Several ways to find suitable $\bar{V}$ (e.g., Q-table, features)
  - Finding optimal value functions equates finding optimal policy

- Disadvantages of VFA:
  - Falls apart for continuous- and large action spaces
    - Must evaluate $\bar{V}(s, a)$ for every action a in state s.
  - Indirect and unnatural way of decision-making
    - Dynamic Programming not intuitive for everyone

DIGITAL

Value function approximation [2/2]

- We already abandoned optimality, no need to stick to Dynamic Programming approach
- Objective is <u>not</u> to solve Bellman equation, but to maximize reward over certain time horizon!

- Alternative: Directly adjust decision-making policy
  - Often more natural
  - Value functions are just a *means* to improve policy
- Recall: policy simply maps state to action!

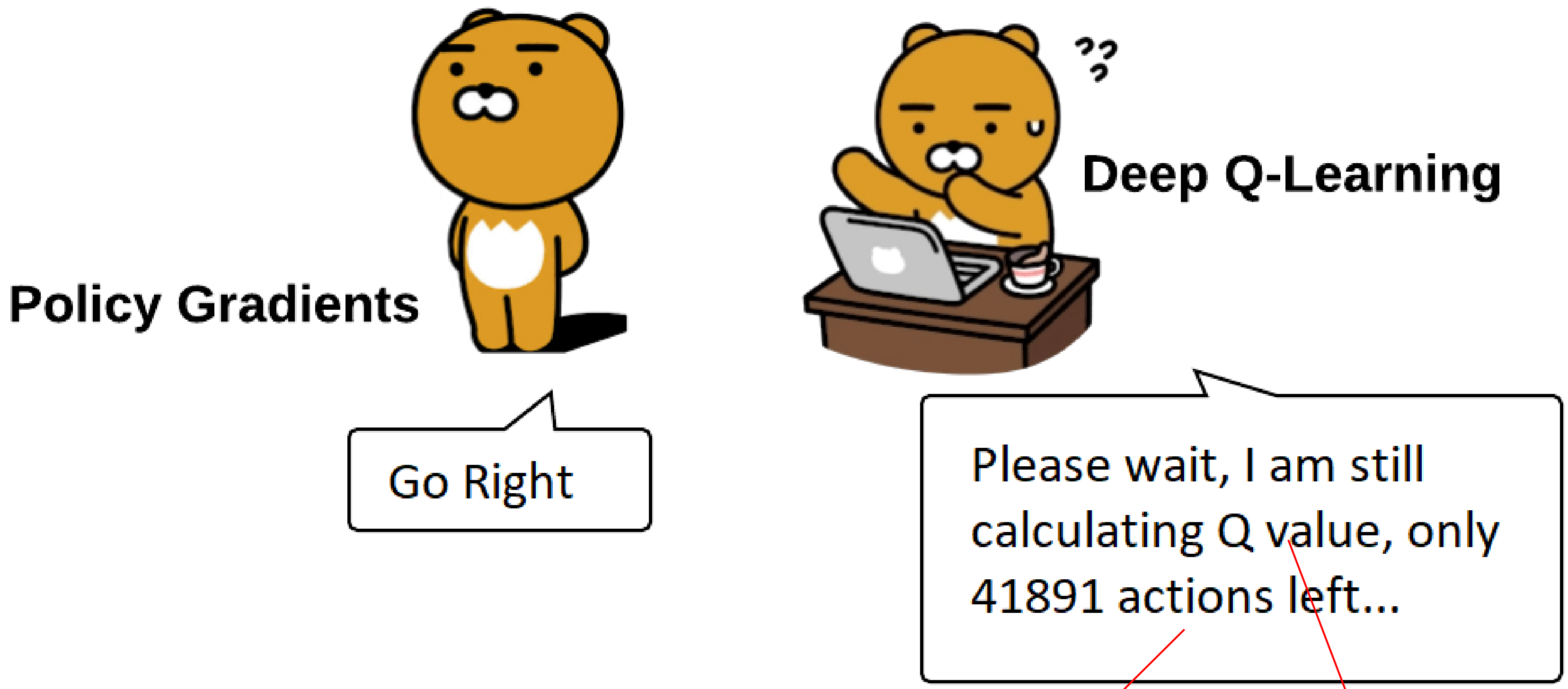
$$\pi : s \rightarrow a$$

DIGITAL

Policy function approximation – PFA vs VFA

Source: https://pylessons.com/Beyond-DQN

Note: discrete policy gradients still require full enumeration of action probabilities!



$$a = \pi_\theta(s)$$

$$a = \operatorname*{argmax}_{\boxed{a\in\mathcal{A}}}[r(s,a) + Q(s,a)]$$

Policy function approximation

- How do we improve a policy?

DIGITAL

Policy function approximation

- How do we improve a policy?

- Basic mechanism:
  - Define policy π_θ with tunable parameters θ
  - Take actions according to policy
  - Observe corresponding rewards
    - Typically reward trajectory $r(\tau) = \sum_{t=0}^{T} r_t$
  - Adjust policy π_θ (i.e., adjust parameters θ)
  - Observe whether rewards improve
  - Repeat

DIGITAL

Policy function approximation

- But: how do we know in what direction to update policy?
  - Sell higher/lower? Keep less/more inventory?

- A possible solution is to work with stochastic policies.
  - Allows measuring the *difference* between actions
  - So far, we used policies $\pi: s \rightarrow a$ (deterministic)
  - Now, we will use policies $\pi: s \rightarrow \mathbb{P}(a|s)$ (stochastic)

- We have two sources of information: (i) reward trajectory and (ii) probability of trajectory
  - Intuition: increase probability of high-reward trajectories

DIGITAL

Policy function approximation

- We adjust the tunable policy π_θ based on observed reward- and probability trajectories.
  - Mathematically speaking, we compute the gradient
    - Gradient is simply a vector of partial derivatives for each θ

- We can express the gradient as an *expectation*, thus we can use simulation (*sampling*) to approximate it

- PFA *may* tackle both large state- and action spaces
  - However, there are drawbacks as well

DIGITAL

Policy function approximation

- Gradient method not the only way to tailor policy
- Non-gradient solutions:
  - Genetic algorithms
  - SIMPLEX
  - Hill climbing
- Gradient methods often more efficient
  - Stochastic gradient descent
  - Newton's method
- This course focuses on policy gradient methods.

DIGITAL

Stochastic policies – Probabilistic actions

- So far, we worked with *deterministic* policies
  - Return single action for a given state $\pi: s \rightarrow a$
- Now, the policy is a probability function:
  - Discrete action space: assign probability to each action
  - Continuous action space: draw action from distribution

- Policy gradient:
  - Probability function (policy) should be differentiable
  - Tune differentiable parameters θ
    - Linear scheme: $\phi(s, a) \cdot \theta$

DIGITAL

Stochastic policies – General form

- General form of stochastic policy
  - Conditional probability function: $\pi(a|s) = \mathbb{P}(a|s)$
  - Map state to action probabilities: $s \rightarrow \mathbb{P}(a|s)$

- Examples:
  - Softmax: $\pi_\theta = e^{\phi(s,a)\theta}$
  - Gaussian: $\pi_\theta = \frac{1}{\sqrt{2\pi\sigma}} e^{-\frac{(a-\mu)^2}{2\sigma^2}}$
  - Neural network: $N_\theta : s \rightarrow \mathbb{P}(a|s)$

DIGITAL

Stochastic policies - Benefits

- Benefits compared to value-based learning (Q-learning)
  - Theoretically smooth updates, just follow gradient
    - Should lead to at least a local optimum
  - Often effective in (relatively) high-dimensional and continuous action spaces (latter yields infinite value functions)
  - Exploration mechanism already built in
    - Much exploration under high uncertainty, limited exploration once good policy has been found
  - Sometimes necessary (e.g., rock, paper, scissors)
    - Especially in multi-agent settings, opponents may counter deterministic policies.

DIGITAL

Stochastic policies - Drawbacks

- Disadvantages of stochastic policies
  - Discrete action space should be enumerable
    - Not always realistic for combinatorial optimization
  - High variance in reward trajectories
    - Requires full reward trajectory (like in Monte Carlo)
    - No bootstrapping as in SARSA or Q-learning

DIGITAL

Policy gradient – An example

Cliff walking example – PFA

- Four actions at each tile (up, down, left, right)
  - Softmax policy attaches probability to each action
  - Note we do not explicitly include downstream effects!

$$\pi_\theta(s,a) \propto e^{\phi(s,a)^\top \theta}$$

$$\pi_\theta(s,a) = \frac{e^{\phi(s,a)^\top \theta}}{\sum_{a' \in \mathcal{A}} e^{\phi(s,a')^\top \theta}}$$

DIGITAL

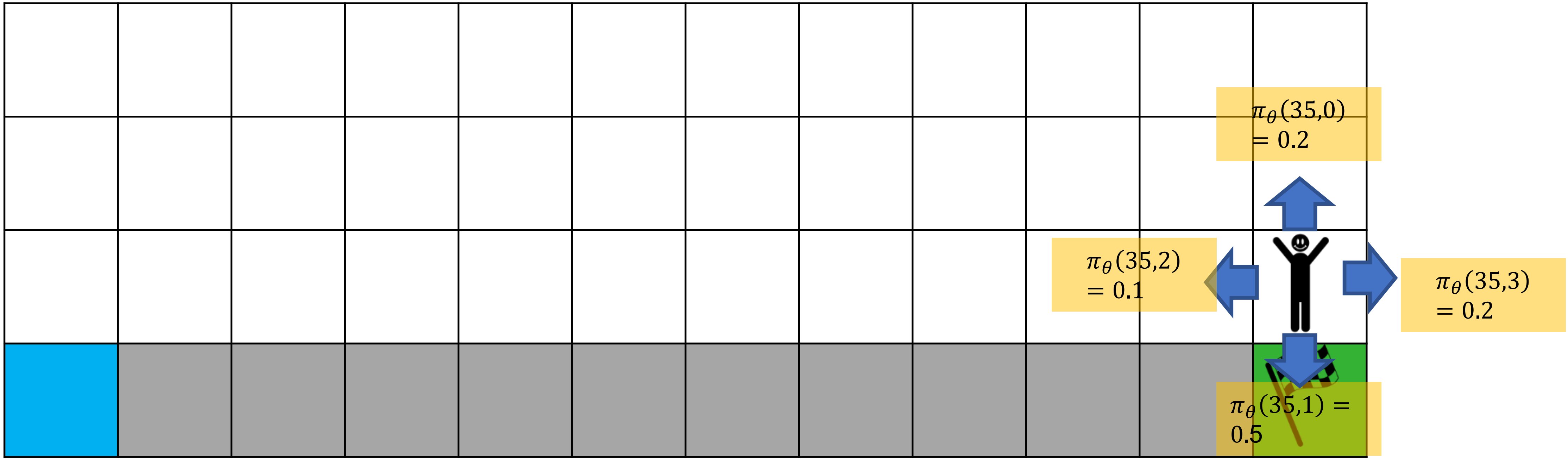
Cliff walking example – Feature design

- Feature vector: one-hot encoding of state-action pair
  - Feature vector of length $48 \cdot 4$, format $\phi(s, a) = [0,0 \ldots, 0,1, \ldots, 0,0]$
  - In this case, requires feature weights θ for each state-action pair state (vector $\theta = \mathbb{R}^{|48 \cdot 4|}$)
  - For sake of illustration, features are similar to lookup table
    - Approach can be generalized to more abstract features

  - In this example, a four-dimensional vector is used for conciseness!

