Reinforcement Learning Hands-On

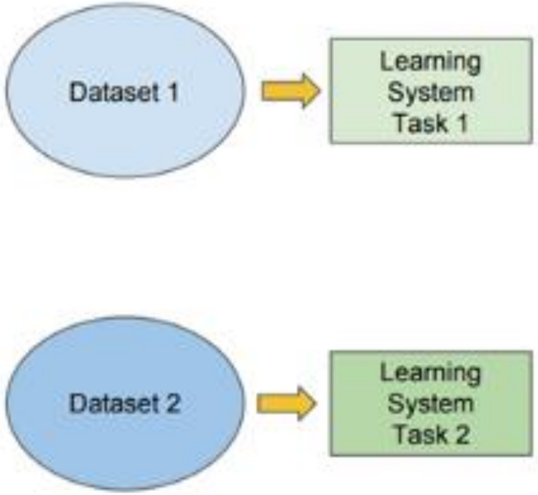
Transfer Learning

Transfer Learning

Traditional ML vs Transfer Learning

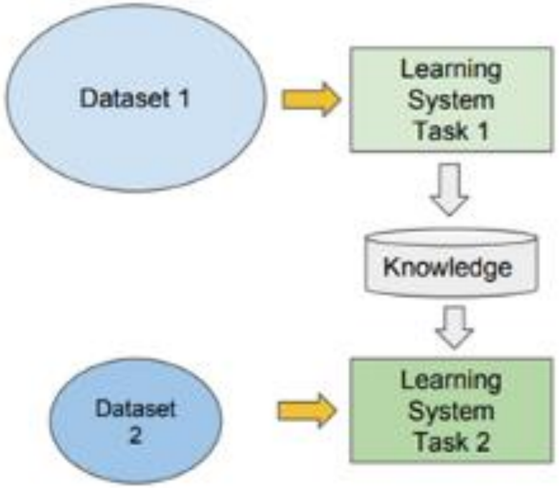
Traditional ML

- Isolated, single task learning:
  - Knowledge is not retained or accumulated. Learning is performed w.o. considering past learned knowledge in other tasks



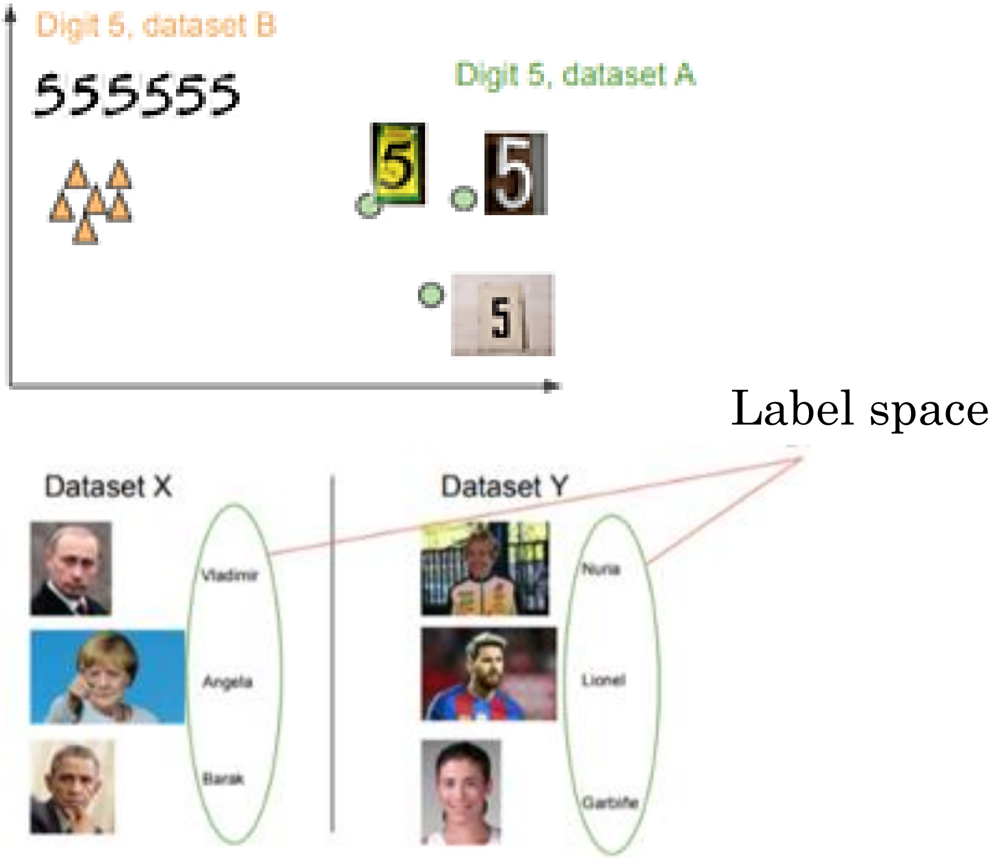
Transfer Learning

- Learning of a new tasks relies on the previous learned tasks:
  - Learning process can be faster, more accurate and/or need less training data

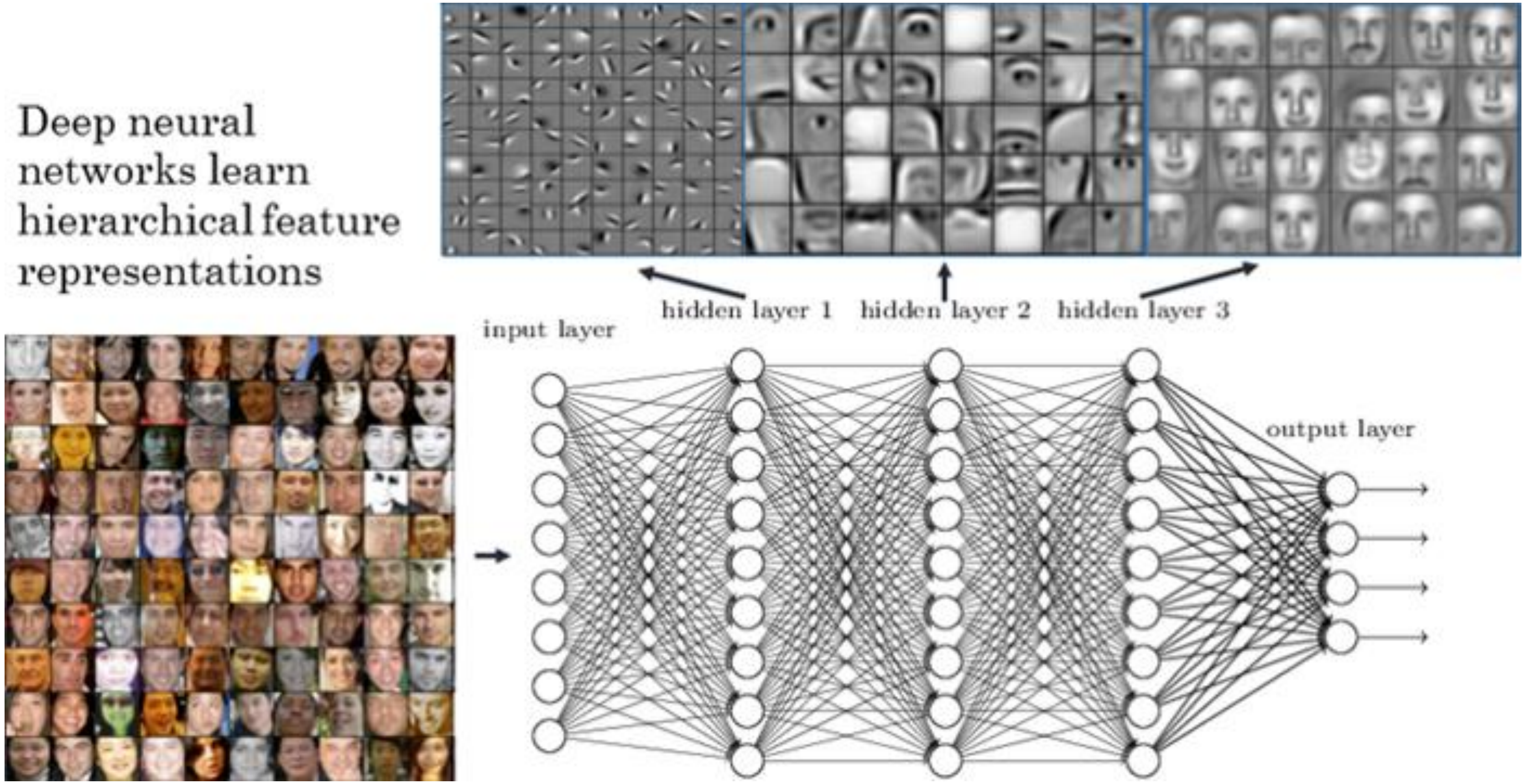


Transfer Learning

- **What to transfer**
  - Source-specific
  - Common domain source-target
- **When to transfer**
  - Negative transfer
- **How to transfer**
  - Freeze Layers
  - Fine-tuning Layers



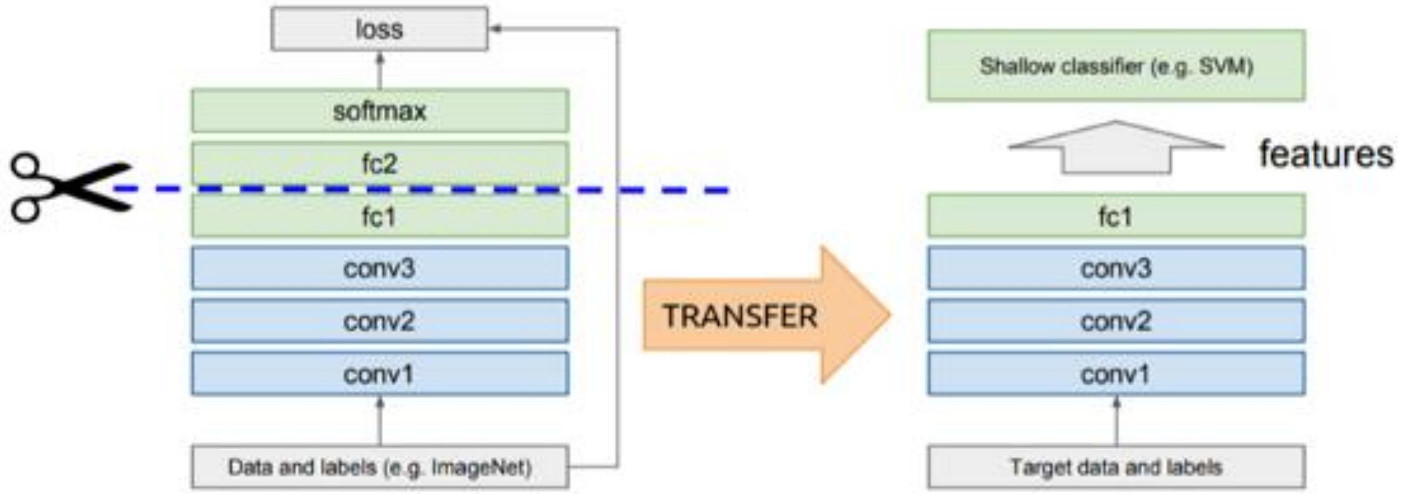
Transfer Learning



Transfer Learning

Idea: use outputs of one or more layers of a network trained on a different task as generic feature detectors. Train a new shallow model on these features.

