Reinforcement Learning: From Exact Optimization Methods to Machine Learning

Andreas Ktenidis

George Zois

Introduction to Reinforcement Learning and Its Application to Job Shop Scheduling

Exact Vs Learning

Exact Optimization Methods

- **Linear / Mixed-Integer Programming:** optimize an objective under algebraic constraints
  Example: production planning, assignment, scheduling
- **Constraint Programming:** search for feasible solutions under logical and temporal constraints
  Example: Job Shop Scheduling, timetabling, routing
- **Dynamic Programming:** solve problems by decomposing them into smaller subproblems
  Example: shortest paths, inventory control, small sequential decisions
- **Branch-and-Bound / Branch-and-Cut:** systematically explore the solution space with pruning
  Example: MILP solvers,

Learning-Based Methods

- **Supervised Learning:** learn classification or regression models from labeled data
  Example: image classification, demand forecasting, defect detection
- **Unsupervised Learning:** discover patterns in data without labels
  Example: clustering, anomaly detection, dimensionality reduction
- **Reinforcement Learning:** learn actions through interaction with an environment and rewards
  Example: robotics, games, adaptive scheduling, resource allocation
- **Self-Supervised Learning:** representation learning from automatically generated targets

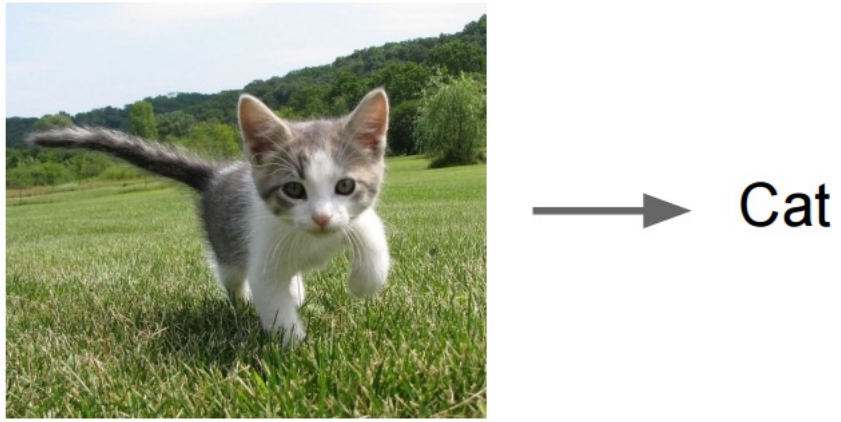
Data: (x, y) x is data, y is label

Goal: Learn a function to map x -> y

Classification : Email Spam – No Spam

Regression : Predict the price of house

Example
Linear Regression
Logistic Regression



Supervised Learning

The task of inferring a classification or regression from labeled training data

Unsupervised Learning

Data: x Just data, no labels!

Goal: Learn some underlying hidden structure of the data

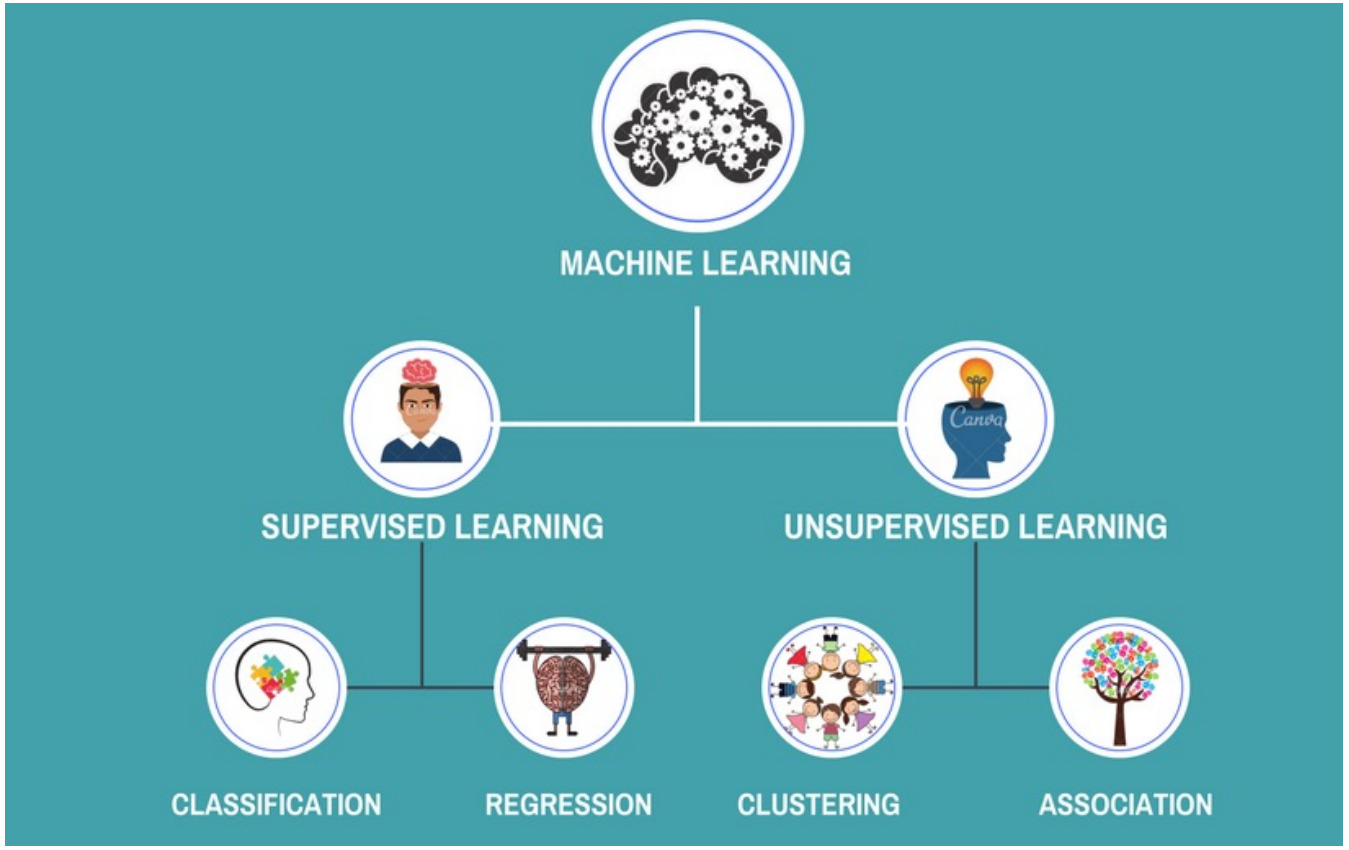
Clustering

Real – world Example: Netflix groups users into specific " clusters" based on their viewing speeds, preferred actors, and watch times without using pre-defined age or gender categories.

Association

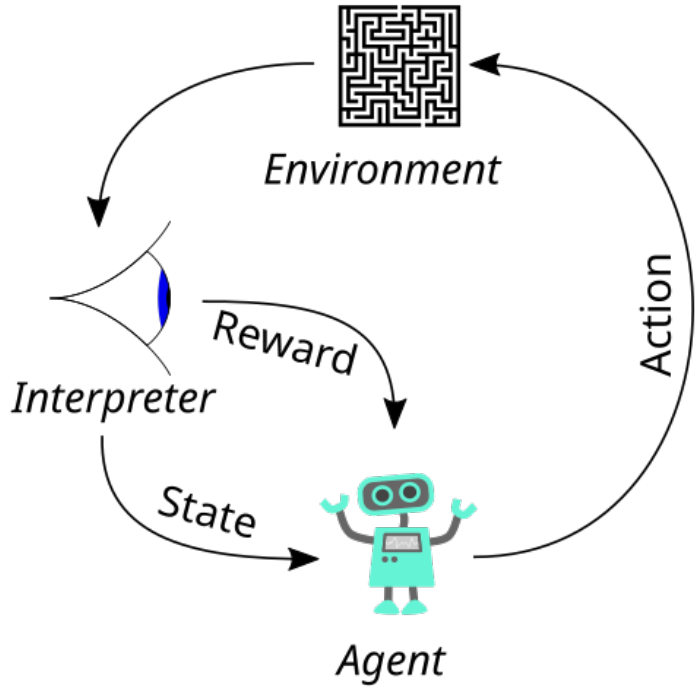
Real-world Example: Amazon's recommendation engine analyzes millions of checkout histories to discover that customers who buy a smartphone also frequently purchase a protective screen guard in the same transaction.

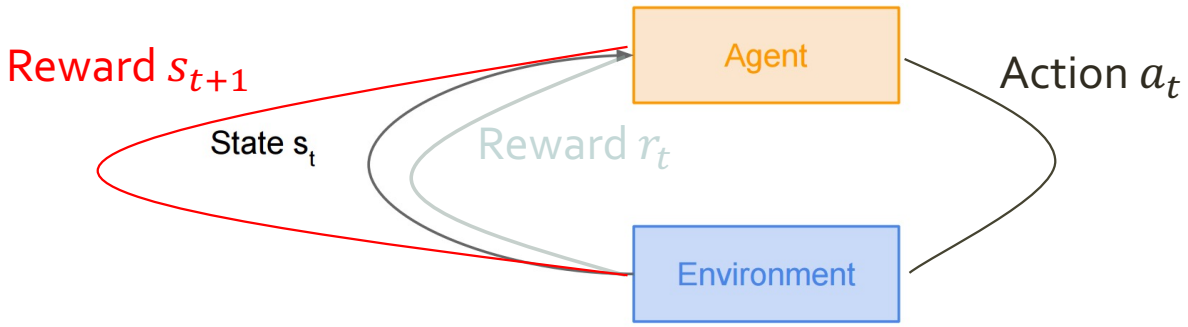
Supervised Vs Unsupervised Learning

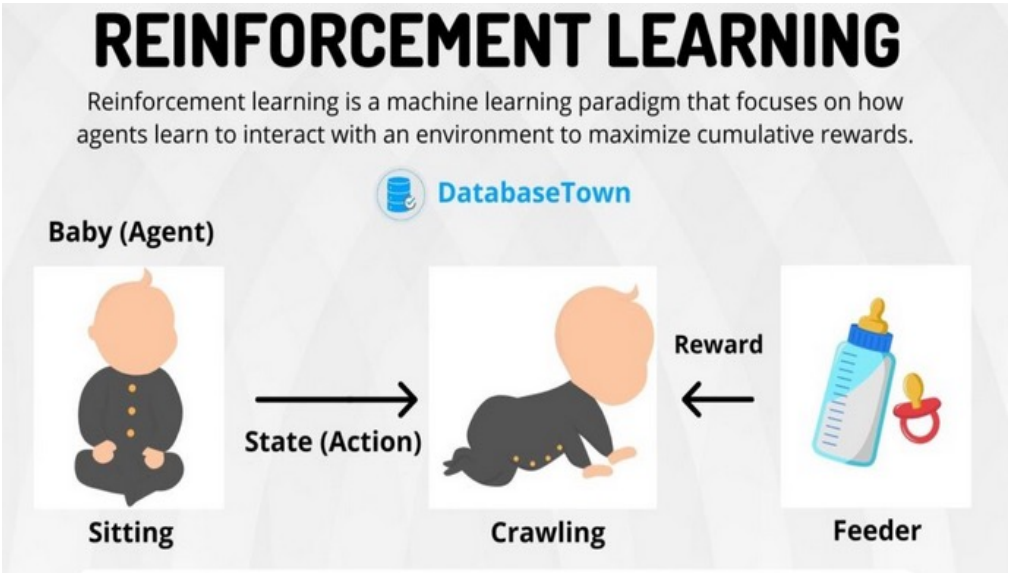


Reinforcement learning (RL)

The task of learning how agents ought to take sequences of actions in an environment in order to maximize cumulative rewards.