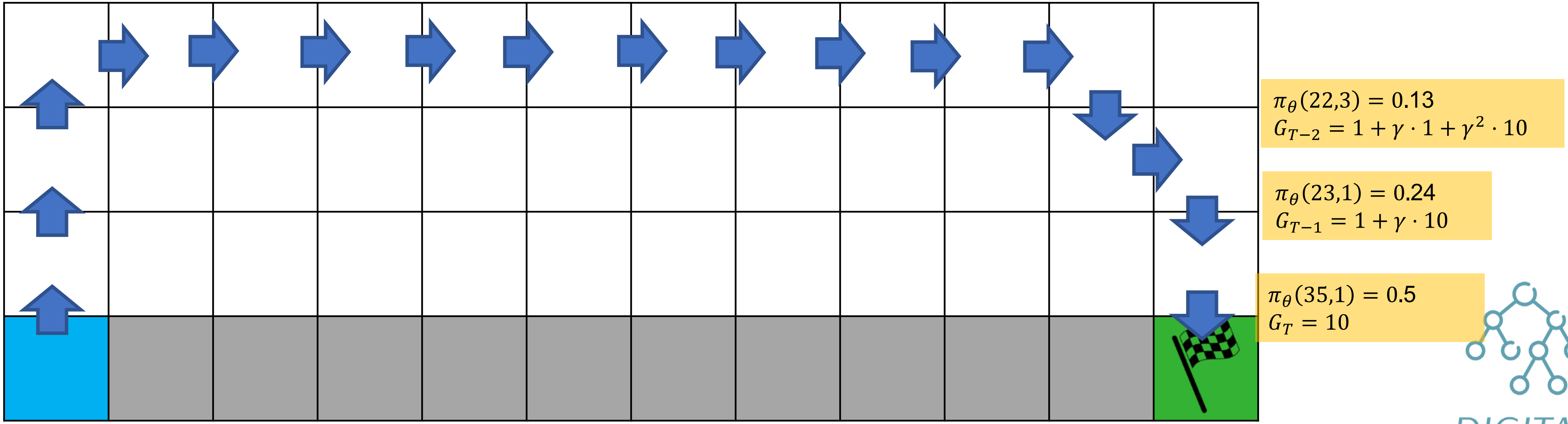
Cliff walking problem – Decision-making [1/2]

- Select action according to probability π_θ
  - In a sense, we are always exploring!



Cliff walking problem – Decision-making [2/2]

- Suppose we selected action 1 ('down')
- We can now observe full reward trajectory $R(\tau)$
  - Derive cumulative rewards $G_t = r_t + \gamma r_{t+1} \dots + \gamma^{T-t} r_T$



Cliff walking example – Initialization

- Initialize probabilities for each action to 0.25
  - Simply achieved by setting ($\theta_{s,a} = 0, \forall s, a$)
- In this example:
  - Attach weight to each state-action pair
  - Comparable to Q-table (learn parameter for each (s, a))

$$\theta = \begin{bmatrix} \theta_{0,0} & \cdots & \theta_{0,|\mathcal{A}|} \\ \vdots & \ddots & \vdots \\ \theta_{|S|,0} & \cdots & \theta_{|S|,|\mathcal{A}|} \end{bmatrix}$$

DIGITAL

Cliff walking example – Learning procedure

- Select action according to π_θ
- Run trajectory $\tau = s_0, a_0, \ldots s_T, a_t$
  - Store action probabilities and rewards of full trajectory
- After completion, loop over all time steps
  - Compute G_t
  - Compute $\phi(s,a)$ and probability-weighted vector $\sum_{a' \in \mathcal{A}} \phi(s,a)$
  - Compute score function (gradient of softmax)
    - Partial derivative for each feature
    - $\nabla_\theta \log_\theta(s,a) = \phi(s,a) - \mathbb{E}_{\pi_\theta}[\phi(s,\cdot)]$
- Update weights
  - $\theta = \theta + \alpha \nabla_\theta \log_\theta(s,a)$

DIGITAL

Cliff walking example

- Feature vector (selected action):

$$\phi(s,a) = [0,1,0,0]$$

(keep in mind, original is $48 \cdot 4!$)

- Expected feature vector (weighted over all actions):

$$\mathbb{E}_{\pi_\theta}\phi(s,\cdot) = \sum_{a' \in \mathcal{A}} \phi(s,a)$$

$$= [1,0,0,0] \cdot 0.2 + [0,1,0,0] \cdot 0.5 + [0,0,1,0] \cdot 0.1 + [0,0,0,1] \cdot 0.1$$
$$= [0.2, 0.5, 0.1, 0.2]$$

- Note we weight all possible feature vectors given s

DIGITAL

Cliff walking example – Weight update

- Score function:
$$\phi(s,a) - \mathbb{E}_{\pi_\theta}[\phi(s,\cdot)] = [0,1,0,0] - [0.2, 0.5, 0.1,0]\,.$$

- Weight update:
$$\Delta\theta = \alpha * \nabla_\theta \log_\theta(s,a) * G_t = 0.01 \cdot [-0.2,0.5,-0.1,-0.2] \cdot 10$$
$$= [-0.02, \mathbf{0.05}, -0.01, -0.02]$$

- Result: update increases weight of second action (due to positive reward)
  - In the future, we select this action with higher probability
  - High reward/low probability → strong update

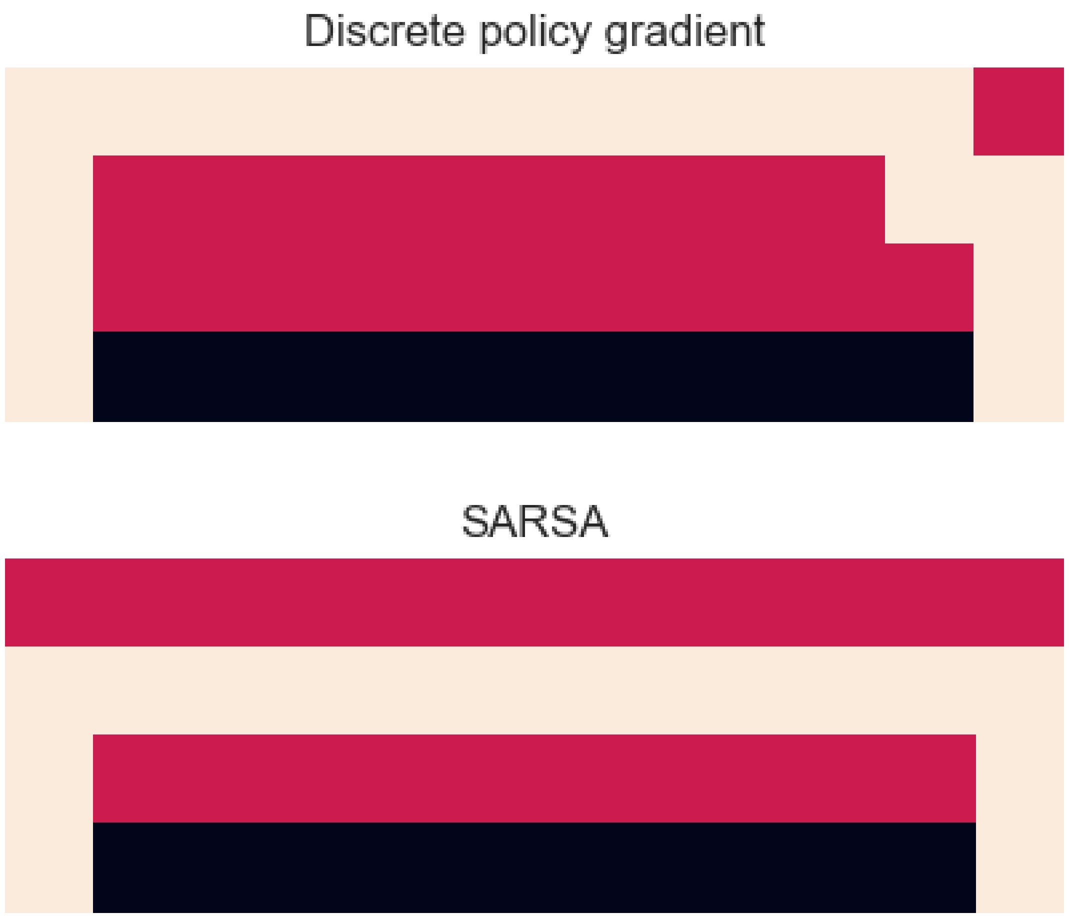
DIGITAL

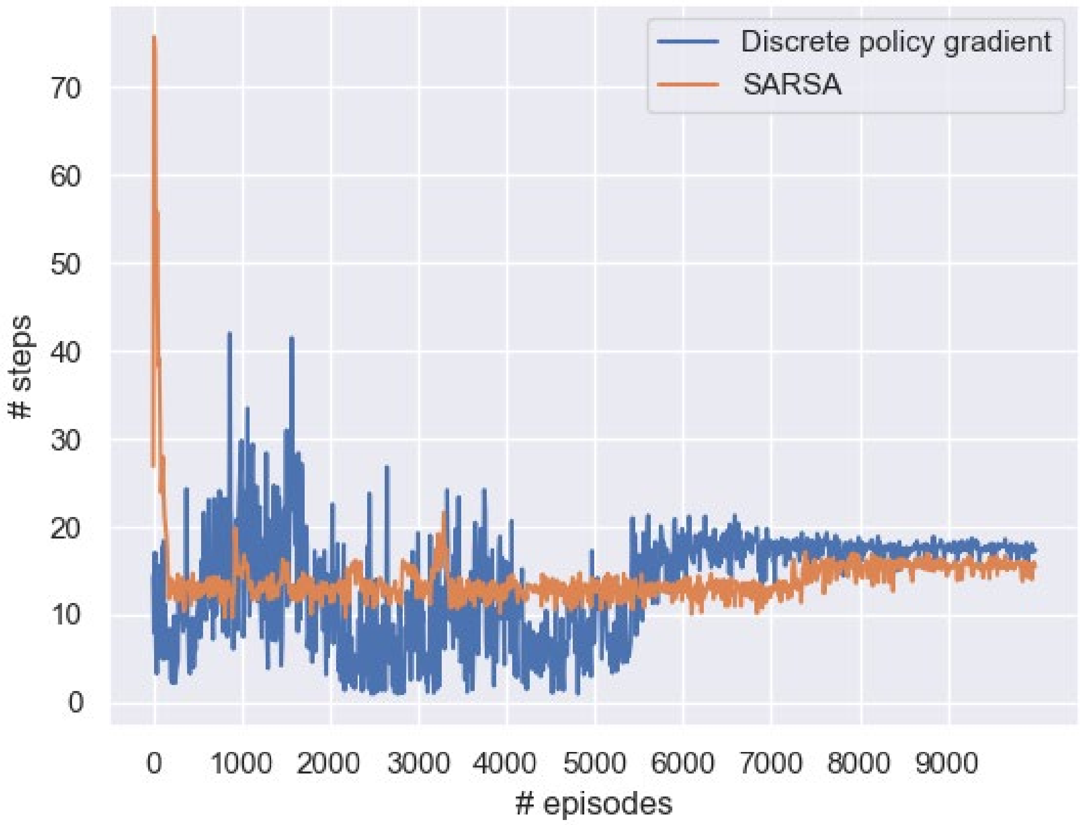
Cliff walking example – Policy learning

- Like SARSA and Monte Carlo learning, policy gradients are on-policy
  - Walking in the cliff contributes to actual reward
  - SARSA explored just 5%, much more exploration here (at least initially)
  - Also, reward variance (of full trajectory) is high, just as for Monte Carlo learning

  - What type of policy do you expect?