Assumes that $D_S = D_T$



Transfer Learning

Freeze or fine-tune?

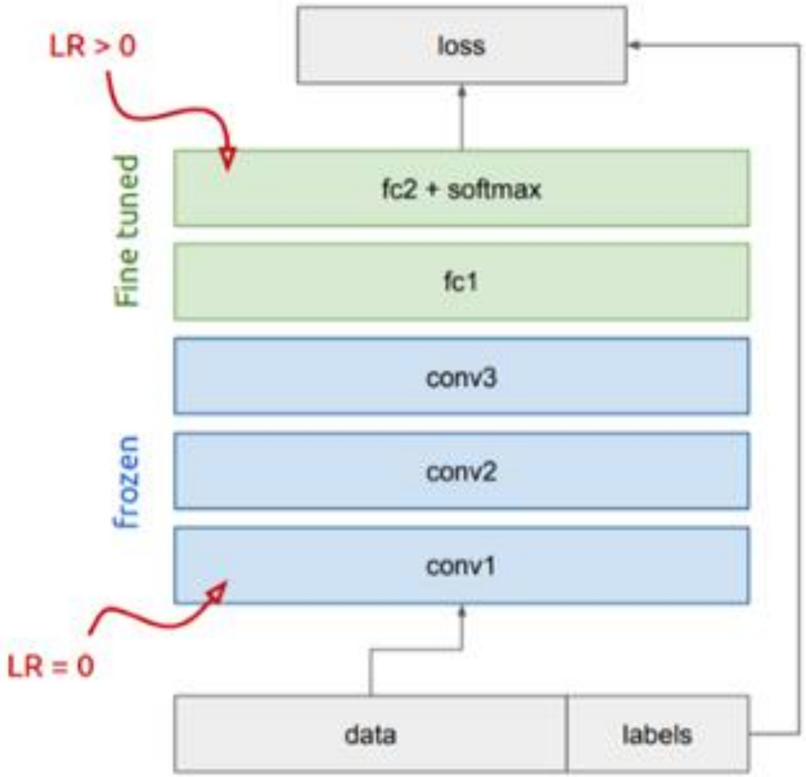
Bottom *n* layers can be frozen or fine tuned.

- **Frozen**: not updated during backprop
- **Fine-tuned**: updated during backprop

Which to do depends on target task:

- **Freeze**: target task labels are scarce, and we want to avoid overfitting
- **Fine-tune**: target task labels are more plentiful

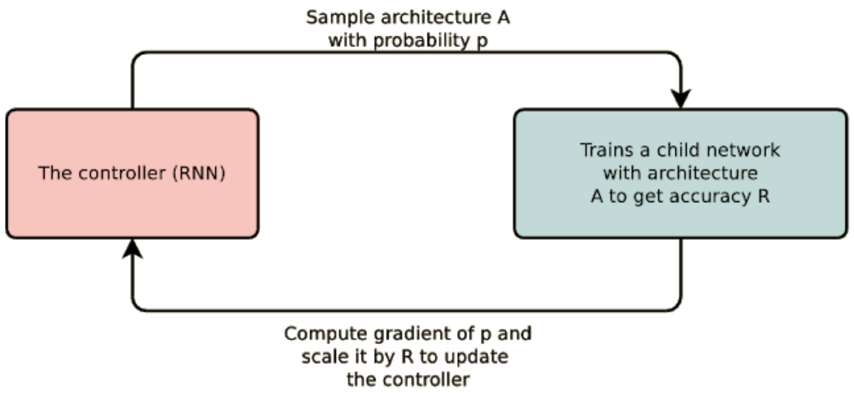
In general, we can set learning rates to be different for each layer to find a tradeoff between freezing and fine tuning

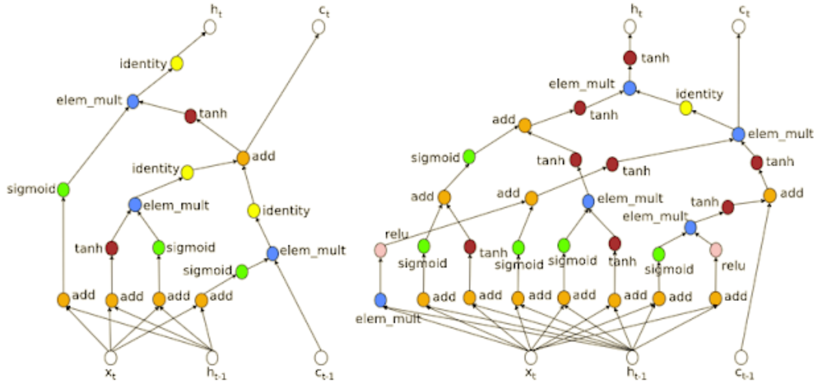


https://colab.research.google.com/github/tensorflow/docs/blob/master/site/en/r2/tutorials/images/transfer_learning.ipynb#scrollTo=wvidPx6jeFzf

NASNet and AutoML

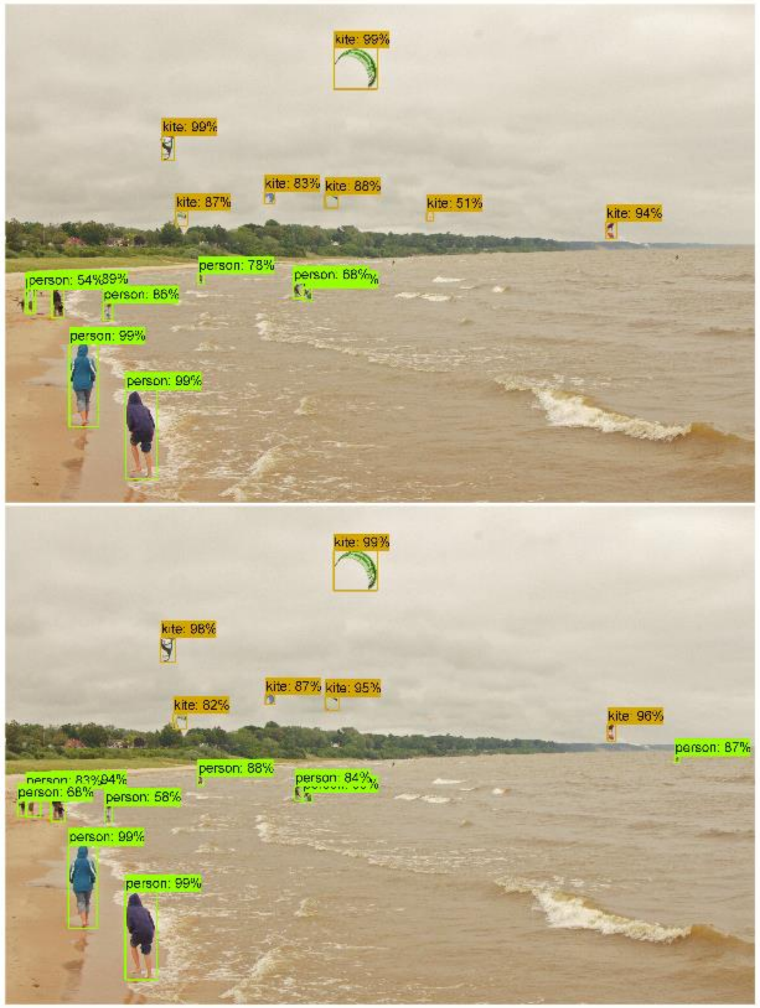
AI build other AI
Using Reinforcement Learning





Learning Transferable Architectures for Scalable Image Recognition, Jul 2017
Google, https://arxiv.org/pdf/1707.07012.pdf
https://research.googleblog.com/2017/05/using-machine-learning-to-explore.html

Neural Architecture Search



Pipeline:
Faster R-CNN
Feature map Generator:
Inception-ResNet

Pipeline:
Faster R-CNN
Feature map Generator:
NASNet

OpenAI
Education

OpenAI
Spinning Up

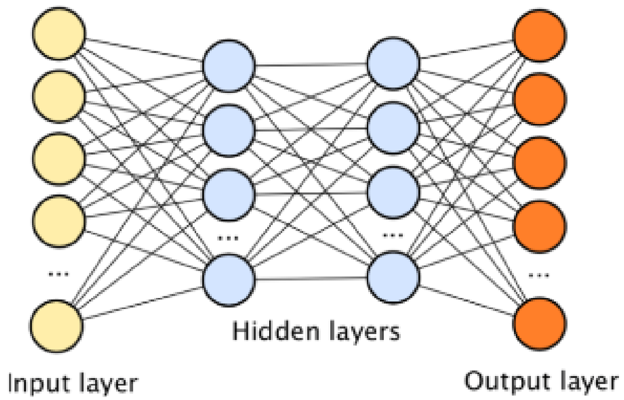
https://blog.openai.com/openai-charter/

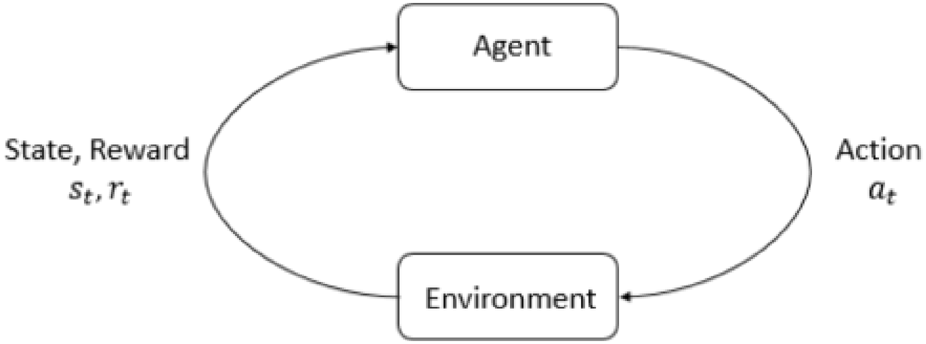
Deep RL Introduction

What is Deep RL?

