Deep Q Networks

Deep Reinforcement Learning

University of Cambridge

Agenda

- Review

Agenda

- Review
- State of the field

Agenda

- Review
- State of the field
- Implement Deep Q Networks (DQN) (Mnih et al.)

Agenda

- **Review**
- State of the field
- Implement Deep Q Networks (DQN) (Mnih et al.)

Review

Review:

- Finished up MDPs

Review

Review:

- Finished up MDPs
- The return and optimal policies

Review

Review:

- Finished up MDPs
- The return and optimal policies
- Deep Q learning

Review

Review:

- **Finished up MDPs**
- The discounted return and optimal policies
- Deep Q learning

Review

Review:

- Finished up MDPs
- **The return and optimal policies**
- Deep Q learning

Review

The **discounted return (G)**

$$G = \sum_{t=0}^{\infty} \gamma^t R(s_{t+1}) = \sum_{t=0}^{\infty} \gamma^t r_t$$

Review

The **discounted return (G)**

$$G = \sum_{t=0}^{\infty} \gamma^t R(s_{t+1}) = \sum_{t=0}^{\infty} \gamma^t r_t$$

With a reward of 1 at each timestep and $\gamma = 0.9$

$$G = \sum_{t=0}^{\infty} \gamma^t r_t = 1 + 0.9 + 0.81 + ... = \frac{1}{1 - \gamma} = 10$$

Review

The **expected discounted return (G_π)**

$$G_\pi = \mathbb{E}\left[\sum_{t=0}^{\infty} \gamma^t r_t \mid a_t \sim \pi(s_t)\right]$$

Review

$$G_\pi = \mathbb{E}\left[\sum_{t=0}^{\infty} \gamma^t r_t \mid a_t \sim \pi(s_t)\right]$$

Transitions and policy are stochastic. Consider uncertainty in the reward.

Review

$$G_\pi = \mathbb{E}\left[\sum_{t=0}^{\infty} \gamma^t r_t \mid a_t \sim \pi(s_t)\right]$$

Transitions and policy are stochastic. Consider uncertainty in the reward.

$$r_t \sim \left[\underbrace{R(s_{t+1})}_{\text{reward fn}} \overbrace{T(s_{t+1} \mid s_t, a_t)}^{\text{state trans. probs}} \underbrace{\pi(a_t \mid s_t)}_{\text{action probs}}\right]$$

Review

$$r_t \sim \left[\underbrace{R(s_{t+1})}_{\text{reward fn}} \overbrace{T(s_{t+1} \mid s_t, a_t)}^{\text{state trans. probs}} \underbrace{\pi(a_t \mid s_t)}_{\text{action probs}} \right]$$

The **expectation** turns that distribution into a single number. This tells us what reward to expect "on average"

$$\mathbb{E}[r_t] = \int_{s_{t+1}} \int_A \underbrace{R(s_{t+1})}_{\text{reward fn}} \overbrace{T(s_{t+1} \mid s_t, a_t)}^{\text{state trans. probs}} \underbrace{\pi(a_t \mid s_t)}_{\text{action probs}}$$

Review

$$G_\pi = \mathbb{E}\left[\sum_{t=0}^{\infty} \gamma^t r_t \mid a_t \sim \pi(s_t)\right]$$

With the expected discounted return, we can define the optimal policy

$$\pi_* = \max_{\pi} \ \mathbb{E}\left[\sum_{t=0}^{\infty} \gamma^t r_t \mid a_t \sim \pi(s_t)\right]$$

Review

Review:

- Finished up MDPs
- The return and optimal policies
- **Deep Q learning**

Review

The Plan:

1. Derive the value function V
2. Derive Q function from V
3. Figure out a behavior policy using Q
4. Learn to train Q

Review

The Plan:

1. **Derive the value function V**
2. Derive Q function from V
3. Figure out a behavior policy using Q
4. Learn to train Q

Review

With the expected return

$$G_\pi = \mathbb{E}\left[\sum_{t=0}^{\infty} \gamma^t r_t \mid a_t \sim \pi(s_t)\right]$$

We derived the **Value Function (V_π)** $\quad V_\pi : S \to \mathbb{R}$

Review

With the expected return

$$G_\pi = \mathbb{E}\left[\sum_{t=0}^{\infty} \gamma^t r_t \mid a_t \sim \pi(s_t)\right]$$

We derived the **Value Function (V_π)** $\quad V_\pi : S \rightarrow \mathbb{R}$

$$V_\pi(s_0) = \mathbb{E}\left[\sum_{t=0}^{\infty} \gamma^t r_t \mid a_t \sim \pi(s_t)\right]$$

Review

With the expected return

$$G_\pi = \mathbb{E}\left[\sum_{t=0}^{\infty} \gamma^t r_t \mid a_t \sim \pi(s_t)\right]$$

We derived the **Value Function (V_π)** $\quad V_\pi : S \to \mathbb{R}$

$$V_\pi(s_0) = \mathbb{E}\left[\sum_{t=0}^{\infty} \gamma^t r_t \mid a_t \sim \pi(s_t)\right]$$