Cliff walking example – Resulting policy





Deriving the policy gradient method

Policy gradient derivation

- We now know the application, but why does it work?
  - Mathematically involved procedure based on calculus
  - For exam: not needed to understand all details, but ensure you can explain the rationale

DIGITAL

Policy gradient derivation – Objective function

- To start, we need an objective function that can be influenced by changing θ
- Like before, we optimize expected cumulative rewards over time:

$$J(\theta) = E_{\tau \sim \pi_\theta} R(\tau) = \sum_\tau P(\tau;\theta) R(\tau)$$

- Probability of trajectory $\tau = s_1, a_1, \ldots, s_T, a_T$ affected by θ
  - Note that $P(\tau;\theta)$ is conditional on θ
  - Rewards $R(\tau)$ depend on trajectory

Policy gradient derivation – Maximization function

- The optimization problem corresponds to maximizing the objective function

$$\max_{\theta} J(\theta) = \max_{\theta} \sum_{\tau} P(\tau;\theta) R(\tau)$$

- Aim to maximize the expected reward
- Alternatively: we alter the policy (parameterized by θ) in a way that increases the probability of high-reward trajectories
    - Remind that policy affects probability per action, and thus the trajectory probabilities

DIGITAL

Policy gradient derivation – Probability function

- Let's zoom in on the probability function

$$P(\tau;\theta) = \left[\prod_{t=0}^{T} \underbrace{P(s_{t+1} \mid s_t, a_t)}_{\substack{\text{Transition function} \\ \text{(environment model)}}} \cdot \underbrace{\pi_\theta(a_t \mid s_t)}_{\substack{\text{Policy} \\ \text{(control function)}}}\right]$$

- Two components:
  - Stochastic policy π_θ that samples an action
  - Transition function (time step) that includes ω like before

DIGITAL

Policy gradient derivation – Probability function

- Two problems with this model

$$P(\tau;\theta) = \left[\prod_{t=0}^{T} \underbrace{P(s_{t+1} \mid s_t, a_t)}_{\text{Transition function (environment model)}} \cdot \underbrace{\pi_\theta(a_t, s_t)}_{\text{Policy (control function)}}\right]$$

1. Transition function may be hard to model, or even unknown
2. Product of probabilities yields very small probabilities per trajectory
    - Programming languages have finite precision

- Let's leave these problems for now…

DIGITAL

Policy gradient derivation – Maximization function

- Let's revisit the maximization problem

$$\max_{\theta} J(\theta) = \max_{\theta} \sum_{\tau} P(\tau;\theta) R(\tau)$$

- If this function is differentiable, we can compute the gradient
  - Move policy into the direction of (local) op
    - Steep slope: large updates
    - Gentle slope: cautious updates

$$\nabla_{\theta} J(\theta) = \begin{bmatrix} \frac{\delta J(\theta)}{\delta \theta_1} \\ \frac{\delta J(\theta)}{\delta \theta_2} \\ \vdots \\ \frac{\delta J(\theta)}{\delta \theta_N} \end{bmatrix}$$

- How do we differentiate this function?

Policy gradient derivation – Rewriting [1/3]

- First, recall the equivalence between expected reward

$$\nabla_\theta J(\theta) = \nabla_\theta \mathbb{E}_{\tau \sim \pi_\theta} R(\tau)$$

and probability-weighted rewards

$$= \nabla_\theta \sum_\tau P(\tau;\theta) R(\tau)$$

- The probability-weighted expression is needed to apply sampling (i.e., Monte Carlo simulation)

DIGITAL

Policy gradient derivation – Rewriting [2/3]

- Next, let's bring the gradient sign within the sum

$$= \sum_{\tau} \nabla_{\theta} P(\tau;\theta) R(\tau)$$

- (a gradient of sums equals the sum of gradients)

DIGITAL

Policy gradient derivation – Rewriting [3/3]

- Now, we rewrite the expression to

$$= \sum_{\tau} P(\tau;\theta) \nabla_{\theta} \log P(\tau;\theta) R(\tau)$$

which again is an expectation:

$$= \mathbb{E}_{\tau \sim \pi_{\theta}} \nabla_{\theta} \log P(\tau;\theta) R(\tau)$$

- Let's pause for a moment and break down what happened

Policy gradient derivation – Log derivative trick [1/3]

- We have rewritten

$$= \sum_{\tau} \nabla_\theta P(\tau;\theta) R(\tau)$$

into

$$= \sum_{\tau} P(\tau;\theta) \nabla_\theta \log P(\tau;\theta) R(\tau)$$

Using the log derivative trick

DIGITAL

Policy gradient derivation – Log derivative trick [2/3]

$$= \sum_{\tau} \nabla_{\theta} P(\tau;\theta) R(\tau)$$

$$= \sum_{\tau} \boxed{\frac{P(\tau;\theta)}{P(\tau;\theta)}} \nabla_{\theta} P(\tau;\theta) R(\tau)$$

Multiply by $1 = \frac{P(\tau;\theta)}{P(\tau;\theta)}$

$$= \sum_{\tau} \frac{\boxed{P(\tau;\theta)} \nabla_{\theta} P(\tau;\theta)}{\boxed{P(\tau;\theta)}} R(\tau)$$

Rearrange

$$= \sum_{\tau} \boxed{P(\tau;\theta)} \boxed{\frac{\nabla_{\theta} P(\tau;\theta)}{\boxed{P(\tau;\theta)}}} R(\tau)$$

Rearrange again

$$= \sum_{\tau} P(\tau;\theta) \boxed{\nabla_{\theta} \log P(\tau;\theta)} R(\tau)$$

Substitute using 'log identity'

$$\frac{\nabla_{\theta} P(\tau;\theta)}{P(\tau;\theta)} = \nabla_{\theta} \log P(\tau;\theta)$$

DIGITAL

Policy gradient derivation – Log derivative trick [3/3]

- Log-derivative trick (some background)
  - The derivative of $\log x$ is $\frac{1}{x}$.
  - Combined with chain rule, we get:

$$\nabla_\theta \mathbb{P}(\tau|\theta) = \mathbb{P}(\theta|s_0)\nabla_\theta \log \mathbb{P}(\tau|\theta)$$

DIGITAL

Policy gradient derivation – Log probabilities [1/3]

- Remember that tricky probability function?

$$\nabla_\theta \log P(\tau;\theta) = \nabla_\theta \log \left[\prod_{t=0}^{T} P(s_{t+1} \mid s_t, a_t) \cdot \pi_\theta(a_t \mid s_t) \right]$$

- Turns out we already resolved it!
- We rewrote from probabilities to log probabilities:

$$= \nabla_\theta \left[\sum_{t=0}^{T} \log P(s_{t+1} \mid s_t, a_t) + \sum_{t=0}^{T} \log \pi_\theta(a_t \mid s_t) \right]$$

Policy gradient derivation – Log probabilities [2/3]

- Log probabilities are *additive* rather than *multiplicative*
  - Transition $P(s_{t+1}|s_t, a_t)$ does not depend on θ
  - We can strike it without affecting the gradient

$$= \nabla_\theta \left[\cancel{\sum_{t=0}^{T} \log P(s_{t+1} \mid s_t, a_t)} + \sum_{t=0}^{T} \log \pi_\theta(a_t \mid s_t) \right]$$

- We may not know the transition function, but we do know our policy
  - If we can differentiate π_θ, we can compute the gradient of the objective function

DIGITAL

Policy gradient derivation – Log probabilities [3/3]

- Resulting expression only depends on policy π_θ:

$$= \nabla_\theta \sum_{t=0}^{T} \log \pi_\theta(a_t \mid s_t)$$

- Move in the gradient again, and we have our result:

$$= \sum_{t=0}^{T} \nabla_\theta \log \pi_\theta(a_t \mid s_t)$$

- As a bonus, additive probabilities are numerically more stable

DIGITAL

Policy gradient derivation – To summarize

$$
\begin{aligned}
\nabla_\theta \log P(\tau;\theta) &= \nabla_\theta \log \left[\prod_{t=0}^{T} P(s_{t+1} \mid s_t, a_t) \cdot \pi_\theta(a_t, s_t) \right] \\
&= \nabla_\theta \left[\sum_{t=0}^{T} \log P(s_{t+1} \mid s_t, a_t) + \sum_{t=0}^{T} \log \pi_\theta(a_t, s_t) \right] \\
&= \nabla_\theta \left[\sum_{t=0}^{T} \log P(s_{t+1} \mid s_t, a_t) + \sum_{t=0}^{T} \log \pi_\theta(a_t, s_t) \right] \\
&= \nabla_\theta \left[\prod_{t=0}^{T} \log P(s_{t+1} \mid s_t, a_t) + \sum_{t=0}^{T} \log \pi_\theta(a_t, s_t) \right] \\
&= \nabla_\theta \sum_{t=0}^{T} \log \pi_\theta(a_t, s_t) \\
&= \sum_{t=0}^{T} \nabla_\theta \log \pi_\theta(a_t, s_t)
\end{aligned}
$$

Policy gradient derivation – Approximate gradient

- It is worth mentioning we use approximate gradients
  - Like before, we approximate by repeated sampling

$$\nabla_\theta J(\theta) \approx \hat{g} = \frac{1}{m} \sum_{i=1}^{m} \nabla_\theta \log P\left(\tau^{(i)}; \theta\right) R\left(\tau^{(i)}\right)$$

  - Monte Carlo rationale (Law of Large Numbers)