Deep RL is the combination of **reinforcement learning (RL)** with **deep learning**

- **Reinforcement learning** is about solving problems by **trial and error**
- **Deep learning** is about using **deep neural networks** to solve problems
- $\Longrightarrow$ **Deep reinforcement learning** trains deep neural networks with trial and error





Deep neural network[1] and RL interaction loop

What is Deep RL?

RL is useful when

- you have a sequential decision-making problem
- you do **not** know the optimal behavior already[1]
- but you can still evaluate whether behaviors are good or bad

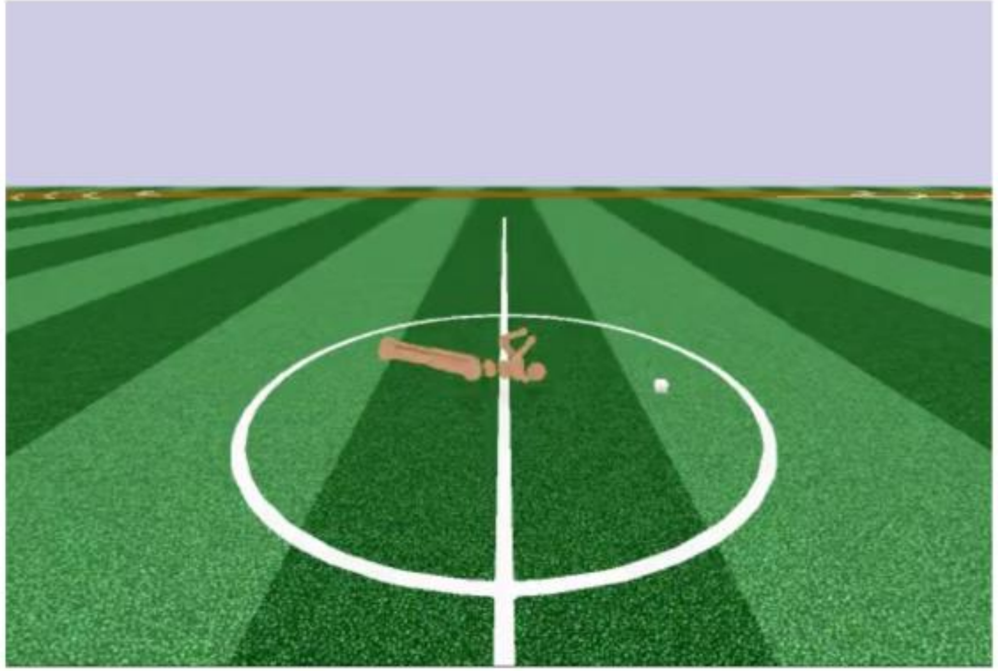
Deep learning is useful when

- you want to approximate a function
- function requires "intelligence"
- inputs and/or outputs are high-dimensional
- lots of data is available

Uses Deep RL …

Deep RL can...

- Play video games from raw pixels
- Control robots in simulation and in the real world
- Play Go, Dota, and Starcraft at superhuman levels



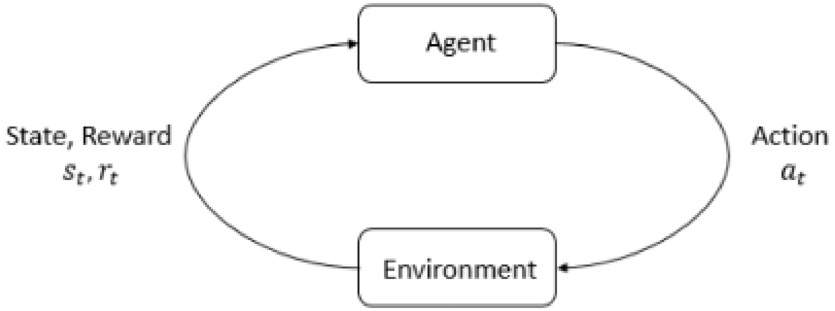
RL Concepts

What are the core parts for understand RL:

- Observations and actions spaces
- Policies
- Trajectories
- Reward and return
- RL optimization problem
- Value and Action-Value Functions

RL Setup

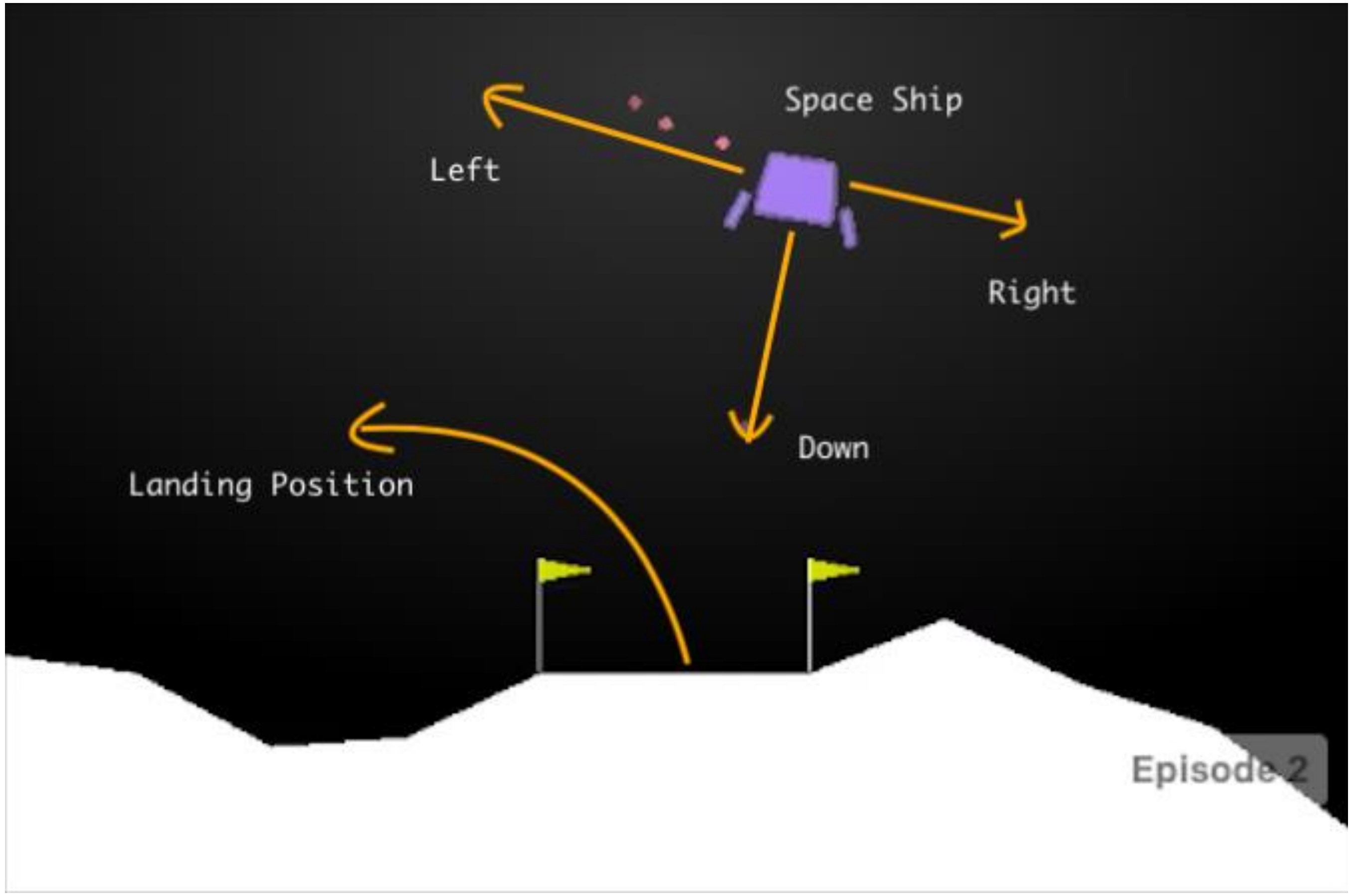
- An *agent* interacts with an *environment*.




```
obs = env.reset()
done = False
while not(done):
  act = agent.get_action(obs)
  next_obs, reward, done, info = env.step(act)
  obs = next_obs
```

- The goal of the agent is to maximize cumulative reward (called *return*).
- The agent figures out how to attain its goal by trial and error.
- Reinforcement learning (RL) is a field of study for algorithms that do that.

Environment



Get environment

- pip install gym
- conda install pystan
- conda install git
- pip install git+https://github.com/Kojoley/atari-py.git
- conda install swig
- pip install box2d-py && pip install Box2D
- pip install gym[all]
- pip install pyglet==1.2.4
- pip install gym[box2d]

Policy

A **policy** π is a rule for selecting actions. It can be either

- **stochastic**, which means that it gives a probability distribution over actions, and actions are selected randomly based on that distribution ($a_t \sim \pi(\cdot|s_t)$),
- or **deterministic**, which means that π directly maps to an action ($a_t = \pi(s_t)$).

Examples of policies:

- Stochastic policy over discrete actions:

```
obs = tf.placeholder(shape=(None, obs_dim), dtype=tf.float32)
net = mlp(obs, hidden_dims=(64,64), activation=tf.tanh)
logits = tf.layers.dense(net, units=num_actions, activation=None)
actions = tf.squeeze(tf.multinomial(logits=logits,num_samples=1), axis=1)
```

- Deterministic policy for a vector-valued continuous action:

```
obs = tf.placeholder(shape=(None, obs_dim), dtype=tf.float32)
net = mlp(obs, hidden_dims=(64,64), activation=tf.tanh)
actions = tf.layers.dense(net, units=act_dim, activation=None)
```

Trajectories

- A **trajectory** τ is a sequence of states and actions in an environment:

$$\tau = (s_0, a_0, s_1, a_1, ...).$$

  - The initial state s_0 is sampled from a *start state distribution* μ:

$$s_0 \sim \mu(\cdot).$$

  - State transitions depend only on the most recent state and action. They could be deterministic:

$$s_{t+1} = f(s_t, a_t),$$

    or stochastic:

$$s_{t+1} \sim P(\cdot|s_t, a_t).$$

- A trajectory is sometimes also called an **episode** or **rollout**.