Difference between G_π and V_π is dependence on s_0

Review

The Plan:

1. Derive the value function V
2. **Derive Q function from V**
3. Figure out a behavior policy using Q
4. Learn to train Q

Review

Factor out first term from the return to introduce a dependence on a_0

$$V_\pi(s_0, a_0) = \mathbb{E}[r_0 \mid a_0] + \mathbb{E}\left[\sum_{t=1}^{\infty} \gamma^t r_t \mid a_t \sim \pi(s_t)\right]$$

Review

Factor out first term from the return to introduce a dependence on a_0

$$V_\pi(s_0, a_0) = \mathbb{E}[r_0 \mid a_0] + \mathbb{E}\left[\sum_{t=1}^{\infty} \gamma^t r_t \mid a_t \sim \pi(s_t)\right]$$

When V depends on a specific action, we call it the **Q function**:

$$S \times A \to \mathbb{R}$$

$$Q_\pi(s_0, a_0) = \mathbb{E}[r_0 \mid a_0] + \mathbb{E}\left[\sum_{t=1}^{\infty} \gamma^t r_t \mid a_t \sim \pi(s_t)\right]$$

Review

The Plan:

1. Derive the value function V
2. Derive Q function from V
3. **Figure out a behavior policy using Q**
4. Learn to train Q

Review

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

In English: Compute Q value for all possible actions and pick the action with the biggest Q value. Repeat at each timestep.

Review

The Plan:

1. Derive the value function V
2. Derive Q function from V
3. Figure out a behavior policy using Q
4. **Learn to train Q**

Review

$$Q(s, a) = r + \gamma \cdot \max_{\{a' \in A\}} Q(s', a')$$

Review

$$Q(s, a) = r + \gamma \cdot \max_{\{a' \in A\}} Q(s', a')$$

This course is **Deep** RL, so we need to use a neural network, parameterized by θ

Review

$$Q(s, a) = r + \gamma \cdot \max_{\{a' \in A\}} Q(s', a')$$

This course is **Deep** RL, so we need to use a neural network, parameterized by θ

$$Q(s, a, \theta) = r + \gamma \cdot \max_{\{a' \in A\}} Q(s', a', \theta)$$

Review

$$Q(s, a) = r + \gamma \cdot \max_{\{a' \in A\}} Q(s', a')$$

This course is **Deep** RL, so we need to use a neural network, parameterized by θ

$$Q(s, a, \theta) = r + \gamma \cdot \max_{\{a' \in A\}} Q(s', a', \theta)$$

$$Q(s, a, \theta) - \left(r + \gamma \cdot \max_{\{a' \in A\}} Q(s', a', \theta) \right) = 0$$

Review

$$Q(s, a) = r + \gamma \cdot \max_{\{a' \in A\}} Q(s', a')$$

This course is **Deep** RL, so we need to use a neural network, parameterized by θ

$$Q(s, a, \theta) - \left(r + \gamma \cdot \max_{\{a' \in A\}} Q(s', a', \theta) \right) = 0$$

Training objective

$$\min_{\theta} \left(Q(s, a, \theta) - \left(r + \gamma \cdot \max_{\{a' \in A\}} Q(s', a', \theta) \right) \right)^2$$

Agenda

- Review
- **State of the field**
- Implement Deep Q Networks (DQN) (Mnih et al.)

Deep RL

Like much of deep learning, Deep RL has a gap between theory and practice

Deep RL

Like much of deep learning, Deep RL has a gap between theory and practice

Just like neural networks (1943) and backpropagation (1970), the theory of value functions (1957) and Q learning (1989) has been around for a long time

Deep RL

Like much of deep learning, Deep RL has a gap between theory and practice

Just like neural networks (1943) and backpropagation (1970), the theory of value functions (1957) and Q learning (1989) has been around for a long time

Hardware advances and good ML software enabled us to take advantage of decades of theory. Mnih et al (2015) took Q learning theory and made it work well with neural networks

Deep RL

Like much of deep learning, Deep RL has a gap between theory and practice

Just like neural networks (1943) and backpropagation (1970), the theory of value functions (1957) and Q learning (1989) has been around for a long time

Hardware advances and good ML software enabled us to take advantage of decades of theory. Mnih et al (2015) took Q learning theory and made it work well with neural networks

Since its inception, Deep RL has added "patches" to combine theory with deep networks to obtain better and better results

Agenda

- Review
- State of the field
- **Implement Deep Q Networks (DQN) (Mnih et al.)**

The Train Loop

Those familiar with deep learning know of the “training loop"

The Train Loop

Those familiar with deep learning know of the “training loop"

```
dataset = load_dataset()
model = nn.Module(dataset.x.size, dataset.y.size)
theta = model.init(seed=0) # Functional

for update in range(num_updates):
  train_data = dataset.sample()
  theta = train(model, theta, train_data) # Functional
  metrics = evaluate(model, theta, dataset.val_set)
```