Reinforcement learning (RL)

From Interaction to Experience

Every interaction between the agent and the environment produces **experience**.

1.The agent observes a **state** s_t

2.It selects an **action** a_t

3.The environment returns a **reward** r_t

4.The environment moves to the **next state** s_{t+1}

5.This transition becomes part of the agent's **experience**

6. At each time step, the agent generates one transition:

$$\mathcal{D} = \{(s_t, a_t, r_t, s_{t+1})\}_{t=0}^{T}$$

The agent uses this experience to learn

And the question is :

Which *actions lead to better long–term outcomes?*

Markov Decision Process (MDP)

- A **Markov Decision Process (MDP)** is a mathematical framework used to model decision-making situations where outcomes are partly random and partly under the control of a decision-maker (agent)
- **Markov Property**: the future state depends **only** on the current state and the current action, not on the history of past states or actions.
- It formalizes how an agent makes decisions, receives rewards, and moves from one state to another.

Markov Decision Process (MDP)

An MDP defines a sequential decision-making problem using five components:

$$MDP = (S, A, P, R, \gamma)$$

S: set of possible **states**
A: set of possible **actions**
$P(s_{t+1} s_t, a_t)$: transition model
$R(s_t, a_t, s_{t+1})$: reward function
γ: discount factor for future rewards

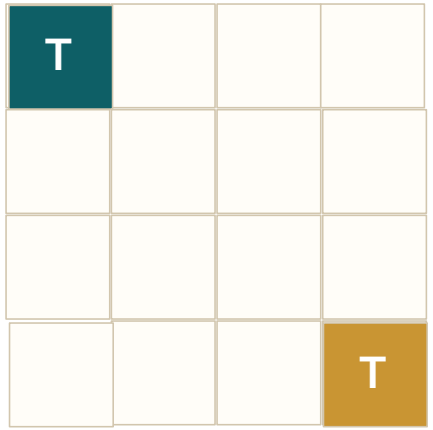
The Markov property
The next state depends only on the current state and action:

$$\mathrm{P}(s_{t+1} | s_t, a_t, s_{t-1}, a_{t-1}, \dots) = P(s_{t+1} | s_t, a_t)$$

Meaning:
The current state must contain all information needed for the next decision.

GridWorld: A Simple MDP

The agent moves in a grid
Each cell is a state
Actions: up, down, left, right
Reward: −1 per move
Goal: reach the target T in as few steps as possible



Actions: ↑ ↓ ← →
Reward: −1 per move
Target T = +10

S = {all grid cells}
A = {↑, ↓, ←, →}
r_t = −1 for each non-terminal move

Random Policy vs Greedy Policy

How actions are selected before and after value estimates become useful

Random policy

Selects each action with equal probability.

$\pi\_random(a|s) = 1 / 4$

The agent does not yet use knowledge about the environment.

Greedy policy with respect to V_k

Selects the action that leads to the best currently estimated next state.

π greedy$(s) = \arg\max_a Q_k(s,a)$
$Q_k(s,a) = r + \gamma V_k(s')$

GridWorld setup

4×4 grid, two terminal states T
Actions: ↑, ↓, ←, →
Reward per move: $r = -1$
Discount: $\gamma = 1$
Initial values: $V_0(s)=0$

At k = 0, all non-terminal states have value 0. A greedy policy cannot yet distinguish good from bad actions.

V_0 values

T	0.0	0.0	0.0
0.0	0.0	0.0	0.0
0.0	0.0	0.0	0.0
0.0	0.0	0.0	T

Random policy

T	↔↕	↔↕	↔↕
↔↕	↔↕	↔↕	↔↕
↔↕	↔↕	↔↕	↔↕
↔↕	↔↕	↔↕	T

Greedy policy after values improve

T	←	←	←↓
↑	↑←	←↓	↓
↑	↑→	→↓	↓
↑→	→	→	T

value estimates guide actions

Policy Evaluation: How V(s) is Updated

Evaluating the expected return under a random policy

Value function under a policy

The value function estimates the expected future reward from each state.

A policy is a rule for selecting actions. It can be stochastic, where it gives probabilities over actions, or deterministic, where it directly selects one action

For the random policy, the next value is the average over all four possible actions.

$$Definition\ (expected\ future\ reward): V_\pi(s) = E_\pi \sum_{\kappa=0}^{\infty} \gamma^k r_{t+k} \mid s_t{=}s$$

Bellman Equation $V_{\kappa+1}(s) = \frac{1}{4}\sum_{\alpha\in A}[r + \gamma V_k(s')]$

k = 0

T	0.0	0.0	0.0
0.0	0.0	0.0	0.0
0.0	0.0	0.0	0.0
0.0	0.0	0.0	T

k = 1

T	-1.0	-1.0	-1.0
-1.0	-1.0	-1.0	-1.0
-1.0	-1.0	-1.0	-1.0
-1.0	-1.0	-1.0	T

Numerical example: state s = (0,1)

At k = 0, all neighboring values are 0.

$V_1(1,1) = ¼\ [\ (-1+0)+(-1+0)+(-1+0)+(-1+0)\ ]$

$V_1(1,1) = -4 / 4 = -1$

Interpretation: after one update, every non-terminal state has value −1 because every move costs −1.

Repeated Updates Propagate Information from the Goal

Values become structured as terminal information spreads through the grid

k = 2

T	-1.7	-2.0	-2.0
-1.7	-2.0	-2.0	-2.0
-2.0	-2.0	-2.0	-1.7
-2.0	-2.0	-1.7	T

k = 3

T	-2.4	-2.9	-3.0
-2.4	-2.9	-3.0	-2.9
-2.9	-3.0	-2.9	-2.4
-3.0	-2.9	-2.4	T

k = 10

T	-6.1	-8.4	-9.0
-6.1	-7.7	-8.4	-8.4
-8.4	-8.4	-7.7	-6.1
-9.0	-8.4	-6.1	T

k → ∞

T	-14	-20	-22
-14	-18	-20	-20
-20	-20	-18	-14
-22	-20	-14	T

States closer to a terminal state become less negative. States far away have lower value because they require more expected steps.

Detailed calculation for $V_2(0,1)$

Using V_1:

$V_2(0,1) = ¼ [(-1+V_1(0,1)) + (-1+V_1(T)) + (-1+V_1(0,2)) + (-1+V_1(1,1))]$

$V_2(0,1) = ¼ [(-1-1) + (-1+0) + (-1-1) + (-1-1)]$

$$V_2(0,1) = (-2-1-2-2)\,/\,4 = -7/4 = -1.75 \approx -1.7$$

Main learning intuition

Repeated updates move information backward from terminal states. The value function gradually tells the agent which regions of the grid are better.

From Values to Greedy Actions

The agent uses current values to choose the best next move

$$Q_k(\mathrm{s,a}) = \mathrm{r} + \gamma\, V_k(\mathrm{s}')$$

$$\pi_{greedy} = \arg max_a\ Q_k(\mathrm{s,a})$$

Example state: s = (0,1), using V_2 and r = −1, γ = 1

Current value estimates V_2

T	-1.7	-2.0	-2.0
-1.7	-2.0	-2.0	-2.0
-2.0	-2.0	-2.0	-1.7
-2.0	-2.0	-1.7	T

Greedy policy induced by V_2

T	←	←	←↓
↑	↑←	←↓	↓
↑	↑→	→↓	↓
↑→	→	→	T

Action-value calculations at s = (0,1)

Left → T: $Q_2 = -1 + V_2(T) = -1 + 0 = -1$
Right → (0,2): $Q_2 = -1 + V_2(0,2) = -1 - 2.0 = -3.0$
Down → (1,1): $Q_2 = -1 + V_2(1,1) = -1 - 2.0 = -3.0$
Up → wall: $Q_2 = -1 + V_2(0,1) = -1 - 1.7 = -2.7$

$$\max Q_2 = -1 \Rightarrow \pi_{\mathrm{greedy}}(0,1) = \leftarrow$$

Interpretation

Because all rewards are negative, the best action is the one that leads to the least negative future value. Greedy action selection converts learned values into behavior.

Without Model - How Repeated Training Leads to Better Decisions

Experience updates values; values guide actions; actions improve over time

Learning cycle

1. Start with unknown values: $V_0(s)=0$
2. Collect or simulate experience: s_t, a_t, r_t, s_{t+1}
3. Update value estimates
4. Improve action selection
5. Repeat until the policy stabilizes

$$V_{k+1}(s) = \frac{1}{4}\sum_a r + \gamma V_k(s')$$

$$\pi(s) = \arg\max_a [r + \gamma V_k(s')]$$

Final learned behavior

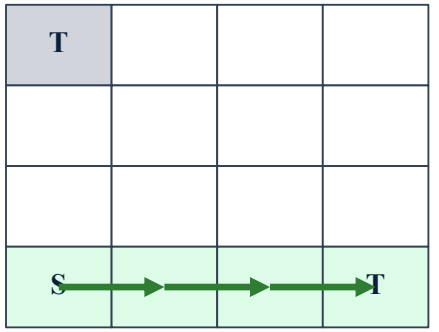
A good path reaches a terminal state in the fewest moves. From the bottom-left start, two shortest paths are possible.