Reward

The **reward** function of an environment measures how good state-action pairs are:

$$r_t = R(s_t, a_t).$$

The **return** of a trajectory is a measure of cumulative reward along it. There are two main ways to compute return:

- Finite horizon undiscounted sum of rewards::

$$R(\tau) = \sum_{t=0}^{T} r_t$$

- Infinite horizon discounted sum of rewards:

$$R(\tau) = \sum_{t=0}^{\infty} \gamma^t r_t$$

  where $\gamma \in (0, 1)$. This makes rewards less valuable if they are further in the future. (Why would we ever want this? Think about cash: it's valuable to have it sooner rather than later!)

Reward-to-go

A closely-related quantity is **reward-to-go**, which is "return starting from a state or timestep":

$$R_t = \sum_{t'=t}^{T} r_{t'}$$

or

$$R_t = \sum_{t'=t}^{\infty} \gamma^{t'-t} r_{t'}$$

What to learn?

In RL we want a policy which maximizes expected return. Thus the performance measure is

$$J(\pi) = \mathop{\mathrm{E}}_{\tau \sim \pi} [R(\tau)],$$

and the **optimal policy** π^* is:

$$\pi^* = \arg\max_{\pi} J(\pi)$$

Note that by $\tau \sim \pi$, we mean

$$s_0 \sim \mu(\cdot), \qquad a_t \sim \pi(\cdot|s_t), \qquad s_{t+1} \sim P(\cdot|s_t, a_t).$$

V^{π} function and Q^{π} function

Following a policy produces sample trajectories (or paths) $s_0, a_0, r_0, s_1, a_1, r_1, \ldots$

How good is a state?

The **value function** at state s, is the expected cumulative reward from following the policy from state s:

$$V^{\pi}(s) = \mathbb{E}\left[\sum_{t \geq 0} \gamma^t r_t | s_0 = s, \pi\right]$$

How good is a state-action pair?

The **Q-value function** at state s and action a, is the expected cumulative reward from taking action a in state s and then following the policy:

$$Q^{\pi}(s, a) = \mathbb{E}\left[\sum_{t \geq 0} \gamma^t r_t | s_0 = s, a_0 = a, \pi\right]$$

Advantage function

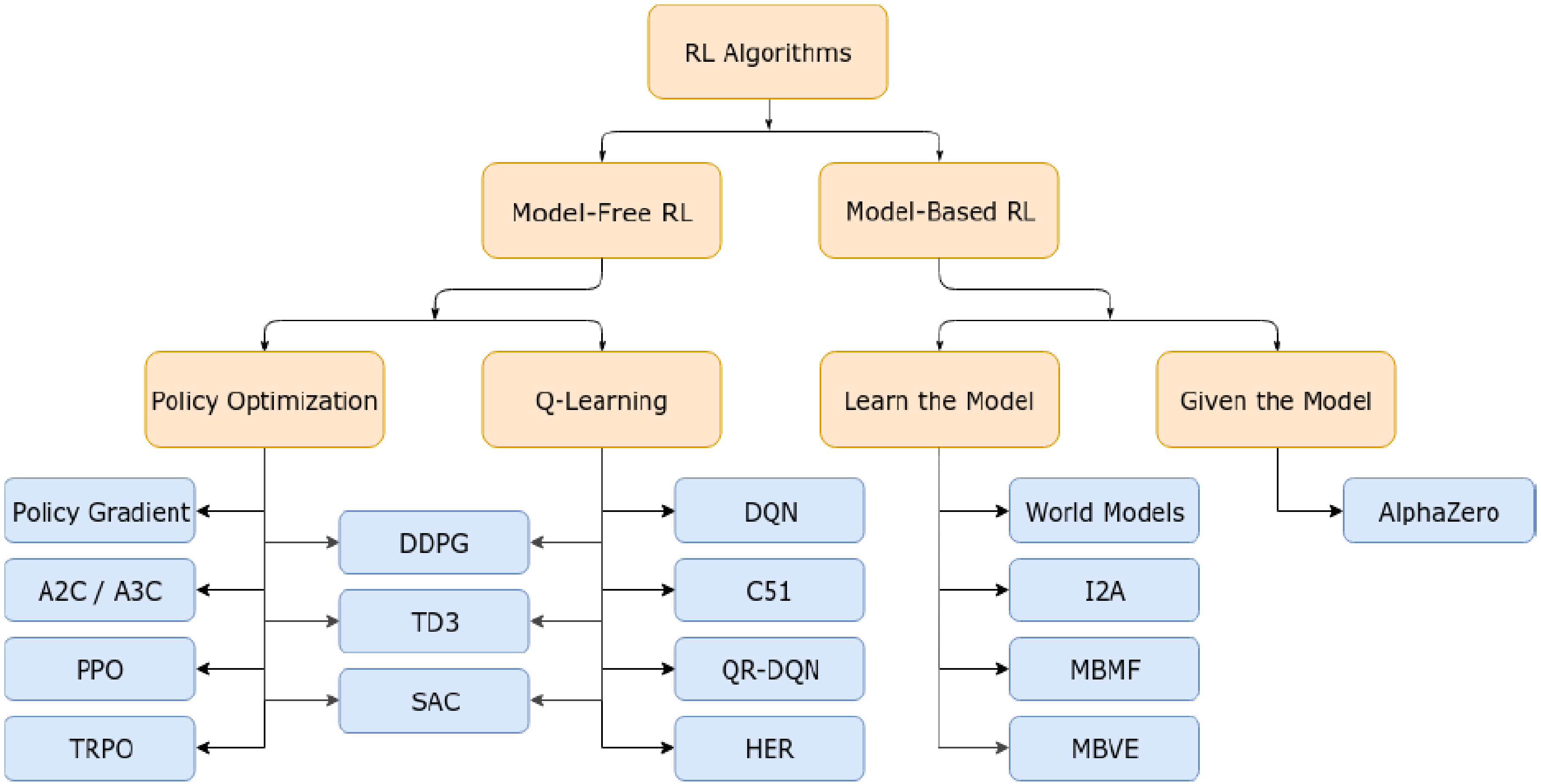
Value and action-value functions are connected:

$$V^{\pi}(s) = \underset{a \sim \pi}{\mathrm{E}} [Q^{\pi}(s, a)]$$

The advantage function for a policy tells you how much better or worse one action is than average:

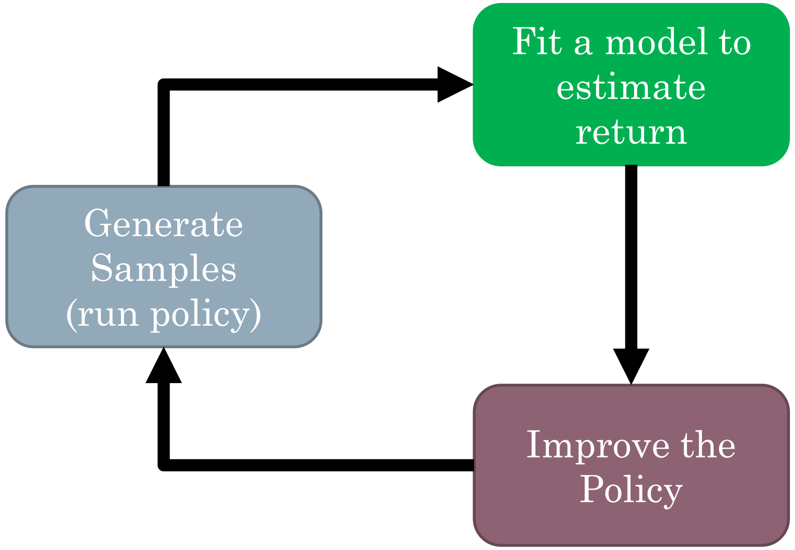
$$A^{\pi}(s, a) = Q^{\pi}(s, a) - V^{\pi}(s)$$

Deep RL Tree



RL Algorithms
Model-Free RL
Model-Based RL
Policy Optimization
Q-Learning
Learn the Model
Given the Model
Policy Gradient
A2C / A3C
PPO
TRPO
DDPG
TD3
SAC
DQN
C51
QR-DQN
HER
World Models
I2A
MBMF
MBVE
AlphaZero

RL flow



- Policy Optimization

- Policy $\pi_\theta(a|s)$
- Optimize θ using objective function $J(\pi_\theta)$
- On-Policy, only use data collected from recent policy
- Can involve learning a value function $V^\pi(s)$
- A2C/A3C , TRPO

$$J(\pi_\theta) = \sum_{t=1}^{T} \gamma^t r_t$$

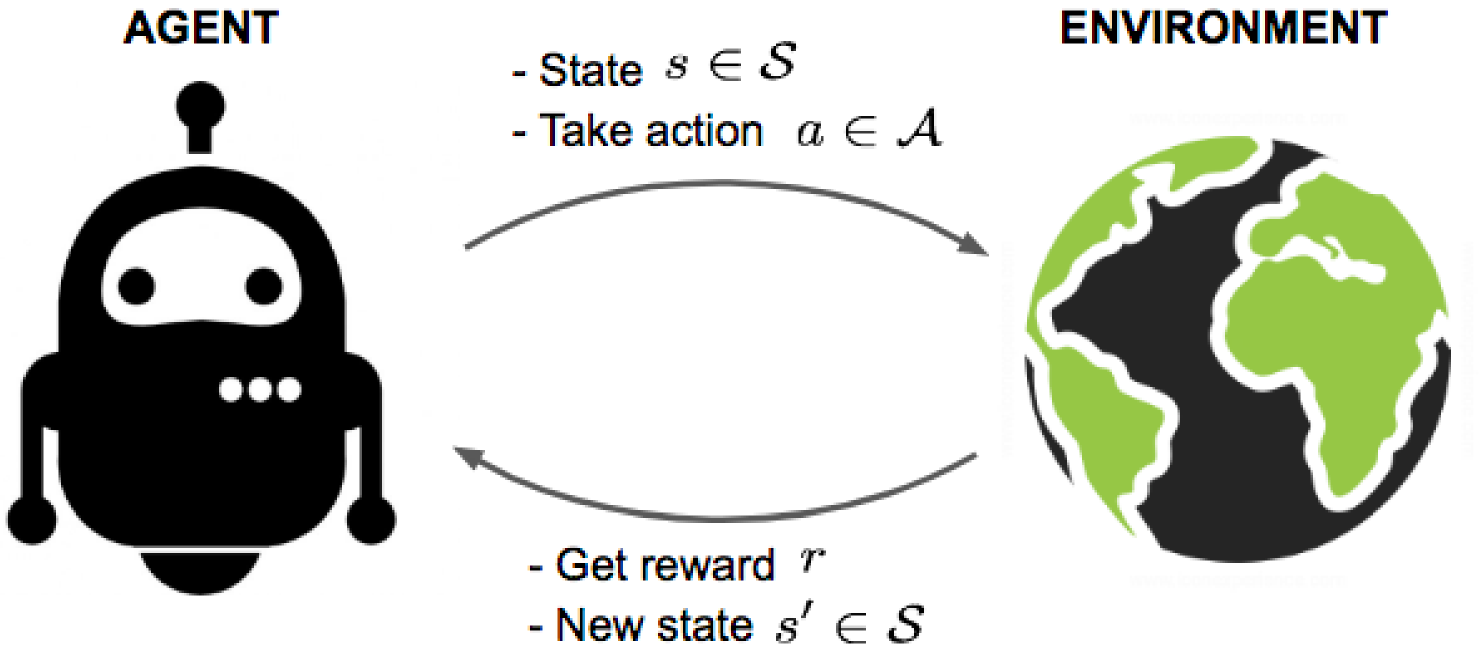
$$\theta \leftarrow \theta + \alpha \nabla_\theta J(\pi_\theta)$$