DIGITAL

Applying policy gradients

Applying policy gradients

- We have the final result now, but it may seem rather abstract

- In practice, two policies are used for the most part
  - Softmax policy: for discrete action spaces
    - Assign probability to each action
  - Gaussian policy: for continuous action spaces
    - Draw action from normal distribution

DIGITAL

Discrete policy gradient

- Softmax policy:

$$\pi_\theta(s,a) = \frac{e^{\phi(s,a)^\top \theta}}{\sum_{a' \in A} e^{\phi(s,a')^\top \theta}}$$

- Gradient of softmax policy

$$\nabla_\theta \log \pi_\theta(a \mid s) = \phi(s,a) - \sum_{a' \in A} \phi(s,a')\pi_\theta(a \mid s)$$

- Remember: gradient is the score function seen earlier

DIGITAL

Continuous policy gradient – Gaussian policy

- Gaussian policy (parameters are μ_θ and σ_θ):

$$\pi_\theta(a \mid s) = \frac{1}{\sqrt{2\pi\sigma_\theta}} e^{-\frac{a-\mu_\theta}{2\sigma_\theta^2}}$$

- Gradient of Gaussian policy

$$\nabla_\theta \log(\pi_\theta(a \mid s)) = \frac{(a-\mu_\theta)\phi(s)}{\sigma_\theta^2}$$

- For each differentiable policy, we can compute the score function

Continuous policy gradient – Gaussian policy

- Update functions for μ_θ and σ_θ

$$\Delta_{\mu_{\boldsymbol{\theta}}}(s) = \alpha v \frac{(a - \mu_{\boldsymbol{\theta}}(s))}{\sigma_{\boldsymbol{\theta}}^2} \,,$$

$$\Delta_{\sigma_{\boldsymbol{\theta}}}(s) = \alpha v \frac{(a - \mu_{\boldsymbol{\theta}}(s))^2 - \sigma_{\boldsymbol{\theta}}^2}{\sigma_{\boldsymbol{\theta}}^3} \,.$$

DIGITAL

REINFORCE algorithm

- Mathematics may be overwhelming, but policy gradient algorithm itself is compact
- Consider the REINFORCE algorithm (Williams, 1993)

Input: $\theta \leftarrow \mathbb{R}^{|\theta|}$ ▶ Initialize θ

for $n = 1$ **to** N **do:**

    $\tau \sim \pi_\theta$ ▶ Generate state-action trajectory with π_θ

    for $t = 1$ **to** T **do**

        $R(\tau \mid t) = R_t + R_{t+1} + \ldots + R_T$ ▶ Compute cum. reward

        $\theta \leftarrow \theta + \alpha R(\tau \mid t) \nabla_\theta \log(\pi_\theta(a \mid s)$ ▶ Update θ

- In the end we only need:
  - A differentiable stochastic policy
  - A sequence of observed rewards, states, and actions

DIGITAL

Application

- Like before, we can define features and weights
  - Basis functions $\phi_f: (s, a) \rightarrow \mathbb{R}$, returning relevant features that explain the key determinants of value
  - Linear expression multiplies weight vector θ and feature vector $\phi(s, a)$:

$$\pi_\theta(s, a) = \frac{e^{\boxed{\phi(s,a)^\top \theta}}}{\sum_{a' \in A} e^{\phi(s,a')^\top \theta}}$$

DIGITAL

Example discrete space

- Example discrete action space:
  - Container shipping (each subset that can be shipped yields attached probability)
    - Features $\phi(s, a)$ could be *#red containers* etc., like before
    - By defining features, we resolve dimensionality of state space (like VFA did)
    - Softmax policy requires looping over all actions (denominator), so action space problem <u>is not resolved</u>

DIGITAL

Example continuous space

- Learn bid price for financial asset:
  - Bid can be accepted or rejected
  - $Payoff - bid$ yields the reward of the action
  - Action is a real-valued decision variable
    - Features $\phi(s, a)$ could capture asset properties (drift, volatility)
    - Try to learn minimal bid that gets accepted
    - Gaussian policy simply draws an action, so action space problem is resolved here

DIGITAL

Deep policy gradient

Deep policy gradients

- Simplest form of policy gradient:
  - Linear expression: $\theta^{\top}\phi(s,a)$
  - May not capture complex (non-linear) patterns
  - Human design capabilities have limits

- We may express policy as a neural network
  - Like in Deep Q-learning, the network generalizes across states
    - Implicitly generates features in hidden layers
  - Often more challenging to train than Deep Q-learning

DIGITAL

Deep policy gradients

- Softmax policy revisited
- Almost the same, we just replace $\phi(s,a)^\top\theta$ with f($\phi(s,a);\theta$), with f representing the neural network parameterized by θ

$$\pi_\theta(s,a) \propto e^{f(\phi(s,a);\theta)}$$

$$\pi_\theta(s,a) = \frac{e^{f(\phi(s,a);\theta)}}{\sum_{a' \in \mathcal{A}} e^{f(\phi(s,a');\theta)}}$$

DIGITAL

Deep policy gradients

- Output can be converted (Tensorflow here) to softmax:

```python
def construct_actor_network(STATE_DIM: int, ACTION_DIM: int):
    """Construct the actor network with action probabilities as output"""
    inputs = layers.Input(shape=(STATE_DIM,))  # input dimension
    hidden1 = layers.Dense(
        25, activation="relu", kernel_initializer=initializers.he_uniform()
    )(inputs)
    hidden2 = layers.Dense(
        25, activation="relu", kernel_initializer=initializers.he_uniform()
    )(hidden1)
    hidden3 = layers.Dense(
        25, activation="relu", kernel_initializer=initializers.he_uniform()
    )(hidden2)
    probabilities = layers.Dense(
        ACTION_DIM, kernel_initializer=initializers.Ones(), activation="softmax"
    )(hidden3)

    actor_network = keras.Model(inputs=inputs, outputs=[probabilities])
```

Activation function in output layer applies softmax

DIGITAL

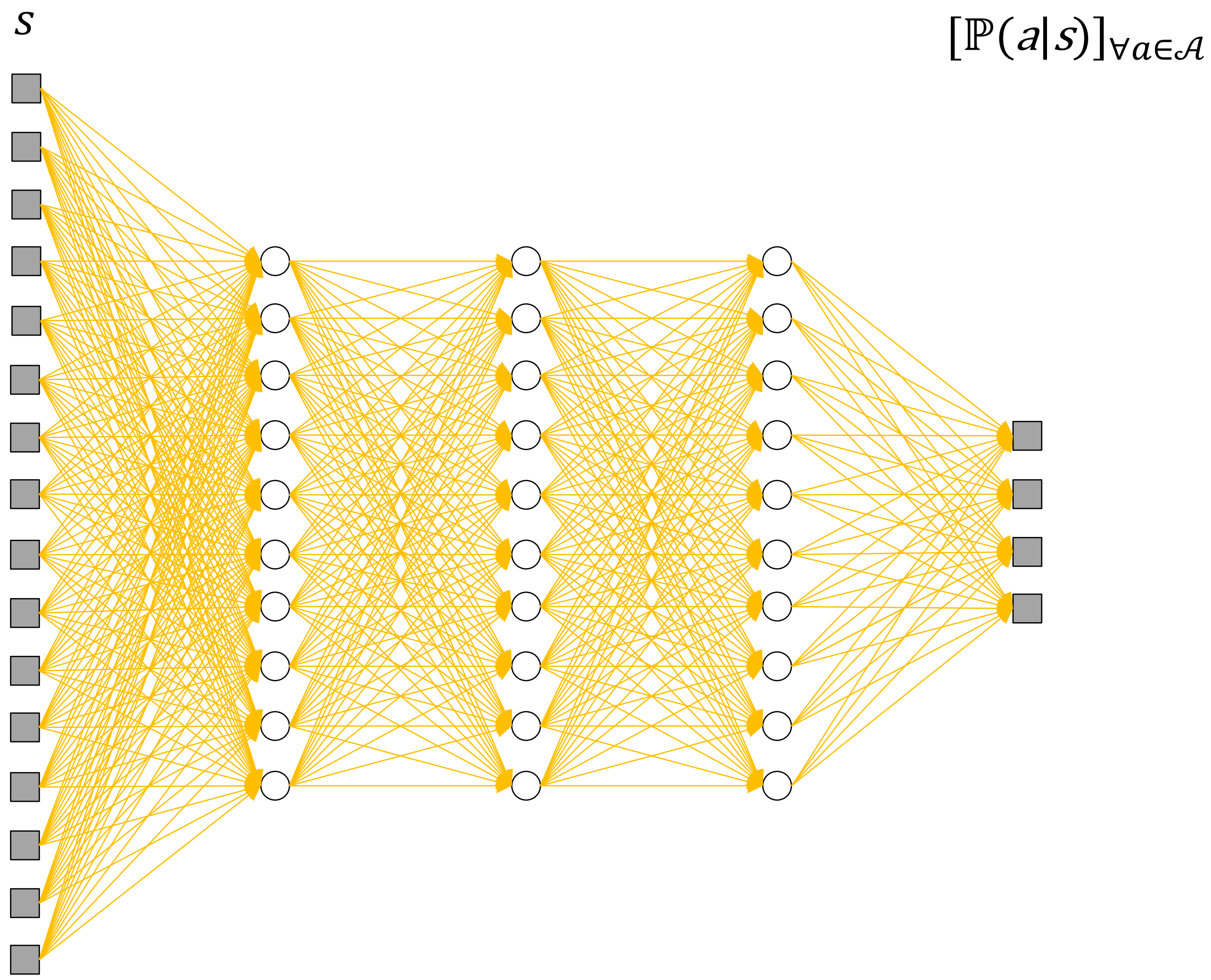
Deep policy gradients

- Basic architecture
  - Input: state- or feature vector
  - Hidden layers
    - Activation functions (e.g., ReLU)
    - Weights θ (i.e., network weights are the tunable parameters!)
  - Output: action probabilities
    - For softmax policy, dimension of output layer equal to $|\mathcal{A}|$
    - For Gaussian policy, output is $\mu_\theta, \sigma_\theta$