Connection to model-free Q-learning

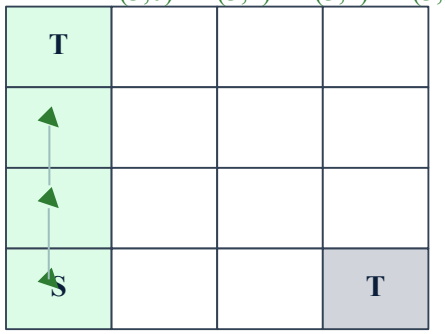
In pure RL, the agent may not know the full transition model. Instead, it learns from sampled experience.

$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha\,[r_t + \gamma\, max_{\alpha'} \;\; Q(s_{t+1}, a'_t) - Q(s_t, a_t)\,]$

The term $r_t + \gamma\, max_{\alpha'} \;\; Q(s_{t+1}, a'_t)$ is the learning target.



Path 1: (3,0) → (3,1) → (3,2) → (3,3)

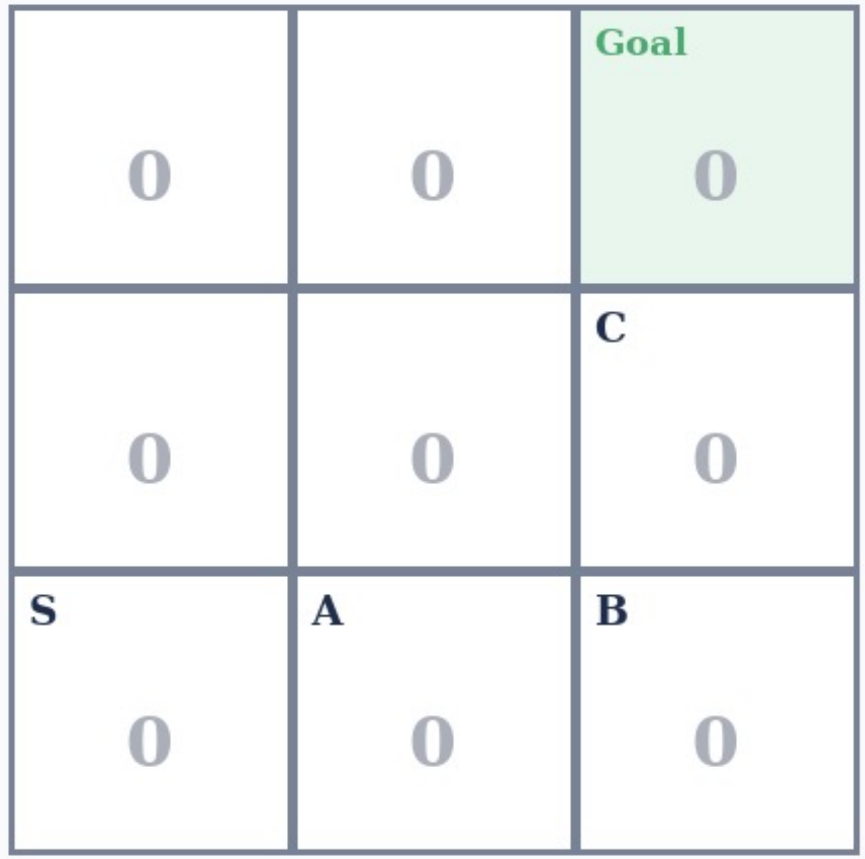


Path 2: (3,0) → (2,0) → (1,0) → (0,0)

Repeated training makes useful actions receive higher values, so the greedy policy eventually follows a shortest path.

GridWorld Q-learning: Information Propagation

Initial Q-values



All Q-values start at 0

Code Demo: Tabular Q-Learning

```
Q = np.zeros((n_states, n_actions))

for episode in range(num_episodes):
    s = env.reset()
    done = False

    while not done:
        if np.random.rand() < epsilon:
            a = env.sample_action()
        else:
            a = np.argmax(Q[s])

        s_next, r, done = env.step(a)

        td_target = r + gamma * np.max(Q[s_next])
        td_error = td_target - Q[s, a]
        Q[s, a] += alpha * td_error

        s = s_next
```

Initialize Q-table
Choose action with ε-greedy
Step the environment
Compute TD target and TD error
Update one Q-table entry

Before training: random paths
After training: shorter paths

$Q[s,a] \leftarrow Q[s,a] + \alpha \cdot$ TD error

Why Tabular RL Does Not Scale

In small GridWorld problems, we can store values explicitly:

$$V(s) \text{ or } Q(s, a)$$

For example:

$$| S | = 16, | A | = 4$$

$$| S | \times | A | = 64$$

But in large scheduling problems:

- The state includes thousands of jobs, machines, remaining operations, and resource availability
- The number of possible partial schedules grows combinatorially
- The set of valid actions changes at every step
- The agent cannot visit every state-action pair many times
- A table cannot generalize to unseen states
- 20x20 size instance where each state include available jobs , machines , bottlenecks etc has $\frac{(nm)!}{m^n} = 10^{501}$

Tabular *RL is useful for understanding, but not enough for large scale scheduling.*

From Tables to Neural Networks

To scale RL, we replace tables with function approximation.

Instead of storing: $Q(s, a)$ we learn: $Q(s, a, \theta)$

where θ are neural network parameters.

The neural network learns patterns from state and action features.

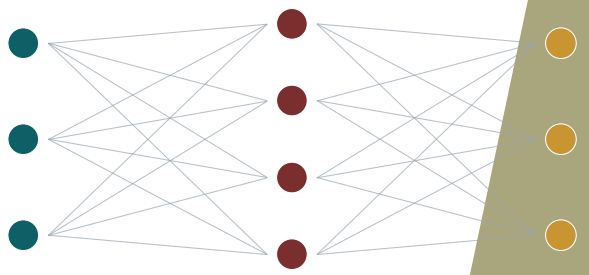
For scheduling, the input may include:

job progress

processing times

remaining work

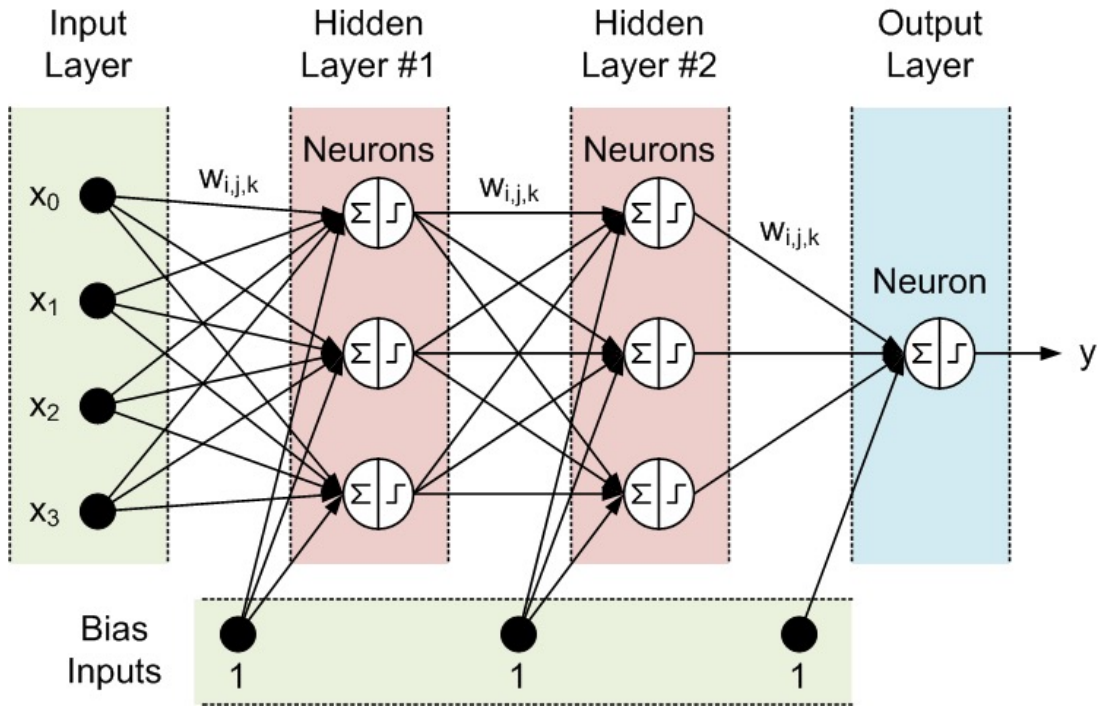
The output can be: $Q(s, a, \theta)$ or action probabilities: $\pi_\theta(a \mid s)$