- Q-Learning

- Learn to optimize an action-value function $Q_\theta(s, a)$
- Function based on *Bellman-equation*
- Off-policy as each update use data collected at any point
- DQN, Double-DQN

$$fit\ Q_{\pi_\theta}(s, a|\theta)$$

$$\pi_\theta(s) = argmax\left(Q_{\pi_\theta}(s, a)\right)$$



Deep Q-Network

DQN

Q-Learning

- Run policy: Step in env with action from Q_θ, store to replay buffer
- Evaluate policy: Update Q_θ to minimize Bellman error, using all previous data (**off-policy**)
- Improve policy: $a^* = \arg\max_a Q_\theta(s, a)$

Q-learning: Use a function approximator to estimate the action-value function

$$Q(s, a; \theta) \approx Q^*(s, a)$$

function parameters (weights)

If the function approximator is a deep neural network => **deep q-learning**!

DQN

- Collect experience in the environment using a policy which trades off between acting randomly and acting according to current Q_θ
- Interleave data collection with updates to Q_θ to minimize Bellman error by bootstrapping:

$$\min_\theta \sum_{(s,a,s',r)\in\mathcal{D}} \left(Q_\theta(s,a) - y(s',r)\right)^2$$

where

$$y(s',r) = r + \gamma \max_{a'} Q_\theta(s',a'),$$

and gradients don't propagate through y.

DQN

Experience replay:

- Data distribution changes over time: as your Q function gets better and you *exploit* this, you visit different (s, a, s', r) transitions than you did earlier
- Stabilize learning by keeping old transitions in a replay buffer, and taking minibatch gradient descent on mix of old and new transitions

Target networks:

- Bootstrapping with function approximators is unstable!
- It's *like* regression, but it's not: targets depend on parameters θ—so an update to Q changes the target!

$$y(s', r) = r + \gamma \max_{a'} Q_\theta(s', a').$$

- Stabilize it by *holding the target fixed* for a while: keep a separate target network, $Q_{\theta_{targ}}$, and every k steps update $\theta_{targ} \leftarrow \theta$
- If unstable, why do it? Because it works well! (Plus theory, later)

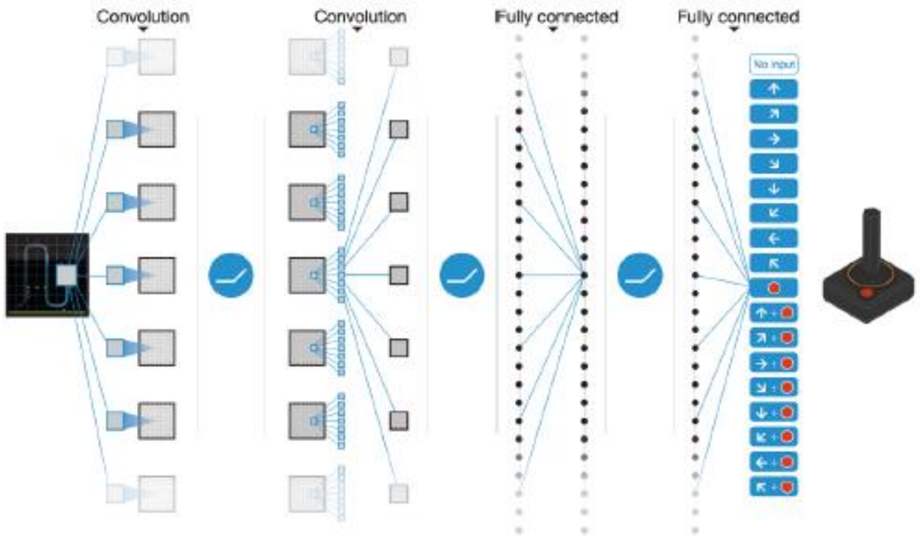
DQN

What we've described so far requires the computation of

$$\max_a Q_\theta(s, a),$$

both for selecting actions and calculating update targets.

Problem: for continuous action spaces, this is nontrivially hard! DQN therefore only applies for **discrete actions**, because we can output one Q-value per action from the network.



DQN

Algorithm 2 Deep Q-Learning

Randomly generate Q-function parameters θ
Set target Q-network parameters $\theta_{targ} \leftarrow \theta$
Make empty replay buffer $\mathcal{D}$
Receive observation s_0 from environment
for $t = 0, 1, 2, \ldots$ **do**
  With probability ϵ, select random action a_t; otherwise select $a_t = \arg\max_a Q_\theta(s_t, a)$
  Step environment to get s_{t+1}, r_t and end-of-episode signal d_t
  Linearly decay ϵ until it reaches final value ϵ_f
  Store $(s_t, a_t, r_t, s_{t+1}, d_t) \rightarrow \mathcal{D}$
  Sample mini-batch of transitions $B = \{(s, a, r, s', d)_i\}$ from $\mathcal{D}$
  For each transition in B, compute

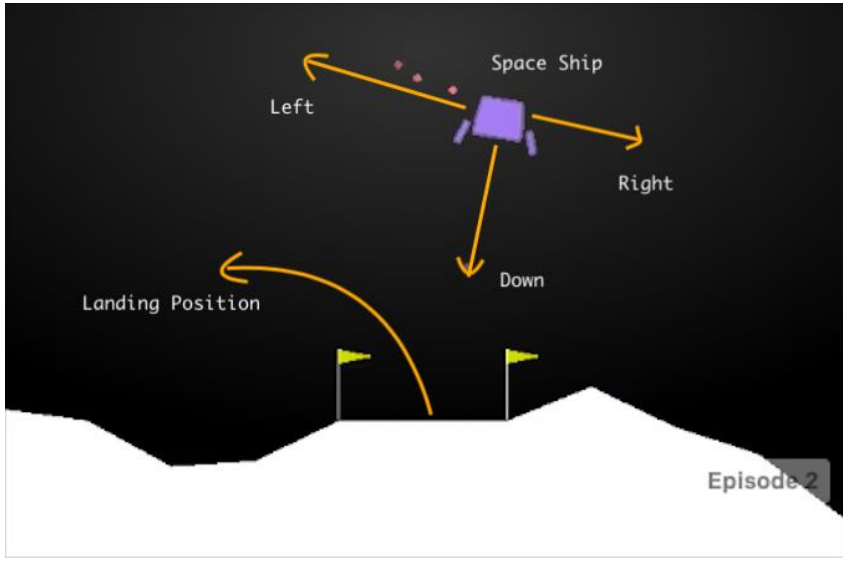
$$y = \begin{cases} r & \text{transition is terminal } (d = \text{True}) \\ r + \gamma \max_{a'} Q_{\theta_{targ}}(s', a') & \text{otherwise} \end{cases}$$

  Update Q by gradient descent on regression loss:

$$\theta \leftarrow \theta - \alpha \nabla_\theta \sum_{(s,a,y) \in B} (Q_\theta(s, a) - y)^2$$

  if $t \% t_{update} = 0$ **then**
    Set $\theta_{targ} \leftarrow \theta$
  end if
end for

Example



- Fuel is infinite, so an agent can learn to fly and then land on its first attempt.

Observation
- Landing pad is always at coordinates (0,0).
- Coordinates are the first two numbers in state vector.

Reward
- From the top of the screen to landing pad and zero speed is about 100..140 points.
- If lander moves away from landing pad it loses reward back.
- Episode finishes if the lander crashes or comes to rest, receiving additional -100 or +100 points.
- Each leg ground contact is +10.
- Firing main engine is -0.3 points each frame.
- Solved is 200 points.

Actions
- Four discrete actions available: do nothing, fire left orientation engine, fire main engine, fire right orientation engine.

DQN

Algorithm 2 Deep Q-Learning

Randomly generate Q-function parameters θ
Set target Q-network parameters $\theta_{targ} \leftarrow \theta$
Make empty replay buffer $\mathcal{D}$
Receive observation s_0 from environment

```
class DQN:
    def __init__(self, action_space, state_space):
        self.action_space = action_space
        self.state_space = state_space
        self.epsilon = 1.0
        self.gamma = .99
        self.batch_size = 64
        self.epsilon_min = .01
        self.lr = 0.001
        self.epsilon_decay = .996
        self.memory = deque(maxlen=1000000)
        self.model = self.build_model()

    def build_model(self):

        model = Sequential()
        model.add(Dense(150, input_dim=self.state_space, activation=relu))
        model.add(Dense(120, activation=relu))
        model.add(Dense(self.action_space, activation=linear))
        model.compile(loss='mse', optimizer=adam(lr=self.lr))
        return model
```

DQN

for $t = 0, 1, 2, \ldots$ **do**
  With probability ϵ, select random action a_t; otherwise select $a_t = \arg\max_a Q_\theta(s_t, a)$
  Step environment to get s_{t+1}, r_t and end-of-episode signal d_t

```python
def act(self, state):

    if np.random.rand() <= self.epsilon:
        return random.randrange(self.action_space)
    act_values = self.model.predict(state)
    return np.argmax(act_values[0])
```

```python
agent = DQN(env.action_space.n, env.observation_space.shape[0])
for e in range(episode):
    state = env.reset()
    state = np.reshape(state, (1, 8))
    score = 0
    max_steps = 3000
    for i in range(max_steps):
        action = agent.act(state)
        env.render()
        next_state, reward, done, _ = env.step(action)
        score += reward
        next_state = np.reshape(next_state, (1, 8))
```