- Loss function is policy-dependent
  - Often not basic Mean Squared Error

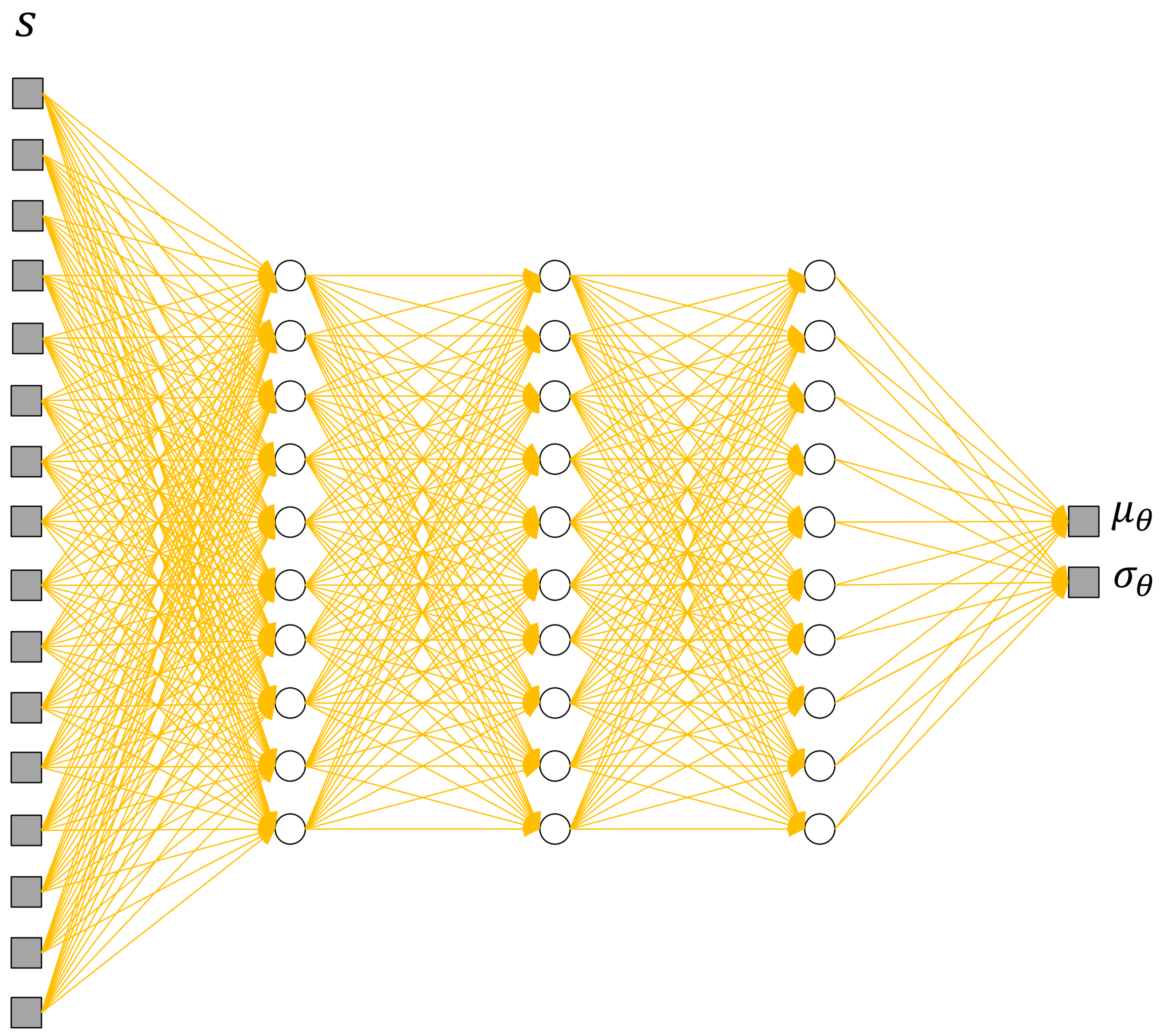
DIGITAL

Actor network – Discrete action space



DIGITAL

Actor network – Continuous action space



DIGITAL

Deep policy gradient

- Loss function is different now!
- For Q-network, we used standard Mean Squared Error
  - Difference between expected- and ‘observed’ Q-values
- For policy network, what would be the loss?

DIGITAL

Deep policy gradient

- Loss function is different now!
- For Q-network, we used standard Mean Squared Error
  - Difference between expected- and 'observed' Q-values

- For policy network, what would be the loss?
  - We have no 'true' value
  - What we do have:
    - Reward signal
    - Probability of action under current policy
  - Select actions with probabilities that maximize reward signal
  - Measure 'error' of policy in some way

DIGITAL

Deep policy gradient

- Remember the update rule for policy gradients

$$\Delta\theta = \alpha \nabla_\theta \log\left(\pi_\theta(a \mid s)\right) v$$

- This rule is based on gradient *ascent*
- Neural network is trained with gradient *descent*
  - Add minus sign
  - Remove learning rate α and gradient ∇_θ
    - Loss function is only the input for the gradient computations
  - Generic loss function depends on log prob action and reward

$$\mathcal{L}(a, s, v) = -\log(\pi_\theta(a \mid s))v$$

DIGITAL

Deep policy gradient

- To train the policy network, we re-express the generic loss function for the policy:

$$\mathcal{L}(a, s, v) = -\log(\pi_\theta(a \mid s))v$$

- Requires writing out the policy and resolving the expression

- Tensorflow tracks loss function on *GradientTape*
  - May require manual design

DIGITAL

Deep policy gradient – Loss function softmax policy

- Softmax uses cross entropy loss

$$\mathcal{L}(a, s, v) = -\log(\pi_\theta(a \mid s))v$$

(happens to be identical to generic form)

```python
def cross_entropy_loss(probability_action, reward):
    log_probability = tf.math.log(probability_action + 1e-5)
    loss_actor = - reward * log_probability

    return loss_actor
```

DIGITAL

Deep policy gradient – Loss function Gaussian policy

- Gaussian uses normal loss

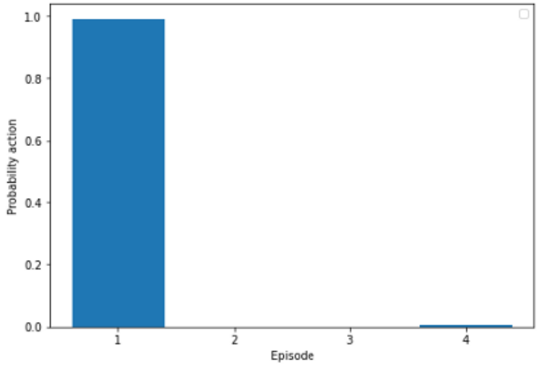
$$\mathcal{L}(a, s, v) = -\log\left(\frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{1}{2}\left(\frac{a-\mu}{\sigma}\right)^2}\right) v$$

```python
"""Weighted Gaussian log likelihood loss function for RL"""
def custom_loss_gaussian(state, action, reward):
  # Predict mu and sigma with actor network
  mu, sigma = actor_network(state)

  # Compute Gaussian pdf value
  pdf_value = tf.exp(-0.5 *((action - mu) / (sigma))**2)
     * 1 / (sigma * tf.sqrt(2 * np.pi))

  # Convert pdf value to log probability
  log_probability = tf.math.log(pdf_value + 1e-5)

  # Compute weighted loss
  loss_actor = - reward * log_probability

  return loss_actor
```

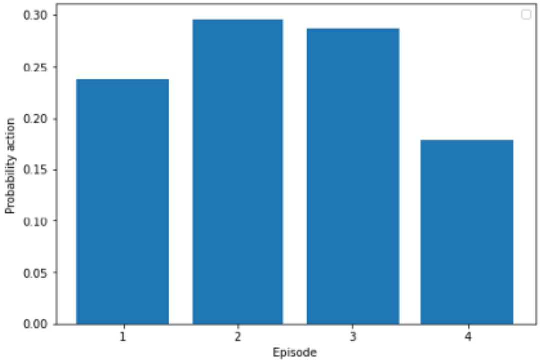
DIGITAL

Examples

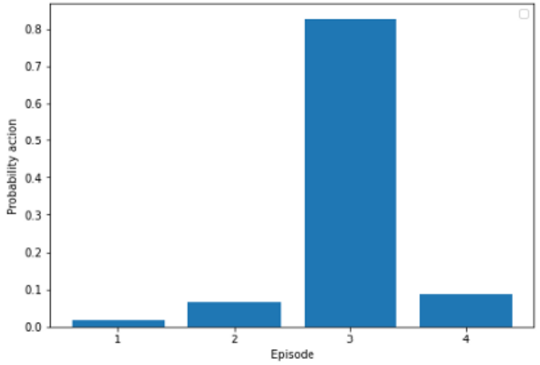
https://github.com/woutervanheeswijk/example_discrete_control



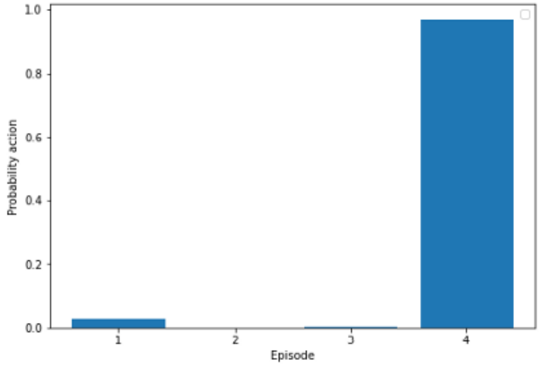
(a) $\mu_1 = 1.1$, $\mu_2 = 0.0$, $\mu_3 = 1.0$, $\mu_4 = 1.0$



(b) $\mu_1 = 1.0$, $\mu_2 = 1.0$, $\mu_3 = 1.0$, $\mu_4 = 1.0$



(c) $\mu_1 = 4.8$, $\mu_2 = 4.9$, $\mu_3 = 5.1$, $\mu_4 = 4.9$



(d) $\mu_1 = 1.0$, $\mu_2 = 0.9$, $\mu_3 = 0.9$, $\mu_4 = 1.0$

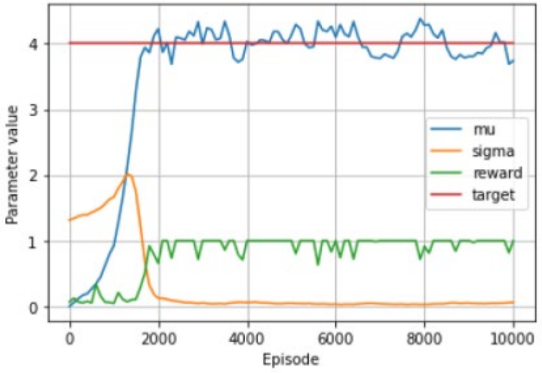
DIGITAL

Examples

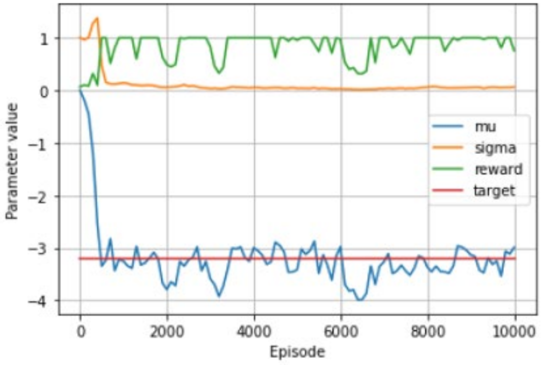
https://github.com/woutervanheeswijk/example_continuous_control