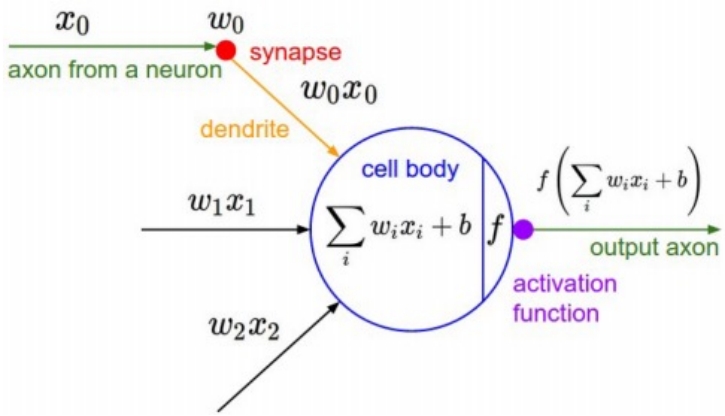


state features → neural network → Q-values

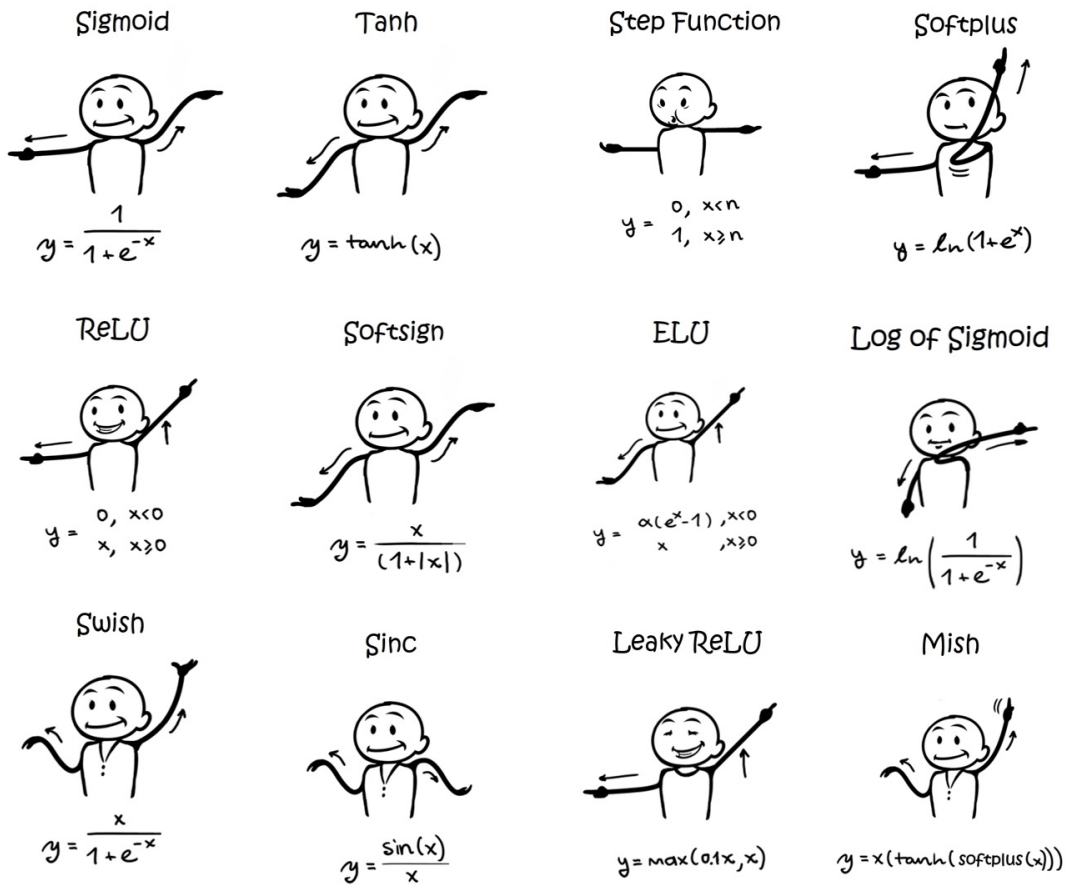
Neural networks allow RL to generalize across similar states and larger instances.

From Tables to Neural Networks





Activation Functions



Deep Reinforcement Learning

Deep RL combines reinforcement learning with neural networks.

The agent still learns from experience: (s_t, a_t, r_t, s_{t+1})
but the learned object is no longer a table.

It can be:

Value-based Deep RL $Q(s, a, \theta)$

The network estimates the value of actions.

Policy-based Deep RL $\pi_\theta(a \mid s)$

The network directly outputs an action policy.

Actor-Critic Deep RL $\pi_\theta(a \mid s), V_\phi(s)$

One network chooses actions, another evaluates states.

Deep RL is the bridge from simple GridWorld learning to complex decision-making problems such as scheduling.

Value-Based vs Policy-Based Reinforcement Learning

There are two main ways to learn decision-making behavior in RL.

Value-Based RL

Learns how good each action is: $Q(s, a)$

Then selects the best action: $a_t = \arg\max_{\alpha} Q(s_t, \theta)$

Examples:

- Q-learning
- Deep Q-Networks
- Value iteration

Policy-Based RL

Learns the policy directly: $\pi_\theta(a \mid s)$

The policy outputs action probabilities: $a_t \sim \pi_\theta(\cdot \mid s_t)$

Examples:

- REINFORCE
- A2C
- PPO

Value-based methods learn action values first.
Policy-based methods learn how to act directly.

How To Find The Best Action?

A neural network produces an output, but learning needs a direction.

$\hat{y} = neural\ network\ prediction$

$L = loss(y, \hat{y})$

$L = (y - \hat{y})^2$

$L = -\sum_i y_i \log(\hat{y}_i)$

The loss function measures how bad the current output is.

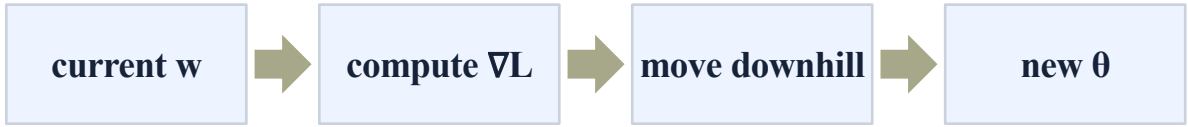
Aggregating all these Losses , we can draw a loss surface

Key points
- The output alone does not tell us how to improve.
- The loss defines what “wrong” means.
- Smaller loss means better performance.
- Training changes weights to reduce the loss.

Gradients: How Do We Improve the Weights?

To reduce the loss, we need to know how the loss changes when each parameter changes.

Gradient descent intuition



dL / dw For all network parameters:

$$\nabla_\theta L = \left[\frac{dL}{dw_1}, \frac{dL}{dw_2}, \ldots, \frac{dL}{dw_n}\right]$$

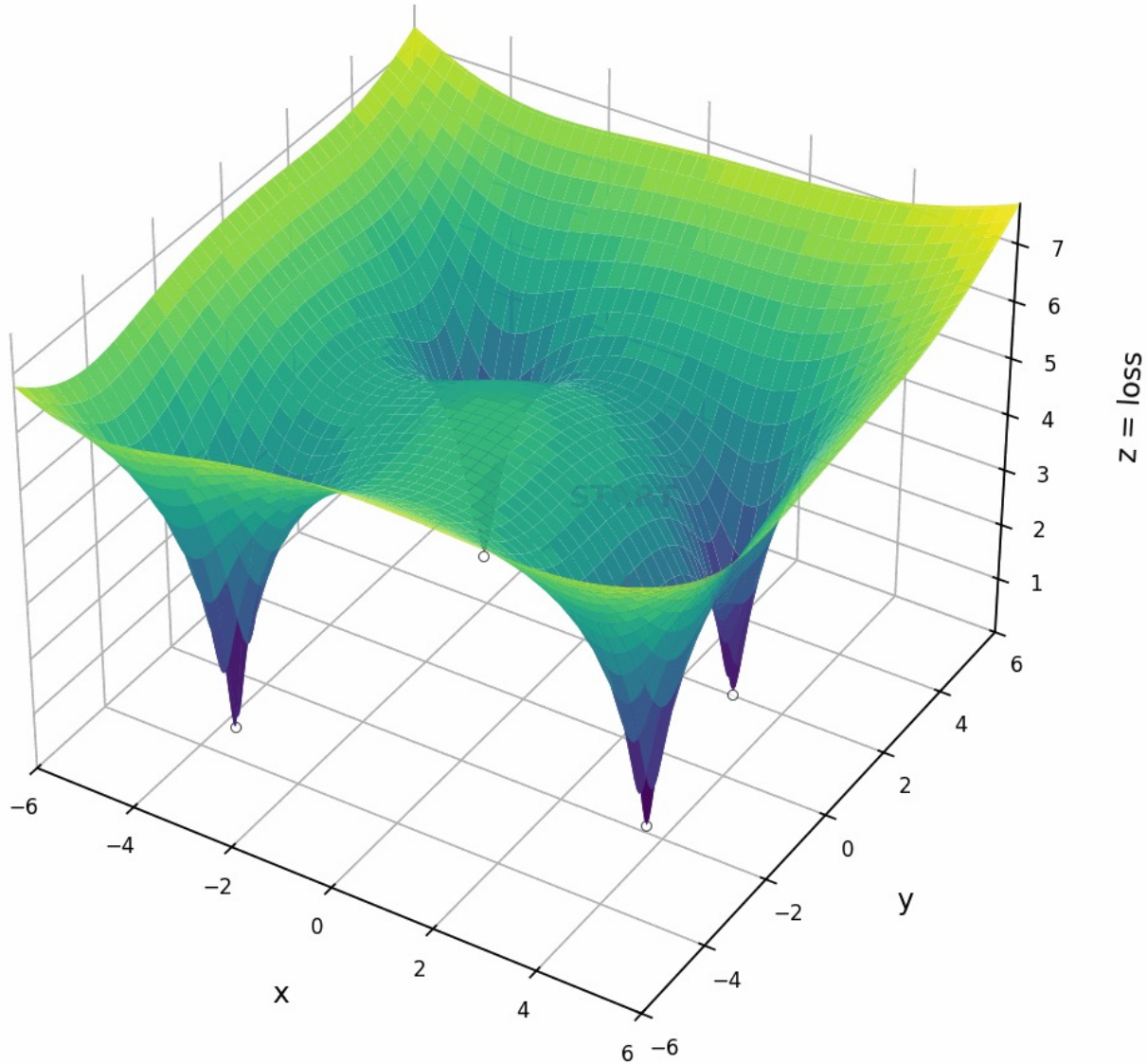
The gradient points toward increasing loss. Gradient descent moves in the opposite direction.

$$w_{new} = w_{old} - \eta \nabla_w L$$

Interpretation
- Positive derivative: increasing the weight increases loss.
- Negative derivative: increasing the weight decreases loss.
- The gradient gives the steepest increase direction.
- We move opposite to reduce the loss.

Loss Surface and Visible Gradient Descent

Surface defined by the loss function



★ **START**

– trajectory

– descent step

● current

Goal: visually show parameters moving from high loss to lower loss.

theta_new = theta_old - eta * grad L(theta)

Backpropagation: Computing the Gradients

Each layer transforms the signal from the previous layer. The final loss depends on all previous weights.

$$z = w^T x + b$$

$$a = f(z)$$

For one weight, the chain rule gives:

$$dL/dw = dL/d\hat{y} \cdot d\hat{y}/da \cdot da/dz \cdot dz/dw$$

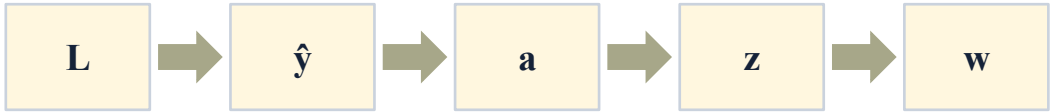
Training loop
- Forward pass: compute prediction.
- Compute loss.
- Backward pass: compute gradients.
- Optimizer step: update weights.

forward → loss → backward → update

Backpropagation: Computing the Gradients

Forward computation

Backward gradients



Backpropagation applies the chain rule efficiently from the output layer back to earlier layers.