The Train Loop

Those familiar with deep learning know of the “training loop"

```
==>    dataset = load_dataset()
       model = nn.Module(dataset.x.size, dataset.y.size)
       theta = model.init(seed=0) # Functional

       for update in range(num_updates):
         train_data = dataset.sample()
         theta = train(model, theta, train_data) # Functional
         metrics = evaluate(model, theta, dataset.val_set)
```

The Train Loop

Those familiar with deep learning know of the "training loop"

```
       dataset = load_dataset()
==>    model = nn.Module(dataset.x.size, dataset.y.size)
       theta = model.init(seed=0) # Functional

       for update in range(num_updates):
         train_data = dataset.sample()
         theta = train(model, theta, train_data) # Functional
         metrics = evaluate(model, theta, dataset.val_set)
```

The Train Loop

Those familiar with deep learning know of the “training loop"

```
        dataset = load_dataset()
        model = nn.Module(dataset.x.size, dataset.y.size)
==>     theta = model.init(seed=0) # Functional

        for update in range(num_updates):
          train_data = dataset.sample()
          theta = train(model, theta, train_data) # Functional
          metrics = evaluate(model, theta, dataset.val_set)
```

The Train Loop

Those familiar with deep learning know of the “training loop"

```
        dataset = load_dataset()
        model = nn.Module(dataset.x.size, dataset.y.size)
        theta = model.init(seed=0) # Functional

==>     for update in range(num_updates):
          train_data = dataset.sample()
          theta = train(model, theta, train_data) # Functional
          metrics = evaluate(model, theta, dataset.val_set)
```

The Train Loop

Those familiar with deep learning know of the “training loop"

```
      dataset = load_dataset()
      model = nn.Module(dataset.x.size, dataset.y.size)
      theta = model.init(seed=0) # Functional

      for update in range(num_updates):
==>     train_data = dataset.sample()
        theta = train(model, theta, train_data) # Functional
        metrics = evaluate(model, theta, dataset.val_set)
```

The Train Loop

Those familiar with deep learning know of the “training loop"

```
        dataset = load_dataset()
        model = nn.Module(dataset.x.size, dataset.y.size)
        theta = model.init(seed=0) # Functional

        for update in range(num_updates):
          train_data = dataset.sample()
==>       theta = train(model, theta, train_data) # Functional
          metrics = evaluate(model, theta, dataset.val_set)
```

The Train Loop

Those familiar with deep learning know of the "training loop"

```
dataset = load_dataset()
model = nn.Module(dataset.x.size, dataset.y.size)
theta = model.init(seed=0) # Functional

for update in range(num_updates):
  train_data = dataset.sample()
  theta = train(model, theta, train_data) # Functional
==>    metrics = evaluate(model, theta, dataset.val_set)
```

The Train Loop

Deep RL has a training loop similar to the deep learning loop

The Train Loop

Deep RL has a training loop similar to the deep learning loop

```
env = LunarLander()
Q = nn.Module(env.state_space, env.action_space)
theta = Q.init(seed=0)
pi = policy(Q, theta)

for update in range(num_updates):
  collected_data = collect_training_data(env, pi)
  dataset += collected_data
  train_data = dataset.sample()
  theta = train(Q, theta, train_data)
  metrics = evaluate(env, pi)
```

The Environment

Main differences with the DL train loop

```
==>   env = LunarLander()
      Q = nn.Module(env.state_space, env.action_space)
      theta = Q.init(seed=0)
==>   pi = policy(Q, theta)

      for update in range(num_updates):
==>     collected_data = collect_training_data(env, pi)
==>     dataset += collected_data
        train_data = dataset.sample()
        theta = train(Q, theta, train_data)
        metrics = evaluate(env, pi)
```

The Environment

Today: Go through the loop line by line to implement DQN

```
==>   env = LunarLander()
      Q = nn.Module(env.state_space, env.action_space)
      theta = Q.init(seed=0)
==>   pi = policy(Q, theta)

      for update in range(num_updates):
==>     collected_data = collect_training_data(env, pi)
==>     dataset += collected_data
        train_data = dataset.sample()
        theta = train(Q, theta, train_data)
        metrics = evaluate(env, pi)
```

The Environment

Instead of loading a static dataset, we collect data from an environment

```
==>     env = LunarLander()
        Q = nn.Module(env.state_space, env.action_space)
        theta = Q.init(seed=0)
        pi = policy(Q, theta)

        for update in range(num_updates):
          collected_data = collect_training_data(env, pi)
          dataset += collected_data
          train_data = dataset.sample()
          theta = train(Q, theta, train_data)
          metrics = evaluate(env, pi)
```

The Environment

The Plan:

1. Terminal states
2. The Gymnasium interface

The Environment

The Plan:

1. **Terminal states**
2. The Gymnasium interface

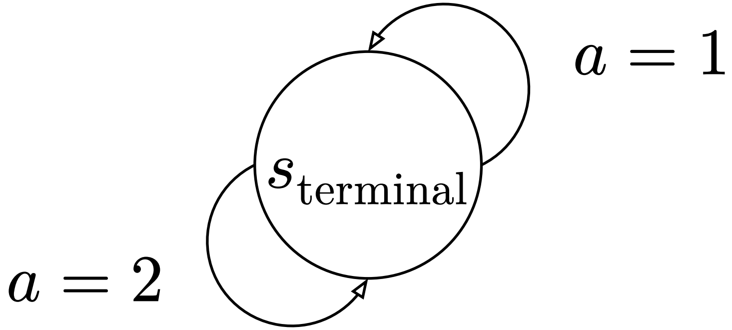
Terminal States

Question: How can we represent a Mario Bros Game Over screen in an MDP?

Terminal States

Question: How can we represent a Mario Bros Game Over screen in an MDP?

Answer: We enter a **terminal state** that we cannot leave



Terminal States

How do we model the return in these terminal states?

Terminal States

How do we model the return in these terminal states?

Recall the discounted return

$$G = \sum_{t=0}^{\infty} \gamma^t r_t$$

Terminal States

How do we model the return in these terminal states?

Recall the discounted return

$$G = \sum_{t=0}^{\infty} \gamma^t r_t$$

After entering the terminal state at $t = n$ all future rewards are zero. We can write the discounted return as

$$G = \sum_{t=0}^{\infty} \gamma^t r_t \cdot (t \leq n) = \sum_{t=0}^{n} \gamma^t r_t$$

Terminal States

$$G = \sum_{t=0}^{\infty} \gamma^t r_t \cdot (t \leq n) = \sum_{t=0}^{n} \gamma^t r_t$$

Terminal States

$$G = \sum_{t=0}^{\infty} \gamma^t r_t \cdot (t \leq n) = \sum_{t=0}^{n} \gamma^t r_t$$

Many environments introduce the **done flag (d)** to simplify data collection and training

Terminal States

$$G = \sum_{t=0}^{\infty} \gamma^t r_t \cdot (t \leq n) = \sum_{t=0}^{n} \gamma^t r_t$$

Many environments introduce the **done flag (d)** to simplify data collection and training

$$\begin{pmatrix} s_0 & s_1 & s_2 & \dots & s_n \\ d_0 = 0 & d_1 = 0 & d_2 = 0 & \dots & d_n = 1 \end{pmatrix}$$

Terminal States

$$G = \sum_{t=0}^{\infty} \gamma^t r_t \cdot (t \leq n) = \sum_{t=0}^{n} \gamma^t r_t$$

Many environments introduce the **done flag (d)** to simplify data collection and training

$$\begin{pmatrix} s_0 & s_1 & s_2 & \dots & s_n \\ d_0 = 0 & d_1 = 0 & d_2 = 0 & \dots & d_n = 1 \end{pmatrix}$$

We call the states from the initial to terminal state an **episode**

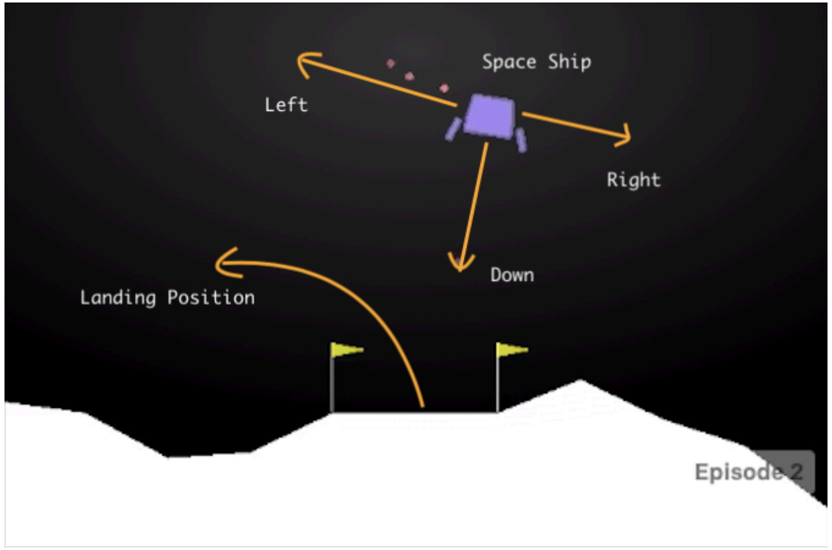
The Environment

The Plan:

1. Terminal states
2. **The Gymnasium interface**

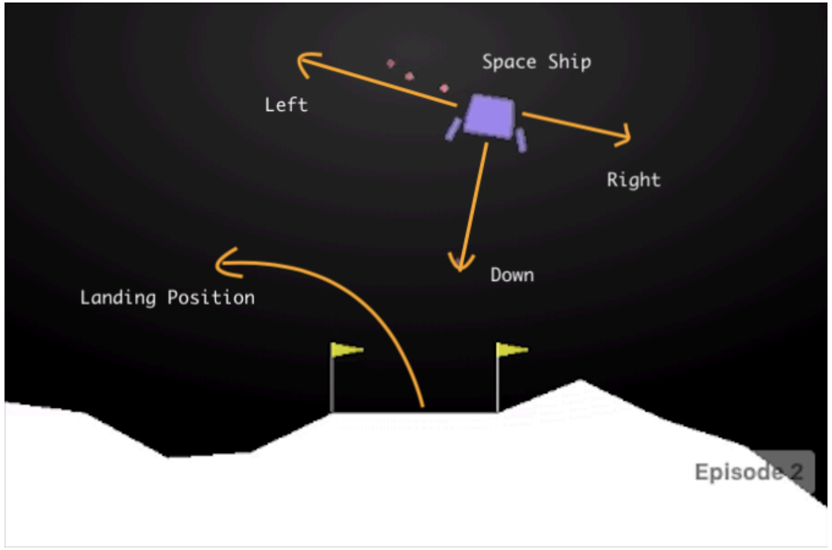
Environment Libraries

The `gymnasium` library contains many popular test environments



Environment Libraries

The `gymnasium` library contains many popular test environments



`gymnasium` also defines the standard environment interface

The Gym Interface

Launching environments is very easy

```
env = gymnasium.make("LunarLander-v2")
```

The Gym Interface

Launching environments is very easy

```
env = gymnasium.make("LunarLander-v2")
```

What is S, A for LunarLander?

```
S, A = env.observation_space, env.action_space
```

The Gym Interface

Launching environments is very easy

```
env = gymnasium.make("LunarLander-v2")
```

What is S, A for LunarLander?

```
S, A = env.observation_space, env.action_space
```

Observations are states that are not guaranteed to be Markov. For LunarLander, they are Markov.

The Gym Interface

```
env = gymnasium.make("LunarLander-v2")
```

The Gym Interface

```
env = gymnasium.make("LunarLander-v2")
```

The environments start "off". We must reset the environment, which returns an initial state.

```
state, _ = env.reset(seed=0)
```

The Gym Interface

```
env = gymnasium.make("LunarLander-v2")
```

The environments start "off". We must reset the environment, which returns an initial state.

```
state, _ = env.reset(seed=0)
```

Step the environment in time by feeding an action (transition function T)

```
next_state, reward, terminated, truncated, _ = env.step(action)
```

The Gym Interface

```
env = gymnasium.make("LunarLander-v2")
```

The environments start "off". We must reset the environment, which returns an initial state.

```
state, _ = env.reset(seed=0)
```

Step the environment in time by feeding an action (transition function T)

```
next_state, reward, terminated, truncated, _ = env.step(action)
```

$$d = \text{terminated} \lor \text{truncated}$$

The Gym Interface

You can write your own environments using `gymnasium`

```
class RoboParking(gymnasium.Env):
  observation_space = spaces.Box(
    # x, y, xdot, ydot
    low=(0, 0, -1, -1),
    high=(4, 4, 1, 1),
    dtype=np.float32
  )
  # left, right, forward, backward
  action_space = spaces.Discrete(4)

  def R(self, pos): # R(s')
    # Our goal is 0,0,0,0
    return norm(sensor.state())
```

```
def T(self, action):
  # We don't know the true T
  wheels.apply_torque(action)
  next_state = sensor.state()
  return next_state

def step(self, action):
  next_state = T(action) # s'
  return (
    next_state,
    R(next_state), # reward R(s')
    norm(next_state) < 0.01 # d
    False, {} # trunc, extra info
  )
```

We discussed the environment

```
==>    env = LunarLander()
       Q = nn.Module(env.state_space, env.action_space)
       theta = Q.init(seed=0)
       pi = policy(Q, theta)

       for update in range(num_updates):
         collected_data = collect_training_data(env, pi)
         dataset += collected_data
         train_data = dataset.sample()
         theta = train(Q, theta, train_data)
         metrics = evaluate(env, pi)
```

Next, let us define the deep Q function

```
      env = LunarLander()
==>   Q = nn.Module(env.state_space, env.action_space)
      theta = Q.init(seed=0)
      pi = policy(Q, theta)

      for update in range(num_updates):
        collected_data = collect_training_data(env, pi)
        dataset += collected_data
        train_data = dataset.sample()
        theta = train(Q, theta, train_data)
        metrics = evaluate(env, pi)
```

The Q Network

Recall the type signature of the Q function $Q : S \times A \to \mathbb{R}$

The Q Network

Recall the type signature of the Q function $Q : S \times A \to \mathbb{R}$

And recall the optimal policy $\pi_*(s) = \text{argmax}_{a \in A} Q_*(s, a, \theta)$

The Q Network

Recall the type signature of the Q function $Q : S \times A \to \mathbb{R}$

And recall the optimal policy $\pi_*(s) = \text{argmax}_{a \in A} Q_*(s, a, \theta)$