- Red line = unknown target
- Closer to target = high reward

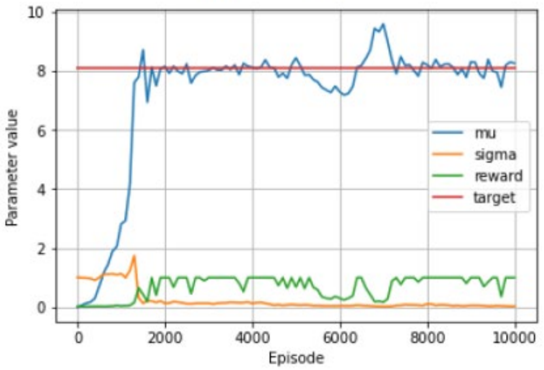
- Far from target → increase σ
- Close to target → decrease σ



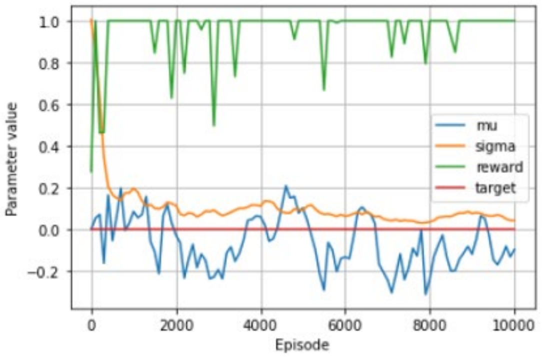
(a) $\tau = 4.0$



(b) $\tau = -3.2$



(c) $\tau = 8.1$



(d) $\tau = 0.0$

Actor-critic models

VFAs and PFAs

- As mentioned in Lecture 1, there are 4 policy classes
- So far, we treated two:
  - Value Function Approximation (VFA): learn $\bar{V}_\theta(\phi(s,a))$
  - Policy Function Approximation (PFA): learn π_θ

- Why not combine the best of both worlds?

DIGITAL

Four policy classes – Refresher

Policy-based methods	**Value-based methods**
Policy function approximation (PFA)	Value function approximation (VFA)
Cost function approximation (CFA)	Direct lookahead approximation (DLA)

Adjust decision-making rules and observe impact

Learn value functions that represent downstream values

DIGITAL

VFAs and PFAs

- Benefits **VFA**
  - Capture downstream rewards in generic function
  - Value functions help when not fully grasping downstream effects
- Benefits **PFA**
  - Exploit knowledge of policy structure
  - Directly influence decision rules

DIGITAL

Hybrid models

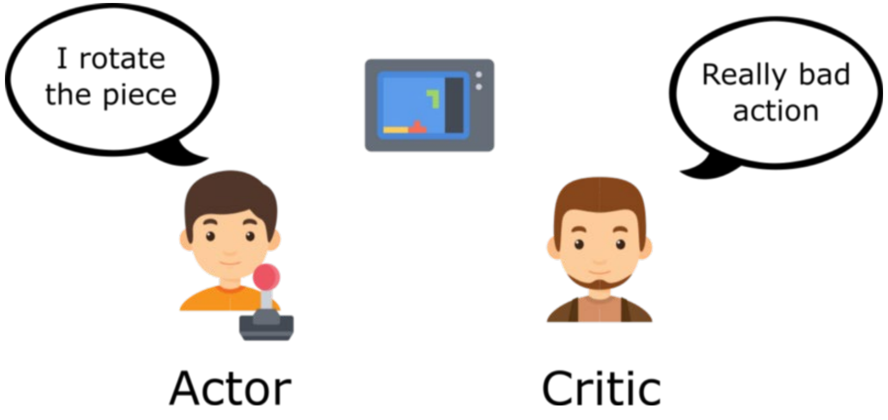
- Possible to combine different classes
  - Also recall Direct Lookahead Approximation (heuristic sampling of downstream effects) and Cost Function Approximation (parameterize uncertainty directly into policy)

- Leverage strengths of multiple classes while negating weaknesses
  - Poor implementations may achieve the opposite!

DIGITAL

Actor-critic models [1/5]

- Actor-critic models are the standard in Reinforcement Learning
  - Mostly applications in Computer Science



Source: freecodecamp.org

- In fact, you have already seen an actor-critic model!
  - Policy iteration algorithm (Lecture 1) entails both value function updates and policy updates

DIGITAL

Actor-critic models [2/5]

- The actor-critic model combines PFA and VFA, recombining the techniques you have seen before
  - Update function policy (using Q_w instead of G_t)
    - $\theta \leftarrow \theta + \alpha_\theta Q_w(s,a)\nabla_\theta(a|s)$
  - For value function, compute temporal difference error:
    - $\delta_t = r_t + \gamma Q_w(s',a') - Q_w(s',a')$ [TD(0) error]
  - Update value function weights with TD(0) error
    - $w \leftarrow w + \alpha_w \delta_t \nabla_w Q_w(s,a)$

- Learning both policy ('actor') and downstream effects ('critic')

DIGITAL

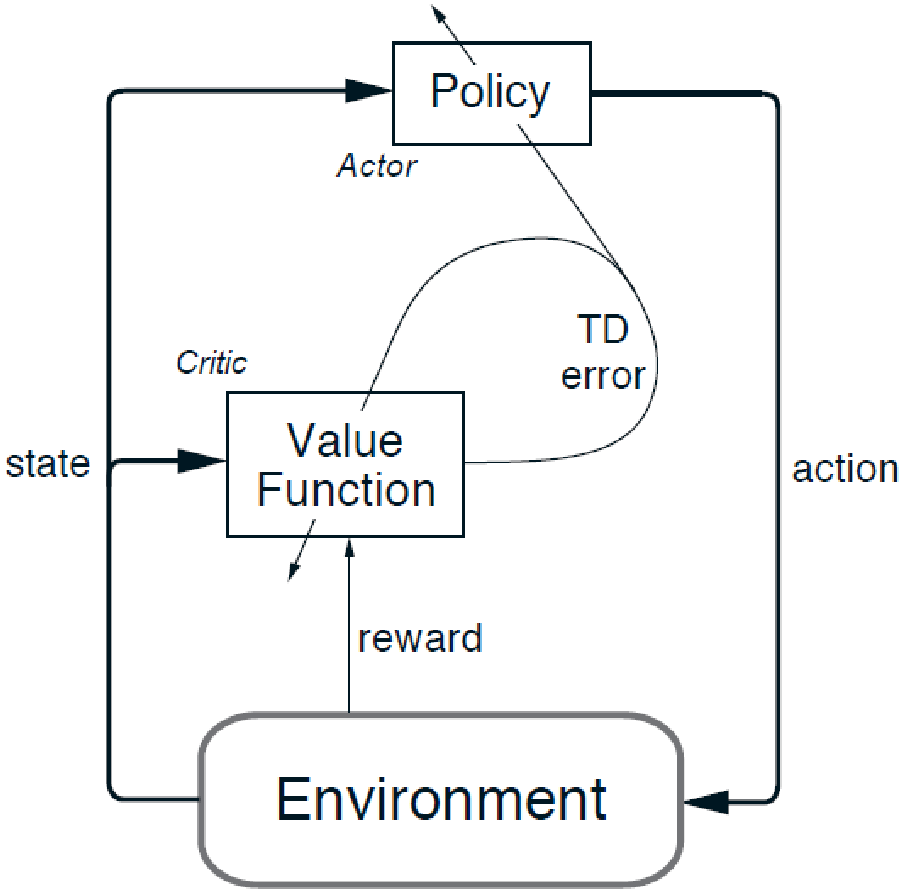
Actor-critic models [3/5]

- See Sutton for details

One-step Actor–Critic (episodic)

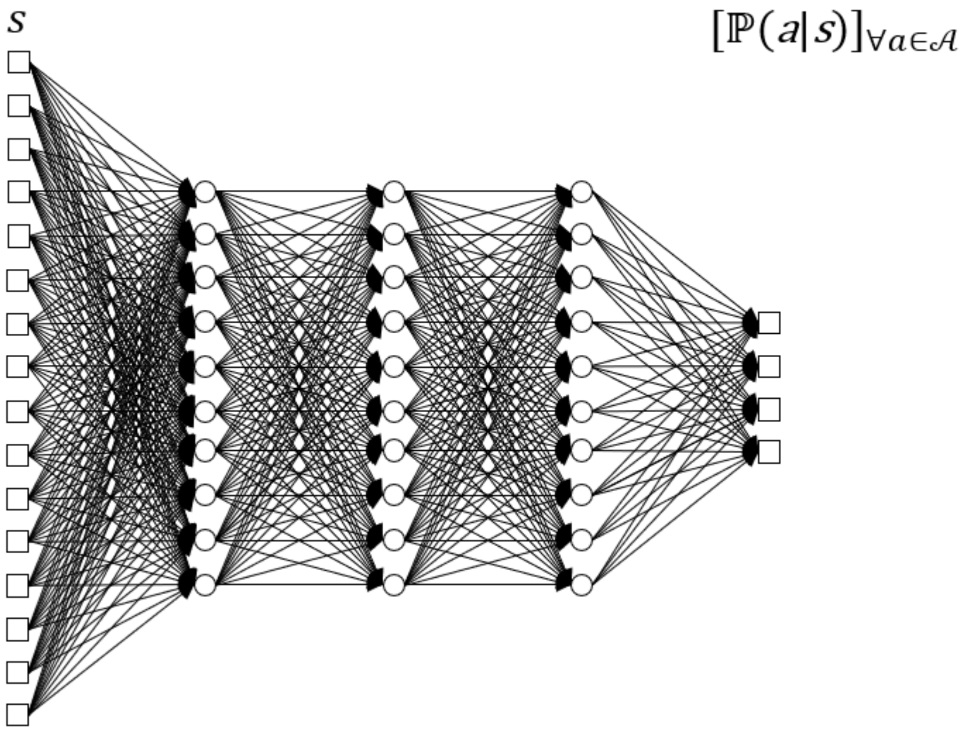
Input: a differentiable policy parameterization $\pi(a|s,\boldsymbol{\theta})$
Input: a differentiable state-value parameterization $\hat{v}(s,\mathbf{w})$
Parameters: step sizes $\alpha^{\boldsymbol{\theta}} > 0$, $\alpha^{\mathbf{w}} > 0$