From Supervised Loss to Reinforcement Learning

Supervised learning
The target is usually known.

$$\min L(y, \hat{y})$$

label y → **prediction ŷ** → **loss L**

Meaning
- Learn by matching labels.
- The error is directly measured.
- Gradients reduce prediction loss.

Reinforcement learning
There is no correct action label for every state.

$$J(\theta) = E_{\pi_\theta} G_t$$

$$\max J(\theta) \qquad L_{RL}(\theta) = -J(\theta)$$

Policy-gradient idea:

$$\nabla_\theta J(\theta) \approx \nabla_\theta \log \pi_\theta(\alpha_t | s_t) \cdot \text{signal}_t$$

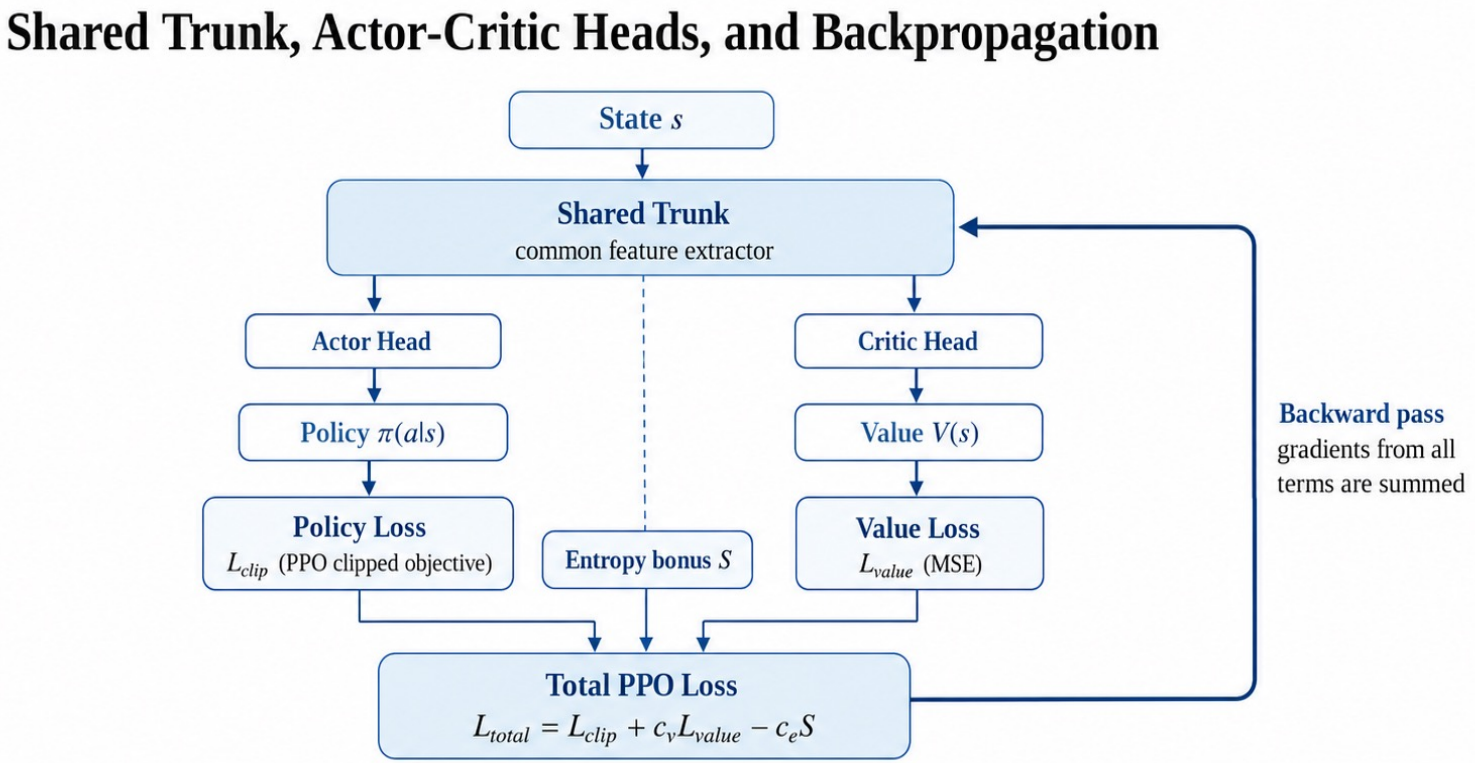
The learning signal can be:

G_t A_t

Connection to algorithms
- REINFORCE uses returns.
- A2C and PPO use advantage estimates.
- The objective comes from reward, not labels.

RL still uses gradients, but the objective comes from reward, not labeled prediction error.

Backpropagation Flow



REINFORCE: Direct Policy Learning

REINFORCE is one of the simplest policy-gradient algorithms.
The policy is a neural network: $\pi_\theta(a \mid s)$
It generates a full trajectory: $\tau = (s_0, a_0, r_0, \ldots, s_T)$

The return from time t is: $G_t = \sum_{k=0}^{T-t} \gamma^k r_{t+k}$

Equivalent loss: $L_{policy} = -\sum_t \log \pi_\theta (a_t \mid s_t) G_t$

The policy is updated using: $\nabla_\theta J(\theta) = \mathbb{E}[\sum_t \nabla_\theta \log \pi_\theta (a_t \mid s_t) G_t]$

Key intuition

- If an action leads to high return, increase its probability.
- If an action leads to low return, decrease its probability.

Limitation

REINFORCE has **high variance**, because it uses full trajectory returns.

From REINFORCE to A2C

REINFORCE uses the raw return: G_t
This can be noisy.

A2C introduces a critic that estimates how good the current state is: $V_\phi(s_t)$

Then we compute the advantage: $A_t = G_t - V_\phi(s_t)$

The advantage answers: Was *this action better or worse than expected?*

Actor loss $L_{actor} = -\log \pi_\theta(a_t \mid s_t) A_t$

Critic loss $L_{critic} = \left(G_t - V_\phi(s_t)\right)^2$

Total A2C loss $L_{Total} = L_{actor} + c_v L_{critic} - c_e H(\pi_\theta)$

State s_t goes to two networks:

$$s_t \rightarrow \begin{cases} \text{Actor:} & \pi_\theta(a \mid s) \\ \text{Critic:} & V_\phi(s) \end{cases}$$

REINFORCE vs A2C: Main Difference

Method	Learning signal	Main idea	Weakness
REINFORCE	G_t	Learn directly from full returns	High variance
A2C	$A_t = G_t - V(s_t)$	Use a critic as a baseline	More components to train

REINFORCE update $L_{REINFORCE} = -\log \pi_\theta (a_t \mid s_t) G_t$

A2C update $L_{A2C} = -\log \pi_\theta (a_t \mid s_t) A_t$

where: $A_t = G_t - V_\phi(s_t)$

A2C is REINFORCE with a learned baseline: The critic reduces variance and makes learning more stable.

REINFORCE: Policy $\rightarrow$ *Action* $\rightarrow$ *Return*
A2C: Policy + *Critic* $\rightarrow$ *Advantage*

From A2C to Actor-Critic PPO

A2C uses an actor and a critic: $\pi_\theta(a \mid s), V_\phi(s)$

But policy-gradient updates can sometimes be too large.

A large update may:

- destroy useful behavior
- overreact to noisy trajectories
- make training unstable
- reduce performance suddenly

Proximal Policy Optimization (PPO) keeps the actor-critic structure but makes policy updates more conservative.

It compares the new policy with the old policy: $r_t(\theta) = \frac{\pi_{\theta,new}(a_t|s_t)}{\pi_{\theta,\,old}(a_t|s_t)}$

Main idea: Improve *the policy, but do not move too far from the old policy.*

PPO = Actor-Critic learning with controlled policy updates.

A2C : State ⟶ Actor + Critic ⟶ update

PPO : State ⟶ Actor + Critic ⟶ ratio $r_t(\theta)$⟶ clipped update

PPO Probability Ratio

PPO compares the new policy with the old policy using the probability ratio: $r_t(\theta) = \frac{\pi_{\theta,new}(a_t|s_t)}{\pi_{\theta,\,old}(a_t|s_t)}$

This ratio measures how much the probability of the selected action changed.

- If $r_t(\theta) = 1$, the action probability did not change.
- If $r_t(\theta) > 1$, the new policy gives higher probability to the action.
- If $r_t(\theta) < 1$, the new policy gives lower probability to the action.

PPO Clipping: Preventing Too Large Updates

Proximal Policy Optimization Algorithms

John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, Oleg Klimov

The main objective we propose is the following:

$$L^{CLIP}(\theta) = \hat{\mathbb{E}}_t\left[\min(r_t(\theta)\hat{A}_t, \text{clip}(r_t(\theta), 1-\epsilon, 1+\epsilon)\hat{A}_t)\right] \tag{7}$$

where epsilon is a hyperparameter, say, $\epsilon = 0.2$. The motivation for this objective is as follows. The first term inside the min is L^{CPI}. The second term, $\text{clip}(r_t(\theta), 1-\epsilon, 1+\epsilon)\hat{A}_t$, modifies the surrogate objective by clipping the probability ratio, which removes the incentive for moving r_t outside of the interval $[1-\epsilon, 1+\epsilon]$. Finally, we take the minimum of the clipped and unclipped objective, so the final objective is a lower bound (i.e., a pessimistic bound) on the unclipped objective. With this scheme, we only ignore the change in probability ratio when it would make the objective improve, and we include it when it makes the objective worse. Note that $L^{CLIP}(\theta) = L^{CPI}(\theta)$ to first order around θ_{old} (i.e., where $r = 1$), however, they become different as θ moves away from θ_{old}. Figure 1 plots a single term (i.e., a single t) in L^{CLIP}; note that the probability ratio r is clipped at $1-\epsilon$ or $1+\epsilon$ depending on whether the advantage is positive or negative.