We would need to evaluate Q function $|A|$ times for each state

The Q Network

Recall the type signature of the Q function $Q : S \times A \to \mathbb{R}$

And recall the optimal policy $\pi_*(s) = \text{argmax}_{a \in A} Q_*(s, a, \theta)$

We would need to evaluate Q function $|A|$ times for each state

$$Q(s, a = 1)$$
$$Q(s, a = 2)$$
$$\vdots$$

The Q Network

Recall the type signature of the Q function $Q : S \times A \to \mathbb{R}$

And recall the optimal policy $\pi_*(s) = \text{argmax}_{a \in A} Q_*(s, a, \theta)$

We would need to evaluate Q function $|A|$ times for each state

$$Q(s, a = 1)$$
$$Q(s, a = 2)$$
$$\vdots$$

Inefficient: $|A| = 100$ means 100 forward passes for each timestep

The Q Network

Recall the type signature of the Q function $Q : S \times A \to \mathbb{R}$

And recall the optimal policy $\pi_*(s) = \operatorname{argmax}_{a \in A} Q_*(s, a, \theta)$

We instead represent the Q network as

$$Q : S \to \mathbb{R}^{|A|}$$

The Q Network

Recall the type signature of the Q function $Q : S \times A \to \mathbb{R}$

And recall the optimal policy $\pi_*(s) = \text{argmax}_{a \in A} Q_*(s, a, \theta)$

We instead represent the Q network as

$$Q : S \to \mathbb{R}^{|A|}$$

Compute the Q value for all actions in a single forward pass

The Q Network

Recall the type signature of the Q function $Q : S \times A \to \mathbb{R}$

And recall the optimal policy $\pi_*(s) = \text{argmax}_{a \in A} Q_*(s, a, \theta)$

We instead represent the Q network as

$$Q : S \to \mathbb{R}^{|A|}$$

Compute the Q value for all actions in a single forward pass

$$\begin{aligned} Q(s, \theta) = & \; Q(s, a = 1, \theta), \\ & \; Q(s, a = 2, \theta), \\ & \; \vdots \end{aligned}$$

The Q Network

We instead represent the Q network as

$$Q : S \to \mathbb{R}^{|A|}$$

The Q Network

We instead represent the Q network as

$$Q : S \to \mathbb{R}^{|A|}$$

Architecture: 2 layer MLP with hidden size of 256 is sufficient for standard benchmarks

The Q Network

We instead represent the Q network as

$$Q : S \to \mathbb{R}^{|A|}$$

Architecture: 2 layer MLP with hidden size of 256 is sufficient for standard benchmarks

```
Q = Sequential(
  Linear(state_size, 256), LeakyReLU(),
  Linear(256, 256), LeakyReLU(),
  Linear(256, action_size)
)
```

The Q Network

We instead represent the Q network as

$$Q : S \to \mathbb{R}^{|A|}$$

Architecture: 2 layer MLP with hidden size of 256 is sufficient for standard benchmarks