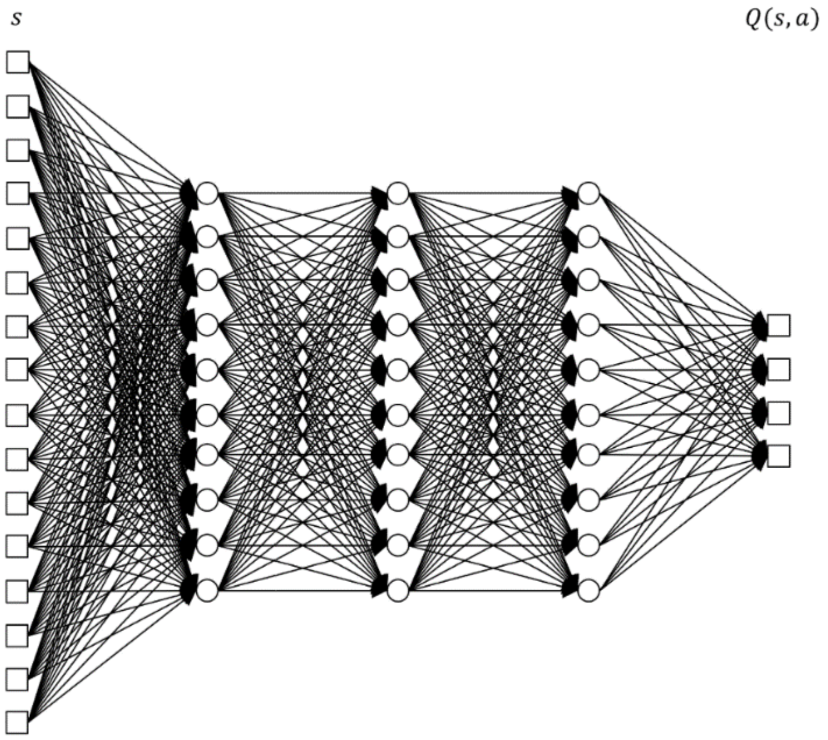
Initialize policy parameter $\boldsymbol{\theta} \in \mathbb{R}^{d'}$ and state-value weights $\mathbf{w} \in \mathbb{R}^{d}$
Repeat forever:
  Initialize S (first state of episode)
  $I \leftarrow 1$
  While S is not terminal:
    $A \sim \pi(\cdot|S,\boldsymbol{\theta})$
    Take action A, observe S', R
    $\delta \leftarrow R + \gamma\hat{v}(S',\mathbf{w}) - \hat{v}(S,\mathbf{w})$ (if S' is terminal, then $\hat{v}(S',\mathbf{w}) \doteq 0$)
    $\mathbf{w} \leftarrow \mathbf{w} + \alpha^{\mathbf{w}} I \delta \nabla_{\mathbf{w}} \hat{v}(S,\mathbf{w})$
    $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \alpha^{\boldsymbol{\theta}} I \delta \nabla_{\boldsymbol{\theta}} \ln \pi(A|S,\boldsymbol{\theta})$
    $I \leftarrow \gamma I$
    $S \leftarrow S'$



Actor-critic models [4/5]

- Nowadays, RL is often network-based
  - Actor network (policy network returning action probabilities)
  - Critic network (value network returning Q-values)





DIGITAL

Actor-critic models [5/5]

- Main benefit of actor-critic models:
  - Learn Q-values instead of observed rewards G_t
    - As reward trajectories may vary a lot, Q-values can add robustness

- Main challenges:
  - Adjusting policy changes value functions
  - Adjusting value functions changes policy
  - Must be updated simultaneously...

DIGITAL

Advanced policy gradients

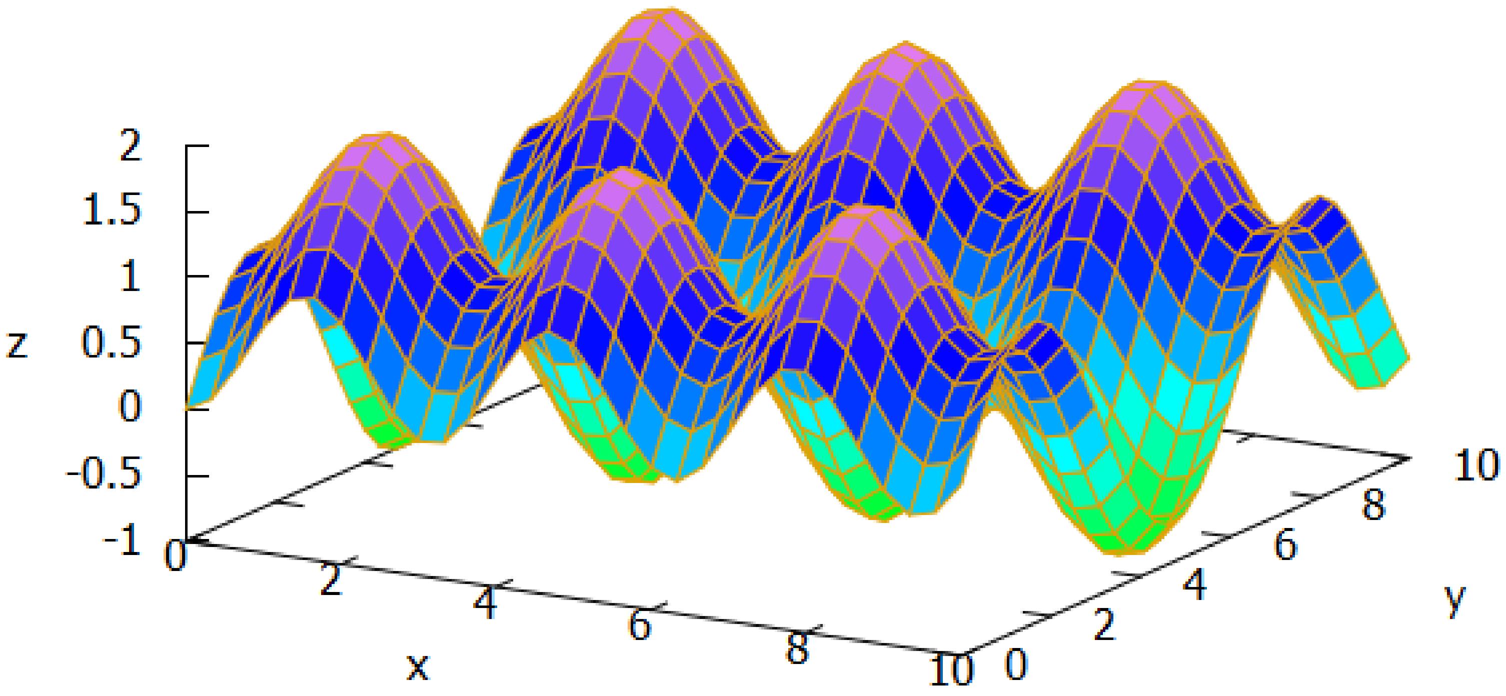
Advanced policy gradients

- Advanced material, in-depth explanation would require multiple lectures
- For exam, just try to understand high-level idea

DIGITAL

Problems with policy gradients [1/3]

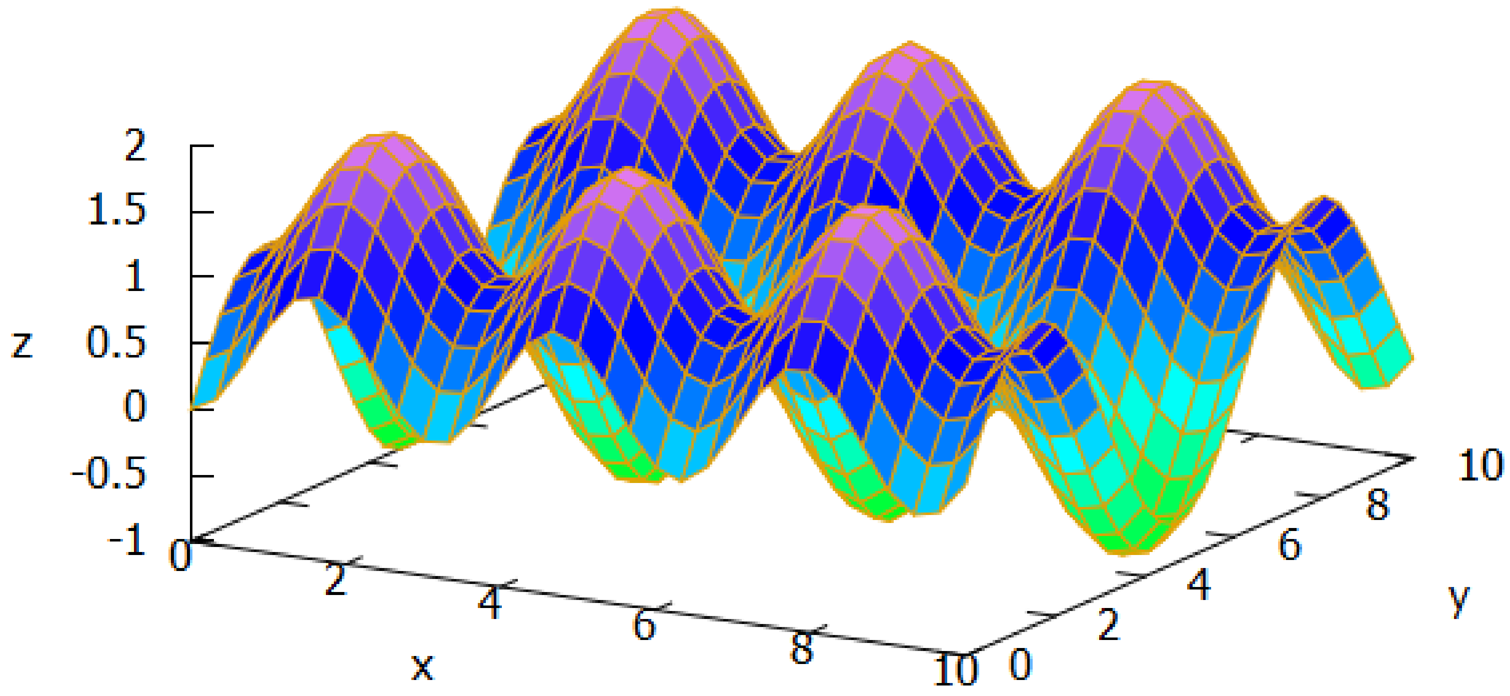
- What is a good stepsize for policy gradient algorithms?