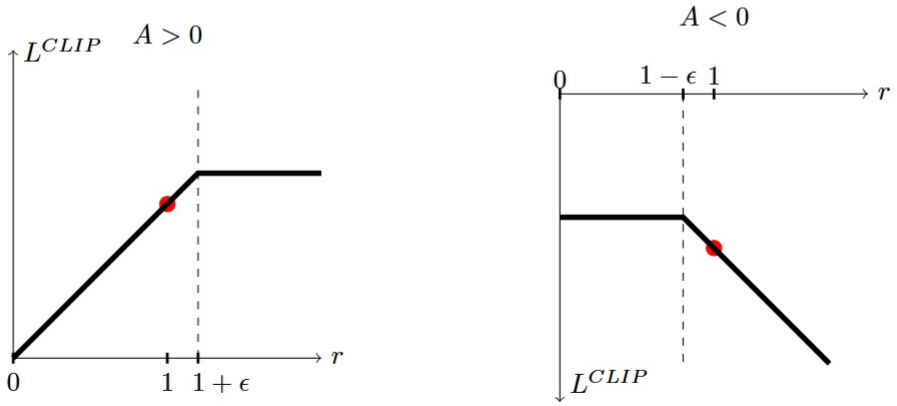


Figure 1: Plots showing one term (i.e., a single timestep) of the surrogate function L^{CLIP} as a function of the probability ratio r, for positive advantages (left) and negative advantages (right). The red circle on each plot shows the starting point for the optimization, i.e., $r = 1$. Note that L^{CLIP} sums many of these terms.

PPO Clipping: Preventing Too Large Updates

A large probability ratio may produce an unstable update.

PPO clips the ratio inside a safe interval: $1 - \epsilon \leq r_t(\theta) \leq 1 + \epsilon$

Usually: $\epsilon = 0.2$

So the safe range is: $0.8 \leq r_t(\theta) \leq 1.2$

The clipped ratio is: $\text{clip}(r_t, (\theta), 1 - \epsilon 1 + \epsilon)$

Example

Suppose: $\pi_{\theta,\, old}(a_t \mid s_t) = 0.2$ and $\pi_{\theta,new}(a_t \mid s_t) = 0.3$

Then: $r_t(\theta) = \frac{0.30}{0.20} = 1.5$

The new policy increased the probability of that action by 50%.

then: $\text{clip}(1.5,0.8,1.2) = 1.2$

PPO Actor Objective

The actor is trained using the clipped PPO objective:

$$L^{CLIP}(\theta) = \mathbb{E}_t[\min(r_t,(\theta)A_t\, clip(r_t,(\theta), 1-\epsilon 1+\epsilon)A_t)]$$

Where: $A_t = G_t - V_\phi(s_t)$

The advantage A_t tells whether the action was better or worse than expected.

- If $A_t > 0$ we want to increase the probability of the action.
- If $A_t < 0$ we want to decrease the probability of the action.

But PPO prevents the update from becoming too large.

Actor loss for implementation

Since we minimize losses in neural network training: $L_{actor} = -L^{CLIP}(\theta)$

Critic Loss: Learning the Value Function

The critic estimates the expected return from the current state: $V_\phi(s_t)$

The target is the discounted return: $G_t = \sum_{k=0}^{T-t} \gamma^k \, r_{t+k}$

The critic loss is: $L_{critic} = \left(V_\phi(s_t) - G_t\right)^2$

The critic learns to answer: How *good is this state?*

If the prediction is wrong, the squared error updates the critic.

Why the critic matters

The critic is used to compute the advantage: $A_t = G_t - V_\phi(s_t)$

A better critic gives a better advantage estimate.

Entropy Bonus and Total PPO Loss

Entropy measures how uncertain the policy is: $H(\pi_\theta) = -\sum_a \pi_\theta\,(a \mid s_t) \log \pi_\theta\,(a \mid s_t)$
High entropy means the policy explores more.
Low entropy means the policy is more deterministic.

Why entropy is useful

- It encourages exploration.
- It prevents the policy from becoming too confident too early.
- It helps avoid premature convergence to poor actions.

Total PPO loss

The total loss combines: $L_{total} = L_{actor} + c_v L_{critic} - c_e H(\pi_\theta)$
where: c_v controls the critic loss weight, and: c_e controls the entropy bonus.

From PPO to Job Shop Scheduling

Until now, the agent was selecting actions in simple environments such as GridWorld.

In Job Shop Scheduling, the agent constructs a schedule step by step.

- At each decision step, the state is the current partial schedule.

$$s_t = current\ partial\ schedule$$

- The action is the next job to dispatch.

$$a_t = next\ job\ to\ dispatch$$

- After the action, the environment returns a reward and the next state.

$$s_t \rightarrow a_t \rightarrow r_t, s_{t+1}$$

A complete schedule corresponds to one full RL episode.

Job Shop Scheduling Problem (JSSP)

A JSSP instance contains jobs, machines, and ordered operations

- Set of jobs: $J = \{1, 2, \ldots, n\}$
- Set of machines: M = {1, 2, …, m}
- Each job contains ordered operations: $O(j,1) \rightarrow O(j,2) \rightarrow \ldots \rightarrow O(j,m)$
- Each operation has a required machine and a processing time:
- $$O(j,k) = (M(j,k), p(j,k))$$

The goal is to minimize the makespan $C_{max} = \max C_j$ and $\min C_{max}$

- Each job follows a fixed operation order.
- Each machine processes one operation at a time.
- Operations cannot be interrupted.
- The objective is global, not local.

MATRIX 1: PROCESSING TIMES

JOBS	OPERATIONS o_1	o_2	o_3	o_4
J_1	$p_{1,1}$	$p_{2,1}$	$p_{3,1}$	$p_{4,1}$
J_2	$p_{1,2}$	$p_{2,2}$	$p_{3,2}$	$p_{4,2}$
J_3	$p_{1,3}$	$p_{2,3}$	$p_{3,3}$	$p_{4,3}$

MATRIX 2: MACHINE ORDERINGS

JOBS	OPERATIONS o_1	o_2	o_3	o_4
J_1	m_1	m_2	m_3	m_4
J_2	m_5	m_6	m_7	m_{t_8}
J_3	m_9	m_{10}	m_{11}	m_{12}

JSSP as a Markov Decision Process

Once JSSP is written as an MDP, PPO can learn a scheduling policy.

The PPO agent needs a fixed-size vector representation of the current schedule

A possible state vector combines job, machine, and global features $s_t = [x_1, \dots, x_2, z_1, \dots, z_m, g_t]$

Job feature vector: x_j = features of job j
Machine feature vector: z_q = features of machine q
Global feature vector: g_t = global schedule features

Useful job features:

- ✓ current operation index
- ✓ remaining operations
- ✓ remaining processing time
- ✓ job availability time
- ✓ next processing time
- ✓ next required machine

Useful machine/global features:

- ✓ machine availability time
- ✓ machine load
- ✓ current partial makespan
- ✓ fraction of completed operations

The state vector is the input to the PPO actor and critic.

Action Space and Valid Action Masking

At each step, the agent selects one job to dispatch $\alpha_t \in \{0, 1, \ldots, n-1\}$

A job is valid only if it still has unscheduled operations

$$A_{valid}(s_t) = \{j : operation\ index[j] < Final\ operation + 1\}$$

Invalid actions must receive zero probability $\pi_\theta(a|s_t) = 0,\ if\ a \notin A_{valid}(s_t)$

In practice, invalid logits are masked before softmax $logits[a] = -10^9$

Then the policy is computed from the masked logits $\pi_\theta(a|s_t) = softmax(masked\ logits)$

Actor outputs one logit per job.
Finished jobs are invalid.
Policy samples only feasible actions.

J_1 ✓ J_2 ✓ J_3 finished J_4 ✓

$$\pi_\theta(j_3) = 0$$

Action masking keeps the learned policy feasible.

Reward and Episode Definition

An episode starts with an empty schedule and ends when all operations have been scheduled.

Flag done = all operations scheduled

The final objective is the makespan $C_{max} = \max C_j$

Sparse reward gives feedback only at the end. $r_t = \begin{cases} 0, & \textit{if schedule is not complete} \\ -C_{max}, & \textit{if schedule is complete} \end{cases}$

A shaped reward gives feedback after each dispatching decision $r_t = -\Delta C_{max,t} - \lambda_{idle} \Delta\, idle_t$

where: $\Delta C_{max,t} = C_{\max,t+1} - C_{max,t}$

$\Delta idle_t = max(0, start\ job\ (j,k) - machine\ available[M(j,k)])$

Sparse reward is aligned with the final objective.
Shaped reward provides more frequent feedback.
Reward design affects PPO training stability.