DQN

for $t = 0, 1, 2, \dots$ **do**
  With probability ϵ, select random action a_t; otherwise select $a_t = \arg\max_a Q_\theta(s_t, a)$
  Step environment to get s_{t+1}, r_t and end-of-episode signal d_t
  Linearly decay ϵ until it reaches final value ϵ_f
  Store $(s_t, a_t, r_t, s_{t+1}, d_t) \rightarrow \mathcal{D}$

```
if self.epsilon > self.epsilon_min:
    self.epsilon *= self.epsilon_decay
```

```
def remember(self, state, action, reward, next_state, done):
    self.memory.append((state, action, reward, next_state, done))
```

```
agent.remember(state, action, reward, next_state, done)
```

DQN

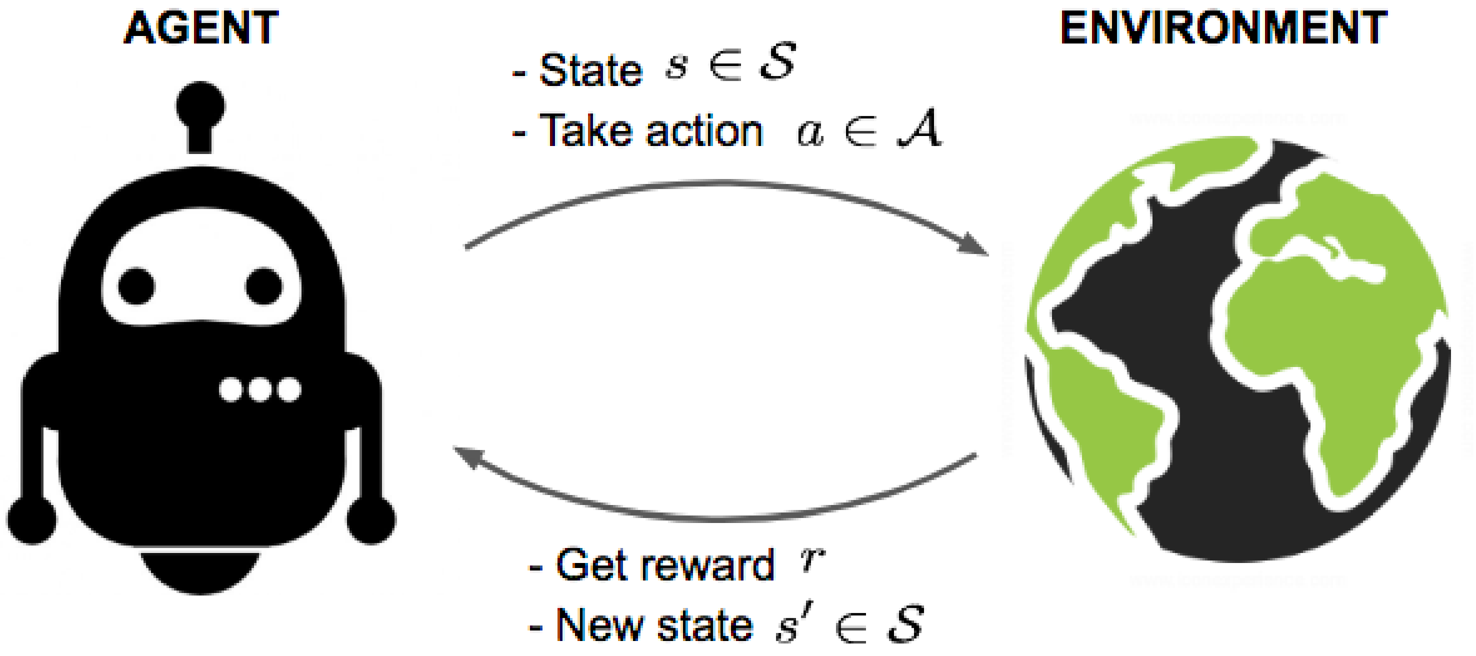
Sample mini-batch of transitions $B = \{(s, a, r, s', d)_i\}$ from $\mathcal{D}$
For each transition in B, compute

$$y = \begin{cases} r & \text{transition is terminal } (d = \text{True}) \\ r + \gamma \max_{a'} Q_{\theta_{targ}}(s', a') & \text{otherwise} \end{cases}$$

Update Q by gradient descent on regression loss:

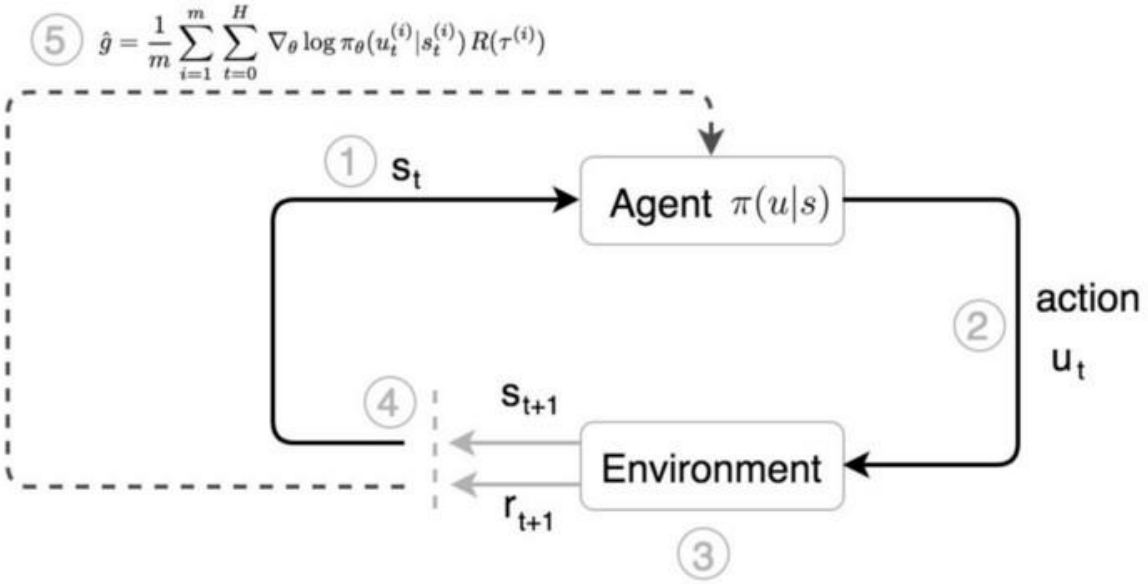
$$\theta \leftarrow \theta - \alpha \nabla_\theta \sum_{(s,a,y) \in B} (Q_\theta(s, a) - y)^2$$

```python
def replay(self):

    if len(self.memory) < self.batch_size:
        return

    minibatch = random.sample(self.memory, self.batch_size)
    states = np.array([i[0] for i in minibatch])
    actions = np.array([i[1] for i in minibatch])
    rewards = np.array([i[2] for i in minibatch])
    next_states = np.array([i[3] for i in minibatch])
    dones = np.array([i[4] for i in minibatch])

    states = np.squeeze(states)
    next_states = np.squeeze(next_states)

    q_val = (np.amax(self.model.predict_on_batch(next_states), axis=1))
    targets = rewards + self.gamma*q_val*(1-dones)
    targets_full = self.model.predict_on_batch(states)
    ind = np.array([i for i in range(self.batch_size)])
    targets_full[[ind], [actions]] = targets

    self.model.fit(states, targets_full, epochs=1, verbose=0)
```



Policy Optimization

REINFORCE



Policy Optimization

- Run policy: Collect trajectories $\tau \sim \pi_\theta$
- Evaluate policy: Estimate V^{π_θ}, A^{π_θ} using current trajectories (**on-policy**)
- Improve policy: Increase likelihood of actions with high advantage

REINFORCE

Goal: derive an expression for $\nabla_\theta J(\pi_\theta)$ which we can compute with a sample estimate, as the basis for a direct gradient ascent algorithm

$$\theta \leftarrow \theta + \alpha \nabla_\theta J(\pi_\theta)$$

Well what happens if we just...

$$\nabla_\theta J(\pi_\theta) = \nabla_\theta \mathop{\mathrm{E}}_{\tau \sim \pi_\theta} [R(\tau)]$$

Problem: parameters are in distribution!

REINFORCE

Solution: expand expectation into integral, use log-derivative trick

$$
\begin{aligned}
\nabla_\theta J(\pi_\theta) &= \nabla_\theta \mathop{\mathrm{E}}_{\tau\sim\pi_\theta}[R(\tau)] \\
&= \nabla_\theta \int d\tau P(\tau|\pi_\theta)R(\tau) \\
&= \int d\tau \nabla_\theta P(\tau|\pi_\theta)R(\tau) \\
&= \int d\tau P(\tau|\pi_\theta)\nabla_\theta \log P(\tau|\pi_\theta)R(\tau) \\
&= \mathop{\mathrm{E}}_{\tau\sim\pi_\theta}[\nabla_\theta \log P(\tau|\pi_\theta)R(\tau)]
\end{aligned}
$$

But are we done yet? No! Still need to compute $\nabla_\theta \log P(\tau|\pi_\theta)$

REINFORCE

What is $P(\tau|\pi_\theta)$?

$$P(\tau|\pi_\theta) = \mu(s_0)\prod_{t=0}^{T} P(s_{t+1}|s_t, a_t)\pi_\theta(a_t|s_t)$$

Thus:

$$\begin{aligned}
\nabla_\theta \log P(\tau|\pi_\theta) &= \nabla_\theta \log\left(\mu(s_0)\prod_{t=0}^{T} P(s_{t+1}|s_t, a_t)\pi_\theta(a_t|s_t)\right) \\
&= \nabla_\theta\left(\log\mu(s_0) + \sum_{t=0}^{T}(\log P(s_{t+1}|s_t, a_t) + \log\pi_\theta(a_t|s_t))\right) \\
&= \nabla_\theta \log\mu(s_0) + \sum_{t=0}^{T}(\nabla_\theta \log P(s_{t+1}|s_t, a_t) + \nabla_\theta \log\pi_\theta(a_t|s_t)) \\
&= \cancel{\nabla_\theta \log\mu(s_0)} + \sum_{t=0}^{T}\left(\cancel{\nabla_\theta \log P(s_{t+1}|s_t, a_t)} + \nabla_\theta \log\pi_\theta(a_t|s_t)\right)
\end{aligned}$$

$$\therefore \nabla_\theta \log P(\tau|\pi_\theta) = \sum_{t=0}^{T}\nabla_\theta \log\pi_\theta(a_t|s_t)$$

REINFORCE

Putting it all together so far:

$$\nabla_\theta J(\pi_\theta) = \mathop{\mathrm{E}}_{\tau\sim\pi_\theta}\left[\sum_{t=0}^{T} \nabla_\theta \log \pi_\theta(a_t|s_t) R(\tau)\right]$$

So we could estimate with:

$$\nabla_\theta J(\pi_\theta) \approx \frac{1}{|\mathcal{D}|} \sum_{\tau\in\mathcal{D}} \sum_{t=0}^{T} \nabla_\theta \log \pi_\theta(a_t|s_t) R(\tau)$$

But not good enough! Variance will be high.