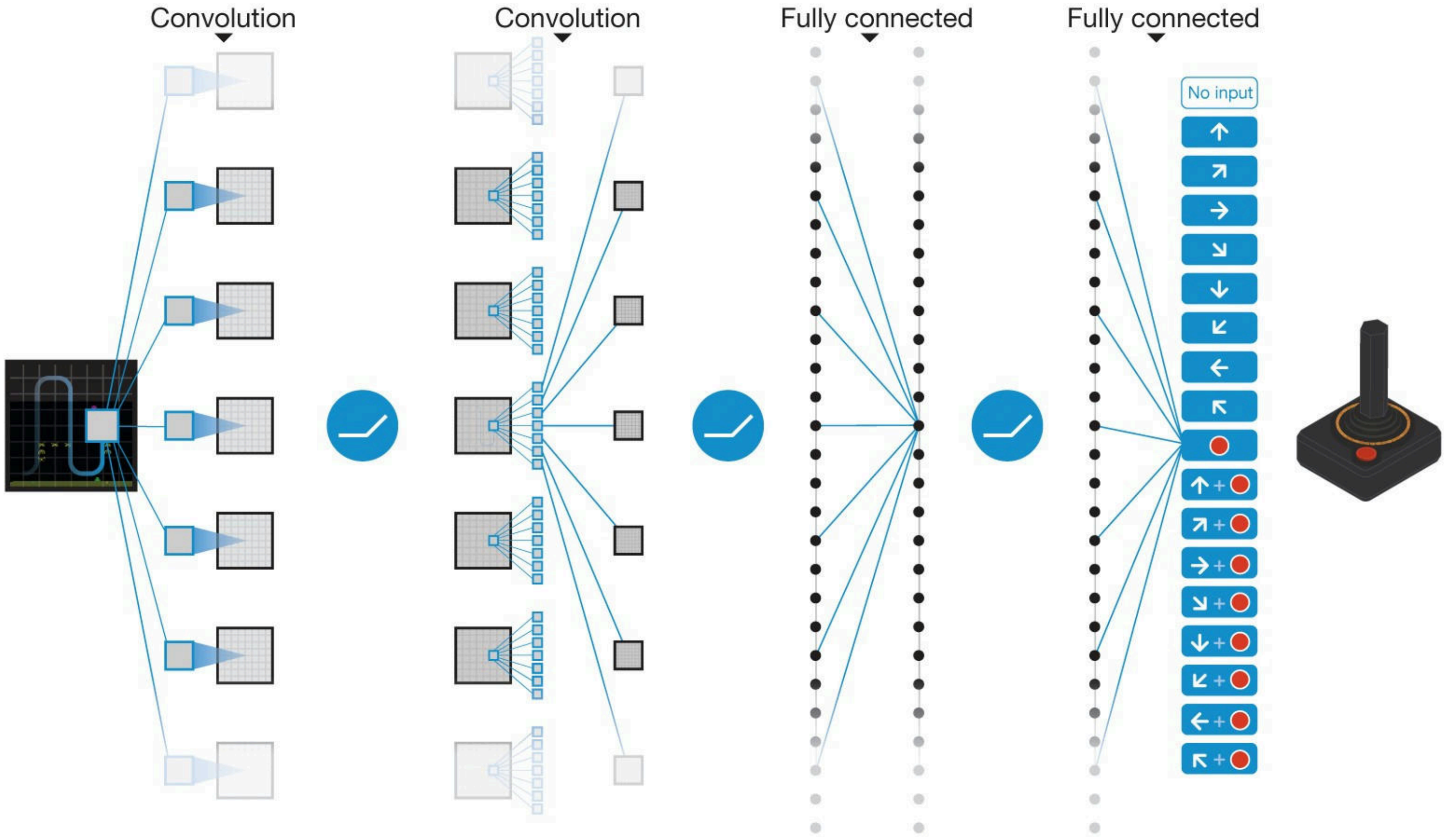
```
Q = Sequential(
  Linear(state_size, 256), LayerNorm(), LeakyReLU(),
  Linear(256, 256), LayerNorm(), LeakyReLU(),
  Linear(256, action_size)
)
```

The Q Network

For states with structure (e.g., pixels), prepend encoders to Q

The Q Network

For states with structure (e.g., pixels), prepend encoders to Q



Init

Done with the Q function architecture, let's discuss init

```
      env = LunarLander()
==>   Q = nn.Module(env.state_space, env.action_space)
      theta = Q.init(seed=0)
      pi = policy(Q, theta)

      for update in range(num_updates):
        collected_data = collect_training_data(env, pi)
        dataset += collected_data
        train_data = dataset.sample()
        theta = train(Q, theta, train_data)
        metrics = evaluate(env, pi)
```

Init

Done with the Q function architecture, let's discuss init

```
        env = LunarLander()
        Q = nn.Module(env.state_space, env.action_space)
==>     theta = Q.init(seed=0)
        pi = policy(Q, theta)

        for update in range(num_updates):
          collected_data = collect_training_data(env, pi)
          dataset += collected_data
          train_data = dataset.sample()
          theta = train(Q, theta, train_data)
          metrics = evaluate(env, pi)
```

The Q Network

Not much to say for parameter initialization

The Q Network

Not much to say for parameter initialization

Tip: Initialize the final layer of your Q function to output values near 0

The Q Network

Not much to say for parameter initialization

Tip: Initialize the final layer of your Q function to output values near 0

```
nn.init.normal(std=1e-3, bias=0)
```

The Q Network

Not much to say for parameter initialization

Tip: Initialize the final layer of your Q function to output values near 0

```
nn.init.normal(std=1e-3, bias=0)
```

Prevents Q value overestimation (to be discussed in depth later)

Policy

Done with init, let's discuss policy π

```
        env = LunarLander()
        Q = nn.Module(env.state_space, env.action_space)
==>     theta = Q.init(seed=0)
        pi = policy(Q, theta)

        for update in range(num_updates):
          collected_data = collect_training_data(env, pi)
          dataset += collected_data
          train_data = dataset.sample()
          theta = train(Q, theta, train_data)
          metrics = evaluate(env, pi)
```

Policy

Done with init, let's discuss policy π

```
       env = LunarLander()
       Q = nn.Module(env.state_space, env.action_space)
       theta = Q.init(seed=0)
==>    pi = policy(Q, theta)

       for update in range(num_updates):
         collected_data = collect_training_data(env, pi)
         dataset += collected_data
         train_data = dataset.sample()
         theta = train(Q, theta, train_data)
         metrics = evaluate(env, pi)
```

Policy

Recall the policy $\pi : S \to \Delta A$

Policy

Recall the policy $\pi : S \to \Delta A$

In practice, we usually have two policies

Policy

Recall the policy $\pi : S \to \Delta A$

In practice, we usually have two policies

1. The optimal policy we are trying to learn (π)

Policy

Recall the policy $\pi : S \rightarrow \Delta A$

In practice, we usually have two policies

1. The optimal policy we are trying to learn (π)
2. An **exploration policy** ($\boldsymbol{\pi_E}$) for collecting training data

Policy

Recall the policy $\pi : S \to \Delta A$

In practice, we usually have two policies

1. The optimal policy we are trying to learn (π)
2. An **exploration policy** ($\boldsymbol{\pi_E}$) for collecting training data

Why do we need π_E?