Reward timelines

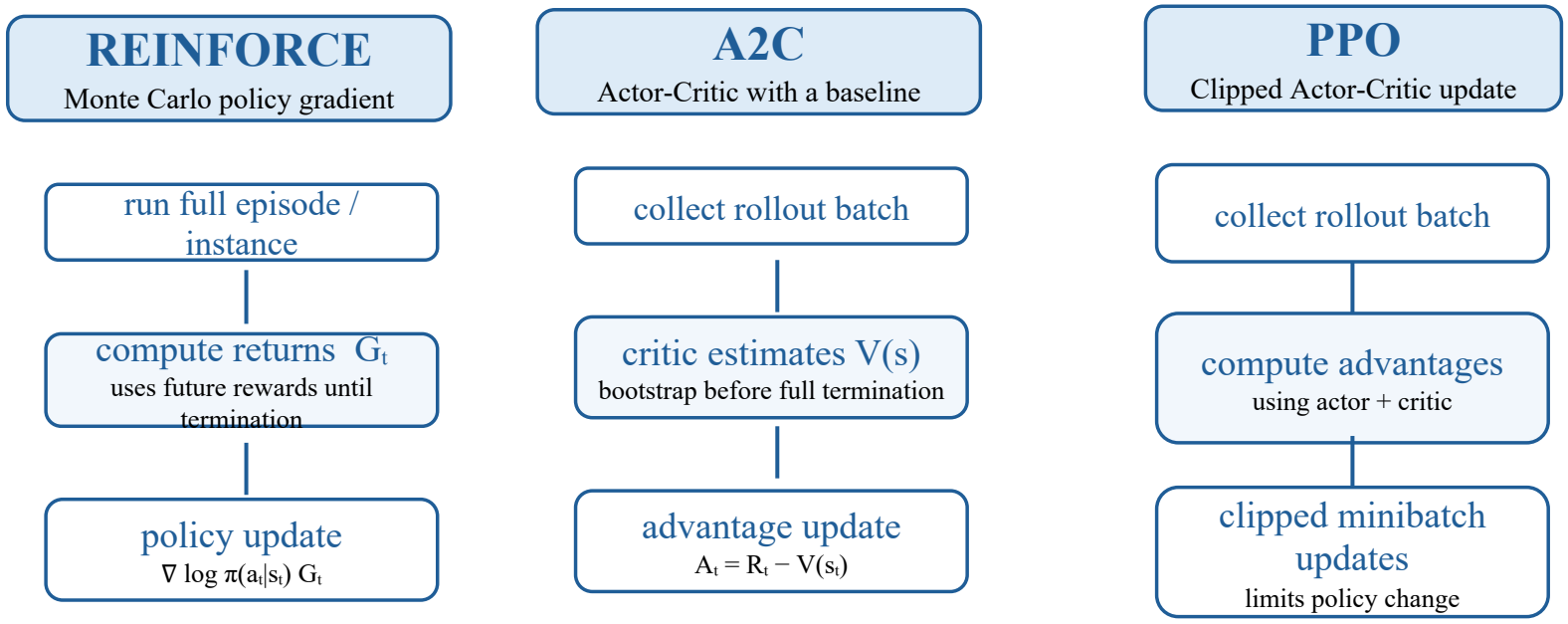
$0, 0, 0, \ldots, -C_{max}$

$r_1 \ldots, r_T$

The reward tells the PPO agent what kind of schedules are good.

REINFORCE vs A2C vs PPO: When Does the Policy Update?

The difference is mainly the learning signal and update rule, not whether the method is tied to one instance size.



Usually high variance. It can average several completed episodes before one update.

Updates can use batches of steps from one or many instances. The critic reduces variance.

More stable than vanilla policy gradient: several minibatch epochs on the same rollout, with clipping.

- Corrected intuition: REINFORCE typically waits for complete trajectories because it uses Monte Carlo returns.
- A2C and PPO are still on-policy, but they can update from rollout batches of steps using value estimates and advantages.
- They are not automatically instance-agnostic; generalization comes from the state representation and shared neural policy.

JSSP Environment Overview

The environment is responsible for constructing feasible schedules.
It receives an action from the PPO agent and updates the partial schedule.

Required environment python methods:

- `reset()`
- `get_available_actions()`
- `get_state()`
- `step(action)`
- `is_done()`
- `compute_makespan()`

JSSP Environment Overview

- The `reset()` function initializes a new episode.
  At the beginning, no operation has been scheduled.
  For each job, the next operation index starts from zero.

- get_available_actions() defines the feasible action set.
  At each decision step, the agent can only select valid jobs.
  A job is valid if it still has at least one unscheduled operation.

- The get_state() function converts the current partial schedule into a vector.
  This vector is the input to the PPO actor and critic.

- The actor outputs one logit per possible job.
- Masking makes the PPO policy respect JSSP feasibility.

- The step(action) function applies one dispatching decision.

Full Environment Interaction Loop

Training data is generated by repeatedly interacting with the environment.
At the beginning of each episode: $s_0 = reset()$

At each step, the actor selects an action: $a_t \sim \pi_\theta(.|s_t)$
The environment applies the action: $s_t, r_t, done = step(a_t)$
The transition is stored: $(s_t, a_t, r_t, s_{t+1}, done)$
The episode stops when: done = true

- Start from an empty schedule.
- Select valid jobs using the policy.
- Schedule one operation at each step.
- Store transitions for PPO training.
- End when all operations are scheduled.

Why State Representation Matters

The PPO agent does not see the schedule directly. It receives a numerical state vector that must summarize the current scheduling situation.

State and networks

s_t = *state vector at decision step t*

$\pi_\theta\ (a_t|s_t)$

$V_\varphi(s_t)$

The actor uses the state to choose the next job.
The critic uses the state to estimate expected return.
Missing information can lead to poor decisions.
Good features make learning easier and more stable.

The state representation is the bridge between the scheduling problem and the PPO neural network.

Per-Job Features

For each job, we create a vector that describes its current status.

Job feature vector

$$x_j = features\ of\ job\ j$$
$$x_j = [op\ idx\ j, rem\ ops j, rem\ work\ j, job\ avail\ j, next\ p_j, next\ m_j, active\ j]$$

Meaning of features

Op idx j: current operation index
rem ops j: remaining number of operations
rem work j: total remaining processing time
job avail j: earliest time the job can continue
$next\ p_j$: processing time of the next operation
$next\ m_j$: machine required by the next operation
$active\ j$: 1 if not finished, 0 otherwise

Example job table

Job	op	rem	avail	next p
J1	2	6	8	3
J2	1	9	5	2
J3	0	12	0	4

Per-job features help the actor compare which job should be dispatched next.

Machine Features

Machine features describe the current status of the resources.

Machine feature vector

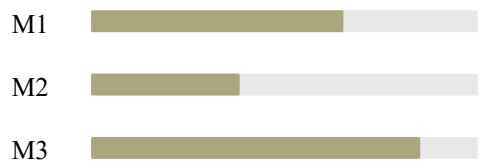
$$z_q = features\ of\ machine\ q$$
$$z_q = [machine\ avail\ q,\ machine\ load\ q,\ rem\ machine\ work\ q,\ util\ q]$$

Meaning of features

Machine avail q: earliest time machine q becomes available
Machine load q: total work already assigned to machine q
Rem machine work q: remaining work requiring machine q
Util q: utilization estimate of machine q

Machine features identify bottlenecks.
They help the policy avoid poor machine usage.
They connect job decisions with resource availability.

Machine availability



Machine features help the PPO agent understand resource constraints.

Global Schedule Features

Global features summarize the whole partial schedule and give the network context.

Global feature vector

$$g_t = \ global\ schedule\ features$$
$$g_t = [C_{max,t}, scheduled\ ops\ t, completion\ ratio\ t, avg\ machine\ avail\ t, \max machine\ avail\ t]$$

Meaning of features

$C_{max,t}$: current partial makespan
scheduled ops t: number of scheduled operations
completion_ratio_t: fraction of completed operations
avg machine_avail t: average machine availability time
max machine avail t: maximum machine availability time

Schedule progress

completion_ratio_t

$C_{max,t}$ current partial makespan

Global features tell the agent how far the schedule has progressed.

Feature Normalization

Raw scheduling values can have very different scales. Processing times, remaining work and machine availability should be normalized.

Examples of normalization

$$p_{norm} = p(j,k) \,/\, p_{max}$$

$$machine\ avail_{norm}\ q = machine\ available\ q \,/\, workLoad$$

Normalization improves neural network training.

It prevents large time values from dominating.

It makes training more stable across different instance sizes.

It helps PPO generalize better.

Raw values	Normalized
5, 80, 1200	0.04, 0.32, 0.91

Normalized features make PPO training more stable and comparable across instances.

Padding up to 20×20 Instances

The assignment uses instances up to 20 jobs and 20 machines. To use one fixed MLP input size, smaller instances can be padded.

Maximum size

Jobs $n = 20$
Machines $m = 20$

$$s_t = [x_1, \ldots, x_{20}, z_1, \ldots, z_{20}, g_t]$$

Padding rules

$$x_j = 0, if\ j > n$$

$$z_q = 0, if\ q > m$$

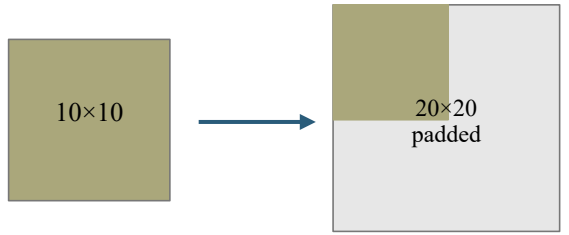
Active $j = 0$*, if job* j *is padded or finished*

Padding creates a fixed input dimension.

Smaller instances can use the same network.

Masks prevent padded jobs from being selected.

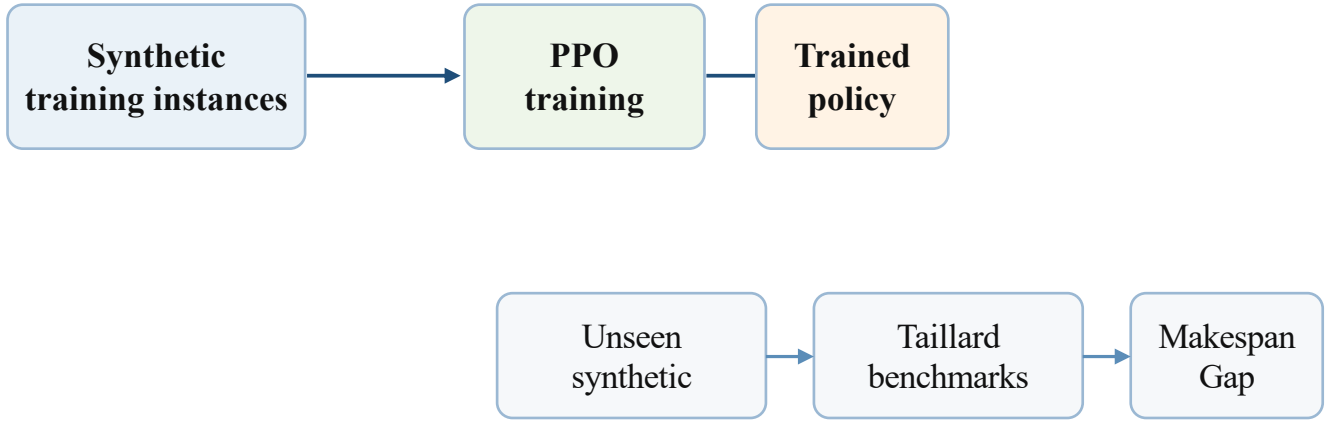
This allows training on mixed-size instances.



Padding allows one PPO model to handle JSSP instances up to 20×20.