DIGITAL

Problems with policy gradients [2/3]

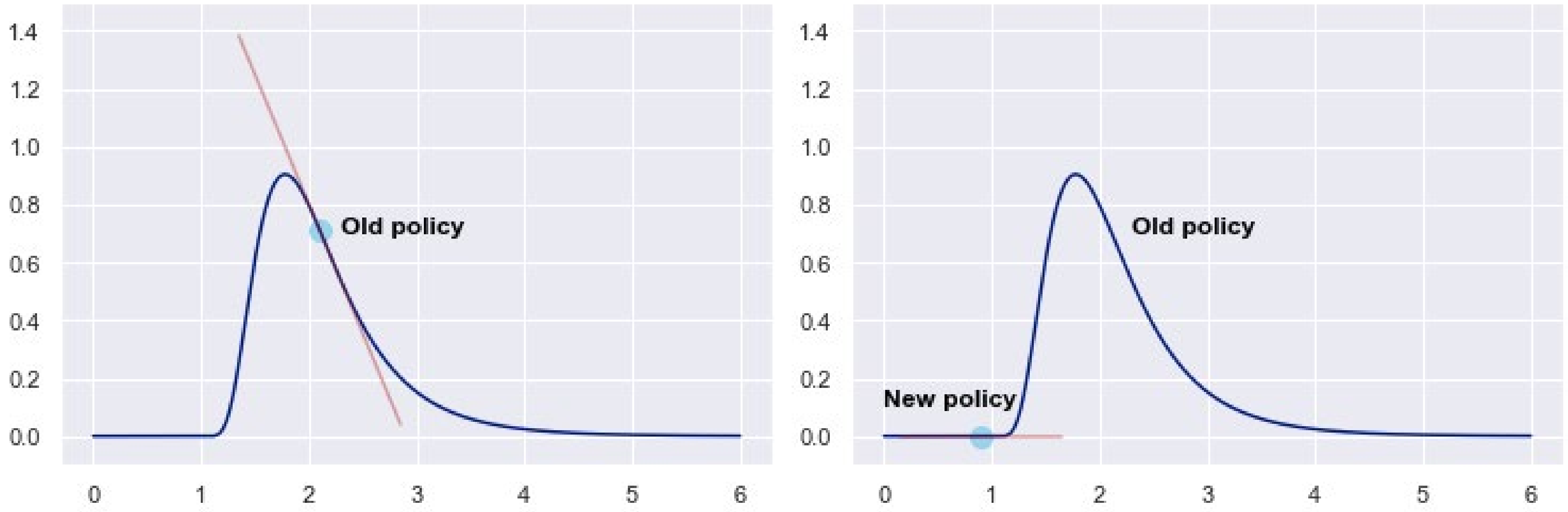
- What is a good stepsize for policy gradient algorithms?
  - Common rationale: steep slope = large step
  - Is there a downside to this rationale?



DIGITAL

Problems with policy gradients [3/3]

- Common problems with policy gradients
  - Overshooting: large update, miss reward peak
  - Undershooting: stuck at suboptimal plateaus



Natural policy gradients [1/4]

- Essentially, we don't want policies to change too much
  - On plateau, we can safely take larger steps
  - On steep slopes, we want to be cautious
  - Opposite of rationale so far

- How do we incorporate this behavior?

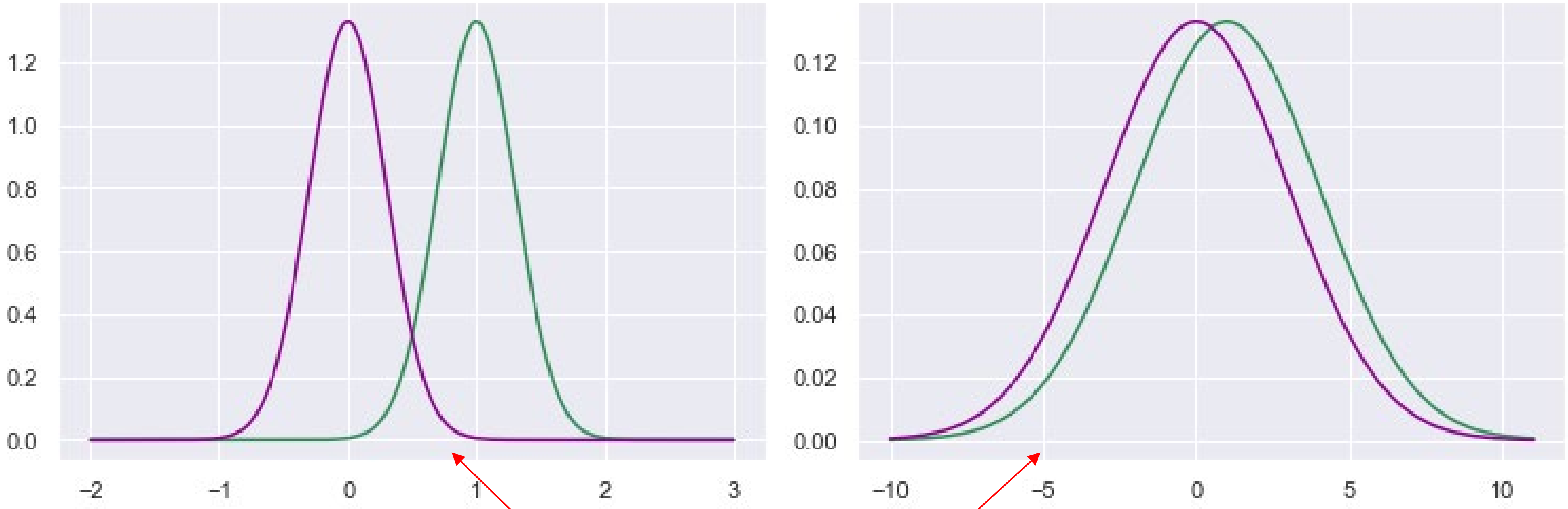
DIGITAL

Natural policy gradients [2/4]

- Capping parameter updates does not work

$$\Delta\theta^* = \arg\max_{\|\Delta\theta\|\le\epsilon} J(\theta + \Delta\theta)$$

Suppose we change $\theta = [\mu, \sigma]$



Same parameter distance ϵ!

DIGITAL

Natural policy gradients [3/4]

- We *can* cap difference between policy before and after update
  - Formalized by Kullback-Leibner (KL) divergence:

$$\mathcal{D}_{\mathrm{KL}}(\pi_\theta \parallel \pi_{\theta+\Delta\theta}) = \sum_{x \in \mathcal{X}} \pi_\theta(x) \log\left(\frac{\pi_\theta(x)}{\pi_{\theta+\Delta\theta}(x)}\right)$$

- We then determine optimal constrained update:

$$\Delta\theta^* = \underset{\mathcal{D}_{\mathrm{KL}}(\pi_\theta \| \pi_{\theta+\Delta\theta}) \leq \epsilon}{\arg\max} \; J(\theta + \Delta\theta)$$

DIGITAL

Natural policy gradients [4/4]

- Natural policy gradients correct for second derivatives, ensuring the policy does not change too much
- Full procedure is rather involved (second-order derivatives, Fisher information matrix, Langrangian relaxation, Taylor expansions, etc.)

- Bottom line:
  - Restrict update size based on local sensitivity of objective
  - Dynamic step size given by
  - Requires substantial memory and $\Delta\theta = \sqrt{\frac{2\epsilon}{\nabla J(\theta)^\top F(\theta)^{-1} \nabla J(\theta)}} \tilde{\nabla} J(\theta)$ resources
    - Inverting large matrices is hard!

Trust Region Policy Optimization (TRPO) [1/2]

- Natural policy gradients based on strong assumptions and requires substantial computational resources
- TRPO proposes three improvements
  - Conjugate gradient: numerical approximation, no need to invert Fisher information matrix
  - Line search: iteratively reduce update size, until maximum divergence is satisfied
  - Improvement check: *check* whether update improves policy, rather than *assume*

- Still memory-intense, complex second-order optimization, etc.

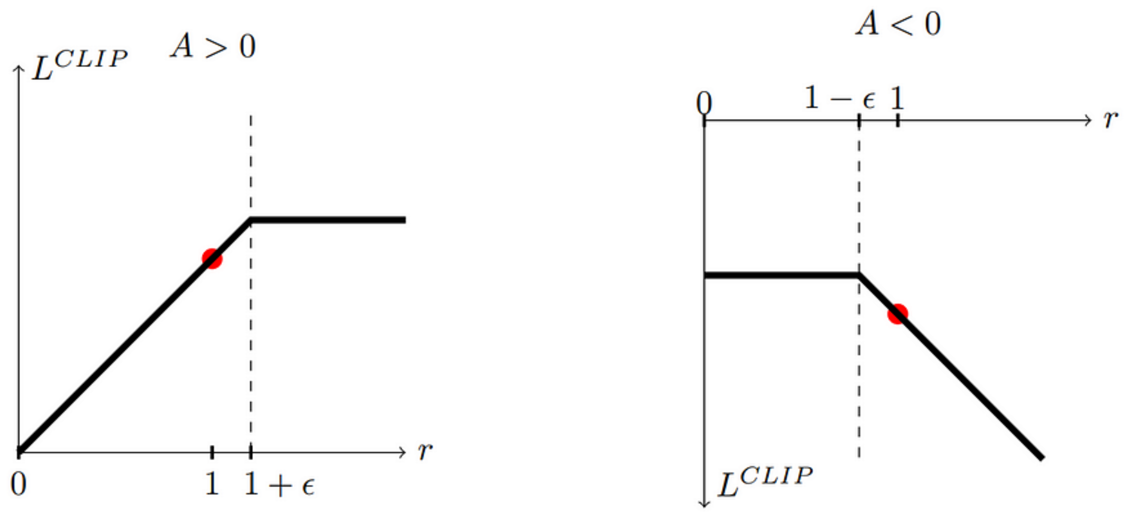
DIGITAL

Trust Region Policy Optimization (TRPO) [2/2]

DIGITAL

Proximal Policy Optimization (PPO) [1/2]

- Natural policy gradients and TRPO are complex second-order methods
- PPO just tosses updates that drift too far from current policy
  - Empirically works well
  - Can apply excellent optimizers such as ADAM (first-order)



[image by Schulman et al. 2017]

DIGITAL

Proximal Policy Optimization (PPO) [2/2]

- Clipped loss function
  - If loss is $1 \pm \epsilon$, gradient is 0!

$$\mathcal{L}_{\pi_\theta}^{CLIP}(\pi_{\theta_k}) = \mathbb{E}_{\tau\sim\pi_\theta}\left[\sum_{t=0}^{T}\left[\min\left(\rho_t(\pi_\theta,\pi_{\theta_k})A_t^{\pi_{\theta_k}}, \mathrm{clip}(\rho_t(\pi_\theta,\pi_{\theta_k}), 1-\epsilon, 1+\epsilon)A_t^{\pi_{\theta_k}}\right)\right]\right]$$

- Ratio: difference action probability before and after update

$$\rho_t(\theta) = \frac{\pi_\theta(a_t \mid s_t)}{\pi_{\theta_k}(a_t \mid s_t)}$$

- Simple to implement, works well with neural networks

DIGITAL

Wrapping up

Recap

- Policy gradients
  - Policy Function Approximation (PFA) methods directly alter the policy, not requiring the Bellman paradigm
  - By using a stochastic policy, we observe differences between actions and compute gradients for update directions
  - Deep policy learning deploys actor network to capture nonlinear effects and extract features

DIGITAL

Further reading

- Sutton & Barto (2018)

  Chapter 11.1 (actor-critic models)

- David Silver (2020) [policy gradients]

https://www.davidsilver.uk/wp-content/uploads/2020/03/pg.pdf

DIGITAL

DIGITAL

Funded by
the European Union

Funded by the European Union. Views and opinions expressed are however those of the author(s) only and do not necessarily reflect those of the European Union or Horizon Europe: Marie Skłodowska-Curie Actions. Neither the European Union nor the granting authority can be held responsible for them.

This project has received funding from the European Union's Horizon Europe research and innovation programme under the Marie Skłodowska-Curie grant agreement No. 101119635

DIGITAL