Insight: future actions and past rewards should be uncorrelated. That is:

$$\text{for } t > t', \quad \mathrm{E}\left[\nabla_\theta \log \pi_\theta(a_t|s_t) r_{t'}\right] = 0$$

$$\therefore \nabla_\theta J(\pi_\theta) = \mathop{\mathrm{E}}_{\tau\sim\pi_\theta}\left[\sum_{t=0}^{T} \nabla_\theta \log \pi_\theta(a_t|s_t) \sum_{t'=t}^{T} r_{t'}\right]$$

REINFORCE

REINFORCE algorithm:

1. sample $\{\tau^i\}$ from $\pi_\theta(\mathbf{a}_t|\mathbf{s}_t)$ (run the policy)
2. $\nabla_\theta J(\theta) \approx \sum_i \left(\sum_t \nabla_\theta \log \pi_\theta(\mathbf{a}_t^i|\mathbf{s}_t^i)\right)\left(\sum_t r(\mathbf{s}_t^i, \mathbf{a}_t^i)\right)$
3. $\theta \leftarrow \theta + \alpha \nabla_\theta J(\theta)$

Policy gradient:

```
# Given:
# actions - (N*T) x Da tensor of actions
# states - (N*T) x Ds tensor of states
# q_values – (N*T) x 1 tensor of estimated state-action values
# Build the graph:
logits = policy.predictions(states) # This should return (N*T) x Da tensor of action logits
negative_likelihoods = tf.nn.softmax_cross_entropy_with_logits(labels=actions, logits=logits)
weighted_negative_likelihoods = tf.multiply(negative_likelihoods, q_values)
loss = tf.reduce_mean(weighted_negative_likelihoods)
gradients = loss.gradients(loss, variables)
```

Policy Gradient Improve

What we have currently: "Reward-to-Go" policy gradient:

$$\nabla_\theta J(\pi_\theta) = \mathop{\mathrm{E}}_{\tau\sim\pi_\theta}\left[\sum_{t=0}^{T} \nabla_\theta \log \pi_\theta(a_t|s_t) \sum_{t'=t}^{T} r_{t'}\right]$$

Observe: expectation can be broken up, letting us transform "Reward-to-Go" into Q^π:

$$\begin{aligned}
\nabla_\theta J(\pi_\theta) &= \mathop{\mathrm{E}}_{\tau\sim\pi_\theta}\left[\sum_{t=0}^{T} \nabla_\theta \log \pi_\theta(a_t|s_t) \sum_{t'=t}^{T} r_{t'}\right] \\
&= \sum_{t=0}^{T} \mathop{\mathrm{E}}_{\tau\sim\pi_\theta}\left[\nabla_\theta \log \pi_\theta(a_t|s_t) \sum_{t'=t}^{T} r_{t'}\right] \\
&= \sum_{t=0}^{T} \mathop{\mathrm{E}}_{\tau_{0:t}\sim\pi_\theta}\left[\nabla_\theta \log \pi_\theta(a_t|s_t) \mathop{\mathrm{E}}_{\tau_{(t+1):T}\sim\pi_\theta}\left[\sum_{t'=t}^{T} r_{t'}\right]\right] \\
&= \sum_{t=0}^{T} \mathop{\mathrm{E}}_{\tau_{0:t}\sim\pi_\theta}\left[\nabla_\theta \log \pi_\theta(a_t|s_t) Q^{\pi_\theta}(s_t, a_t)\right]
\end{aligned}$$

$$\therefore \nabla_\theta J(\pi_\theta) = \mathop{\mathrm{E}}_{\tau\sim\pi_\theta}\left[\sum_{t=0}^{T} \nabla_\theta \log \pi_\theta(a_t|s_t) Q^{\pi_\theta}(s_t, a_t)\right]$$

REINFORCE

What is a **baseline**? A function $b(s_t)$ with

$$\nabla_\theta J(\pi_\theta) = \mathop{\mathrm{E}}_{\tau\sim\pi_\theta}\left[\sum_{t=0}^{T}\nabla_\theta \log \pi_\theta(a_t|s_t)\left(Q^{\pi_\theta}(s_t,a_t) - b(s_t)\right)\right]$$

Claim: this works for any b! Proof:

$$\begin{aligned}
\mathop{\mathrm{E}}_{a_t\sim\pi_\theta(\cdot|s_t)}\left[\nabla_\theta \log \pi_\theta(a_t|s_t) b(s_t)\right] &= \mathop{\mathrm{E}}_{a_t\sim\pi_\theta(\cdot|s_t)}\left[\nabla_\theta \log \pi_\theta(a_t|s_t)\right] b(s_t)\\
&= \left(\int da\pi_\theta(a_t|s_t)\nabla_\theta \log \pi_\theta(a_t|s_t)\right) b(s_t)\\
&= \left(\int da\nabla_\theta\pi_\theta(a_t|s_t)\right) b(s_t)\\
&= \left(\nabla_\theta \int da\pi_\theta(a_t|s_t)\right) b(s_t)\\
&= \cancel{(\nabla_\theta 1)} b(s_t)\\
&= 0
\end{aligned}$$

Policy Gradient Improve

If we choose $b = V^{\pi}$, we get the **advantage form** of the policy gradient:

$$\begin{aligned}\nabla_\theta J(\pi_\theta) &= \underset{\tau \sim \pi_\theta}{\mathrm{E}}\left[\sum_{t=0}^{T} \nabla_\theta \log \pi_\theta(a_t|s_t)\left(Q^{\pi_\theta}(s_t, a_t) - V^{\pi_\theta}(s_t)\right)\right] \\ &= \underset{\tau \sim \pi_\theta}{\mathrm{E}}\left[\sum_{t=0}^{T} \nabla_\theta \log \pi_\theta(a_t|s_t) A^{\pi_\theta}(s_t, a_t)\right]\end{aligned}$$

Why do we want this? Better signal in sample estimate: removes “stuff that would have happened anyway” from Q^{π}

REINFORCE

What we've shown so far:

$$
\begin{aligned}
\nabla_\theta J(\pi_\theta) &= \underset{\tau \sim \pi_\theta}{\mathrm{E}} \left[\sum_{t=0}^{T} \nabla_\theta \log \pi_\theta(a_t|s_t) R(\tau) \right] \\
&= \underset{\tau \sim \pi_\theta}{\mathrm{E}} \left[\sum_{t=0}^{T} \nabla_\theta \log \pi_\theta(a_t|s_t) \sum_{t'=t}^{T} r_{t'} \right] \\
&= \underset{\tau \sim \pi_\theta}{\mathrm{E}} \left[\sum_{t=0}^{T} \nabla_\theta \log \pi_\theta(a_t|s_t) Q^{\pi_\theta}(s_t, a_t) \right] \\
&= \underset{\tau \sim \pi_\theta}{\mathrm{E}} \left[\sum_{t=0}^{T} \nabla_\theta \log \pi_\theta(a_t|s_t) A^{\pi_\theta}(s_t, a_t) \right]
\end{aligned}
$$

Which form do we use? Almost always the last one.

Policy Gradient Improve

$$\nabla_\theta J(\pi_\theta) = \mathop{\mathrm{E}}_{\tau \sim \pi_\theta}\left[\sum_{t=0}^{T} \nabla_\theta \log \pi_\theta(a_t|s_t) A^{\pi_\theta}(s_t, a_t)\right]$$

- Pushes up the probabilities of "good" actions and pushes down probabilities of "bad" actions
- To estimate, **must sample on-policy**

Policy Gradient Improve

- Generalized Advantage Estimation (**GAE**)
- Natural Policy Gradients (**NPG**)

Algorithm 1 Vanilla Policy Gradient Algorithm

Input: initial policy parameters θ_0, initial value function parameters ϕ_0

for $k = 0, 1, 2, \ldots$ **do**

Collect set of trajectories $\mathcal{D}_k = \{\tau_i\}$ by running policy $\pi_k = \pi(\theta_k)$ in the environment.

Compute rewards-to-go $\hat{R}_t$.

Compute advantage estimates, $\hat{A}_t$ (using any method of advantage estimation) based on the current value function V_{ϕ_k}.

Estimate policy gradient as

$$\hat{g}_k = \frac{1}{|\mathcal{D}_k|} \sum_{\tau \in \mathcal{D}_k} \sum_{t=0}^{T} \nabla_\theta \log \pi_\theta(a_t|s_t)|_{\theta_k} \hat{A}_t.$$

Compute policy update, either using standard gradient ascent,

$$\theta_{k+1} = \theta_k + \alpha_k \hat{g}_k,$$

or via another gradient ascent algorithm like Adam.

Fit value function by regression on mean-squared error:

$$\phi_{k+1} = \arg\min_\phi \frac{1}{|\mathcal{D}_k| T} \sum_{\tau \in \mathcal{D}_k} \sum_{t=0}^{T} \left(V_\phi(s_t) - \hat{R}_t \right)^2,$$

typically via some gradient descent algorithm.

end for

Proximal Policy Gradient

Algorithm 1 PPO-Clip

Input: initial policy parameters θ_0, initial value function parameters ϕ_0

for $k = 0, 1, 2, \ldots$ **do**

  Collect set of trajectories $\mathcal{D}_k = \{\tau_i\}$ by running policy $\pi_k = \pi(\theta_k)$ in the environment.

  Compute rewards-to-go $\hat{R}_t$.

  Compute advantage estimates, $\hat{A}_t$ (using any method of advantage estimation) based on the current value function V_{ϕ_k}.

  Update the policy by maximizing the PPO-Clip objective:

$$\theta_{k+1} = \arg\max_{\theta} \frac{1}{|\mathcal{D}_k| T} \sum_{\tau \in \mathcal{D}_k} \sum_{t=0}^{T} \min\left(\frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_k}(a_t|s_t)} A^{\pi_{\theta_k}}(s_t, a_t),\ \ g(\epsilon, A^{\pi_{\theta_k}}(s_t, a_t)) \right),$$

  typically via stochastic gradient ascent with Adam.