Training and Evaluation Overview

The PPO agent is trained on synthetic JSSP instances and evaluated on unseen synthetic and Taillard benchmark instances.



Training teaches the policy, evaluation tests whether the learned policy generalizes.

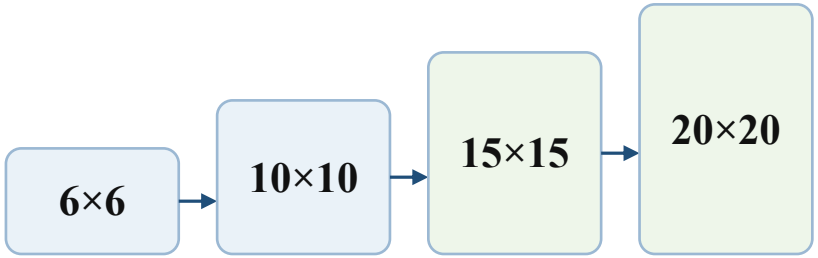
Synthetic Training Instances

Curriculum idea:
learn easier schedules first, then scale up.

Start from smaller instances to stabilize learning.

Progressively increase the number of jobs and machines.

Alternatively, train one model on mixed-size instances.



Synthetic data supports controlled training.

Makespan and Optimality Gap

The primary scheduling metric is **makespan**. If a best-known solution is available, compute the optimality gap.

Makespan

$$C_{max} = \max C_j$$

Optimality gap

$$Gap(\%) = \frac{C_{max}^{method} - C_{max}^{BKS}}{C_{max}^{BKS}} \times 100$$

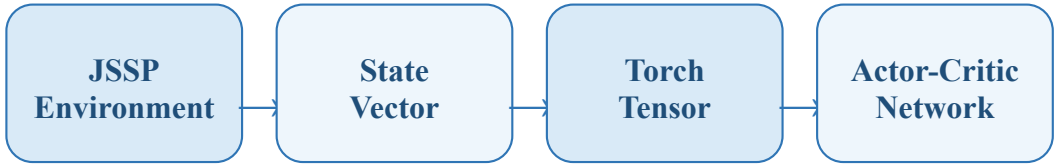
Lower makespan means a better schedule.

Gap compares the method against a reference best-known solution.

From State Vector to Neural Network

The environment returns a numerical state vector.

This vector becomes the input to the actor-critic network.



Code

```
state = env.get_state()
state = torch.tensor(state, dtype=torch.float32)
state = state.unsqueeze(0)
```

```
# typical tensor shape
state.shape == (1, state_dim)
```

The environment produces the state.
The state is converted to a PyTorch tensor.
The neural network receives one state vector.
The actor and critic use the same input.

The input dimension must match the state representation designed for the assignment.

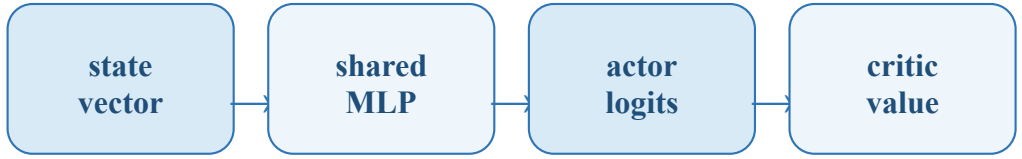
Actor-Critic Network in PyTorch

The PPO agent uses an actor-critic neural network.

The actor outputs one logit per job.
The critic outputs one scalar value.

Network structure

```
class ActorCritic(nn.Module):
    def __init__(self, state_dim, max_jobs,
hidden_dim=128):
        super().__init__()

        self.shared = nn.Sequential(
            nn.Linear(state_dim, hidden_dim),
            nn.ReLU(),
            nn.Linear(hidden_dim, hidden_dim),
            nn.ReLU()
        )

        self.actor = nn.Linear(hidden_dim, max_jobs)
        self.critic = nn.Linear(hidden_dim, 1)
```

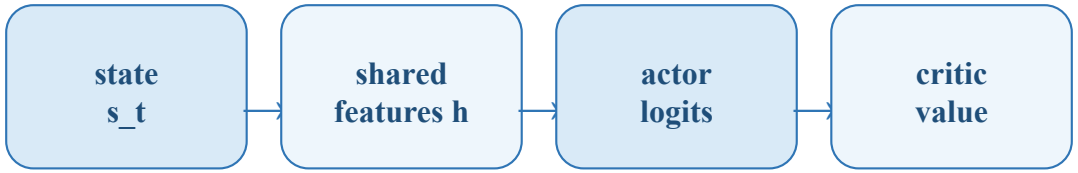


Actor output size = maximum number of jobs
Critic output size = one scalar value estimate

Forward Pass: Actor and Critic Outputs

The forward pass computes the actor and critic outputs from the same state representation.

```
def forward(self, state):
    h = self.shared(state)

    logits = self.actor(h)
    value = self.critic(h)

    return logits, value
```



Shared layers extract features.
Actor head produces job logits.
Critic head predicts state value.
PPO uses both outputs during training.

The forward pass transforms the state into action scores and a value estimate.

Action Masking and Sampling

The actor outputs logits for all jobs.

Invalid jobs must be masked before softmax.

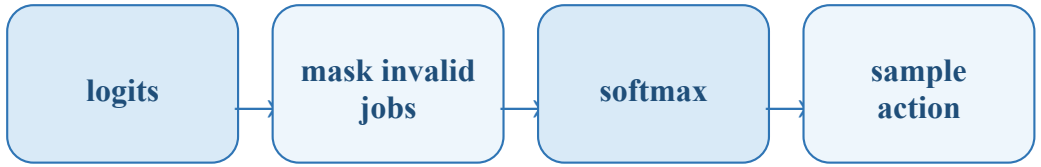
Code

```
logits, value = model(state)

valid_actions = env.get_available_actions()
mask = torch.ones_like(logits, dtype=torch.bool)
mask[:, valid_actions] = False

logits = logits.masked_fill(mask, -1e9)
probs = torch.softmax(logits, dim=-1)

dist = torch.distributions.Categorical(probs)
action = dist.sample()
log_prob = dist.log_prob(action)
```

Equation

$$\pi_\theta(\alpha_t|s_t) = softmax(masked\_logits)$$

Finished jobs receive probability zero.
The policy samples only feasible actions.
The log probability is stored for PPO.
The value estimate is stored for the critic loss.



Loss, Backpropagation, and Gradients

After collecting transitions, PPO computes the total loss.

```
actor_loss = -torch.min(
    ratio * advantage,
    clipped_ratio * advantage
).mean()

critic_loss = (returns - values).pow(2).mean()
entropy_loss = entropy.mean()

loss = actor_loss + value_coef * critic_loss \
       - entropy_coef * entropy_loss
```

Backpropagation computes gradients

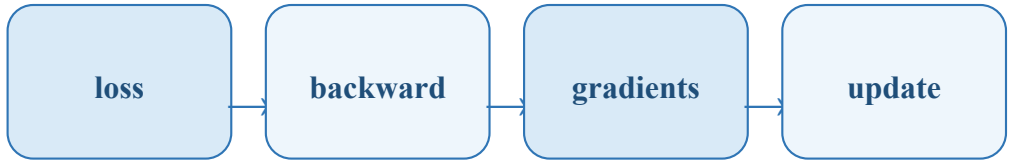
```
loss.backward()
```

Chain Rule

$\partial L / \partial \theta$

$\partial L / \partial \varphi$

Actor loss improves the policy.
Critic loss trains the value function.
Entropy encourages exploration.
Backpropagation computes parameter gradients.



Backpropagation tells the network how each parameter should change.

Adam Optimizer Step

The optimizer updates the neural network parameters using the computed gradients.

Optimizer definition

```
optimizer = torch.optim.Adam(
    model.parameters(),
    lr=3e-4
)
```

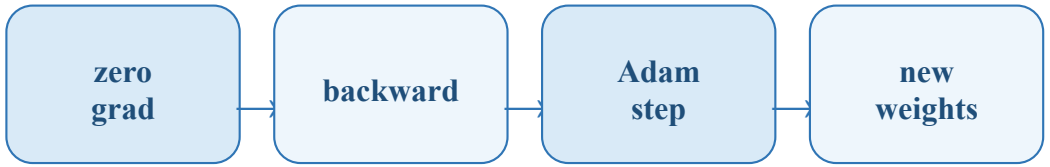
One training update

```
optimizer.zero_grad()
loss.backward()
optimizer.step()
```

Equation

$$\theta \leftarrow \theta - \alpha \cdot Adam(\nabla_\theta L)$$

zero_grad() clears old gradients.
backward() computes new gradients.
step() updates the parameters.
Adam adapts the learning rate per parameter.



Adam is the mechanism that actually changes the actor-critic network weights.

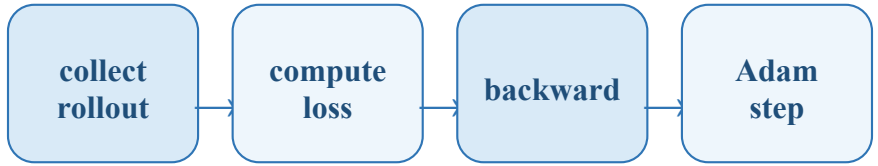
PPO Update

A PPO update combines the forward pass, loss computation, backpropagation, and optimizer step.

```
for epoch in range(num_epochs):

    logits, values = model(states)

    logits = apply_action_mask(logits, valid_masks)
    probs = torch.softmax(logits, dim=-1)

    dist = torch.distributions.Categorical(probs)
    new_log_probs = dist.log_prob(actions)
    entropy = dist.entropy()

    ratio = torch.exp(new_log_probs - old_log_probs)
    clipped_ratio = torch.clamp(ratio, 1 - clip_eps, 1 + clip_eps)

    actor_loss = -torch.min(ratio * advantages, clipped_ratio * advantages).mean()
    critic_loss = (returns - values.squeeze()).pow(2).mean()
    loss = actor_loss + value_coef * critic_loss - entropy_coef * entropy.mean()

    optimizer.zero_grad()
    loss.backward()
    optimizer.step()
```

Implementation pieces

Forward pass: logits and values
Action masking before softmax
New action log probabilities
PPO ratio and clipping
Actor loss and critic loss
Entropy bonus
Backpropagation and Adam step




Reinforcemnt Learning
Thank you