  Fit value function by regression on mean-squared error:

$$\phi_{k+1} = \arg\min_{\phi} \frac{1}{|\mathcal{D}_k| T} \sum_{\tau \in \mathcal{D}_k} \sum_{t=0}^{T} \left(V_\phi(s_t) - \hat{R}_t \right)^2,$$

  typically via some gradient descent algorithm.

end for

Trust Region Policy Gradient

Algorithm 1 Trust Region Policy Optimization

1: Input: initial policy parameters θ_0, initial value function parameters ϕ_0

2: Hyperparameters: KL-divergence limit δ, backtracking coefficient α, maximum number of backtracking steps K

3: **for** $k = 0, 1, 2, ...$ **do**

4: Collect set of trajectories $\mathcal{D}_k = \{\tau_i\}$ by running policy $\pi_k = \pi(\theta_k)$ in the environment.

5: Compute rewards-to-go $\hat{R}_t$.

6: Compute advantage estimates, $\hat{A}_t$ (using any method of advantage estimation) based on the current value function V_{ϕ_k}.

7: Estimate policy gradient as

$$\hat{g}_k = \frac{1}{|\mathcal{D}_k|} \sum_{\tau \in \mathcal{D}_k} \sum_{t=0}^{T} \nabla_\theta \log \pi_\theta(a_t|s_t)\big|_{\theta_k} \hat{A}_t.$$

8: Use the conjugate gradient algorithm to compute

$$\hat{x}_k \approx \hat{H}_k^{-1} \hat{g}_k,$$

where $\hat{H}_k$ is the Hessian of the sample average KL-divergence.

9: Update the policy by backtracking line search with

$$\theta_{k+1} = \theta_k + \alpha^j \sqrt{\frac{2\delta}{\hat{x}_k^T \hat{H}_k \hat{x}_k}} \hat{x}_k,$$

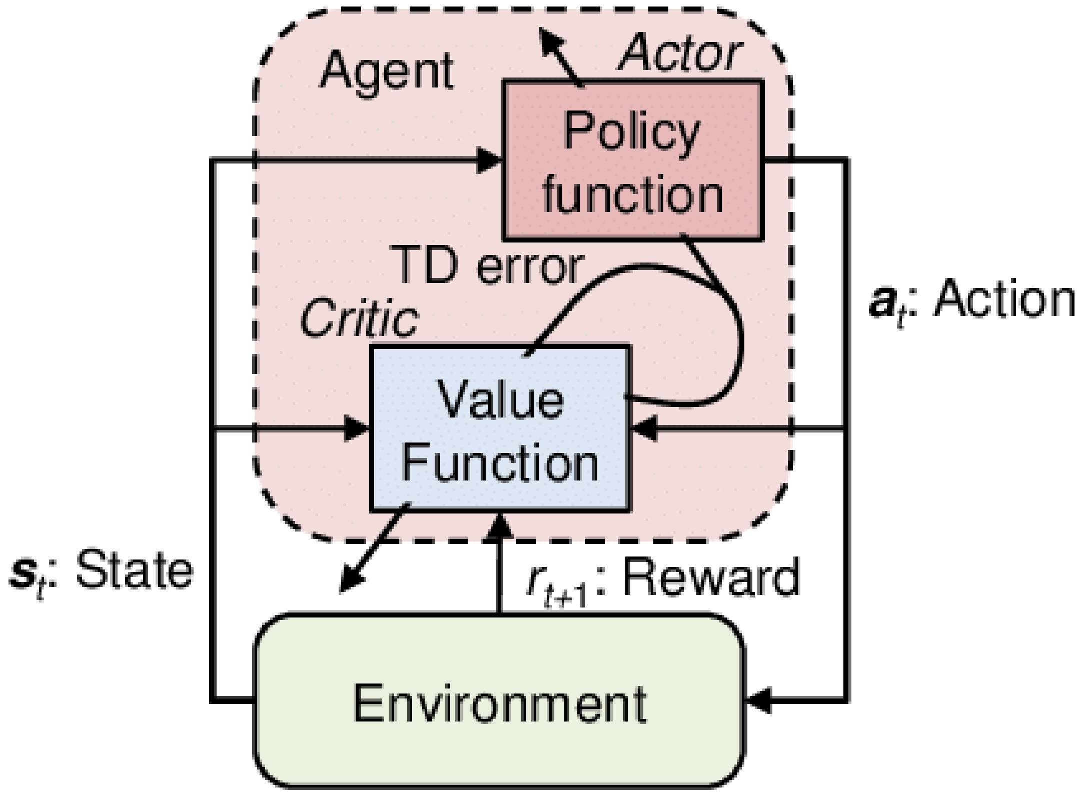
where $j \in \{0, 1, 2, ...K\}$ is the smallest value which improves the sample loss and satisfies the sample KL-divergence constraint.

10: Fit value function by regression on mean-squared error:

$$\phi_{k+1} = \arg\min_\phi \frac{1}{|\mathcal{D}_k| T} \sum_{\tau \in \mathcal{D}_k} \sum_{t=0}^{T} \left(V_\phi(s_t) - \hat{R}_t \right)^2,$$

typically via some gradient descent algorithm.

11: **end for**



Actor-Critic Method

Actor-Critic Method (A2C)

Problem: we don't know Q and V. Can we learn them?

Yes, using Q-learning! We can combine Policy Gradients and Q-learning by training both an **actor** (the policy) and a **critic** (the Q-function).

- The actor decides which action to take, and the critic tells the actor how good its action was and how it should adjust
- Also alleviates the task of the critic as it only has to learn the values of (state, action) pairs generated by the policy
- Can also incorporate Q-learning tricks e.g. experience replay
- **Remark:** we can define by the **advantage function** how much an action was better than expected

$$A^{\pi}(s,a) = Q^{\pi}(s,a) - V^{\pi}(s)$$

DDPG

- Deep Deterministic Policy Gradient (DDPG) is an algorithm which concurrently learns a Q-function and a policy. It uses off-policy data and the Bellman equation to learn the Q-function, and uses the Q-function to learn the policy

  - DDPG is an off-policy algorithm.
  - DDPG can only be used for environments with continuous action spaces.
  - DDPG can be thought of as being deep Q-learning for continuous action spaces.

- To make DDPG policies explore better, we add noise to their actions at training time. The authors of the original DDPG paper recommended time-correlated OU noise, but more recent results suggest that uncorrelated, mean-zero Gaussian noise works perfectly well.

Ornstein–Uhlenbeck process

RL - DDPG

Algorithm 1 Deep Deterministic Policy Gradient

Input: initial policy parameters θ, Q-function parameters ϕ, empty replay buffer $\mathcal{D}$
Set target parameters equal to main parameters $\theta_{\text{targ}} \leftarrow \theta$, $\phi_{\text{targ}} \leftarrow \phi$
repeat
  Observe state s and select action $a = \text{clip}(\mu_\theta(s) + \epsilon, a_{Low}, a_{High})$, where $\epsilon \sim \mathcal{N}$
  Execute a in the environment
  Observe next state s', reward r, and done signal d to indicate whether s' is terminal
  Store (s, a, r, s', d) in replay buffer $\mathcal{D}$
  If s' is terminal, reset environment state.
  if it's time to update **then**
    for however many updates **do**
      Randomly sample a batch of transitions, $B = \{(s, a, r, s', d)\}$ from $\mathcal{D}$
      Compute targets

$$y(r, s', d) = r + \gamma(1 - d) Q_{\phi_{\text{targ}}}(s', \mu_{\theta_{\text{targ}}}(s'))$$

      Update Q-function by one step of gradient descent using

$$\nabla_\phi \frac{1}{|B|} \sum_{(s,a,r,s',d) \in B} \left(Q_\phi(s, a) - y(r, s', d)\right)^2$$

      Update policy by one step of gradient ascent using

$$\nabla_\theta \frac{1}{|B|} \sum_{s \in B} Q_\phi(s, \mu_\theta(s))$$

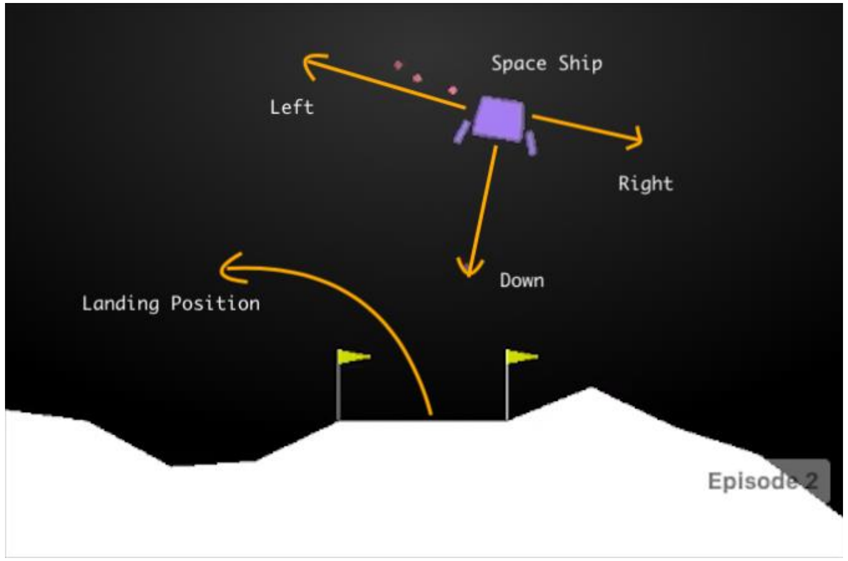
      Update target networks with

$$\phi_{\text{targ}} \leftarrow \rho \phi_{\text{targ}} + (1 - \rho)\phi$$
$$\theta_{\text{targ}} \leftarrow \rho \theta_{\text{targ}} + (1 - \rho)\theta$$

    end for
  end if
until convergence

https://github.com/openai/baselines/tree/master/baselines/ddpg

Example



- Fuel is infinite, so an agent can learn to fly and then land on its first attempt.

https://github.com/germain-hug/Deep-RL-Keras

Observation

- Landing pad is always at coordinates (0,0).
- Coordinates are the first two numbers in state vector.

Reward

- From the top of the screen to landing pad and zero speed is about 100..140 points.
- If lander moves away from landing pad it loses reward back.
- Episode finishes if the lander crashes or comes to rest, receiving additional -100 or +100 points.
- Each leg ground contact is +10.
- Firing main engine is -0.3 points each frame.
- Solved is 200 points.

Actions

- Action is two real values vector from -1 to +1. First controls main engine, -1..0 off, 0..+1 throttle from 50% to 100% power. Engine can't work with less than 50% power. Second value -1.0..-0.5 fire left engine, +0.5..+1.0 fire right engine, -0.5..0.5 off.

Thanks

Contact:

patoalejor@gmail.com