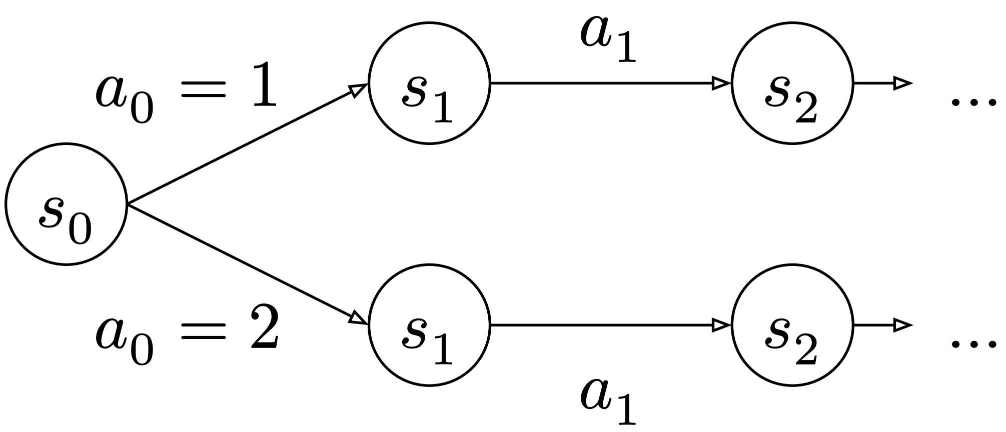
Policy

Recall the policy $\pi : S \to \Delta A$

In practice, we usually have two policies

1. The optimal policy we are trying to learn (π)
2. An **exploration policy (π_E)** for collecting training data

Why do we need π_E?



Exploration Policy

We use the **exploration policy** (π_E) to explore and collect data

Exploration Policy

We use the **exploration policy** (π_E) to explore and collect data

We will use our collected data to train the Q function. What properties should our collected data have?

Exploration Policy

We use the **exploration policy** (π_E) to explore and collect data

We will use our collected data to train the Q function. What properties should our collected data have?

1. To ensure Q is accurate everywhere, take every possible action in every possible state

Exploration Policy

We use the **exploration policy (π_E)** to explore and collect data

We will use our collected data to train the Q function. What properties should our collected data have?

1. To ensure Q is accurate everywhere, take every possible action in every possible state

Given these requirements, we cannot do better than random exploration

$$\pi_E(s) = \mathcal{U}(A)$$

Exploration Policy

$$\pi_E(s) = \mathcal{U}(A)$$

Question: Are there any downsides to this exploration policy?

Exploration Policy

$$\pi_E(s) = \mathcal{U}(A)$$

Question: Are there any downsides to this exploration policy?

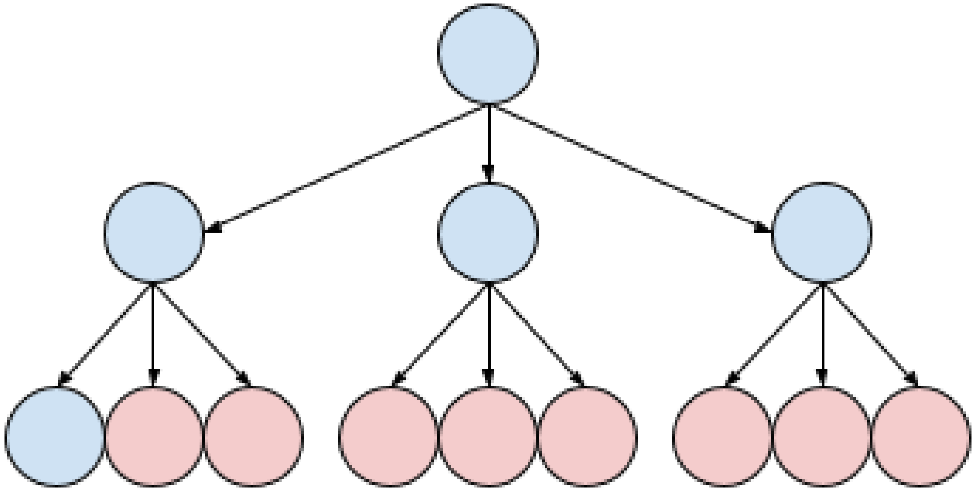
Answer: It could take a really, really long time

Exploration Policy

$$\pi_E(s) = \mathcal{U}(A)$$

Question: Are there any downsides to this exploration policy?

Answer: It could take a really, really long time



Exploration Policy

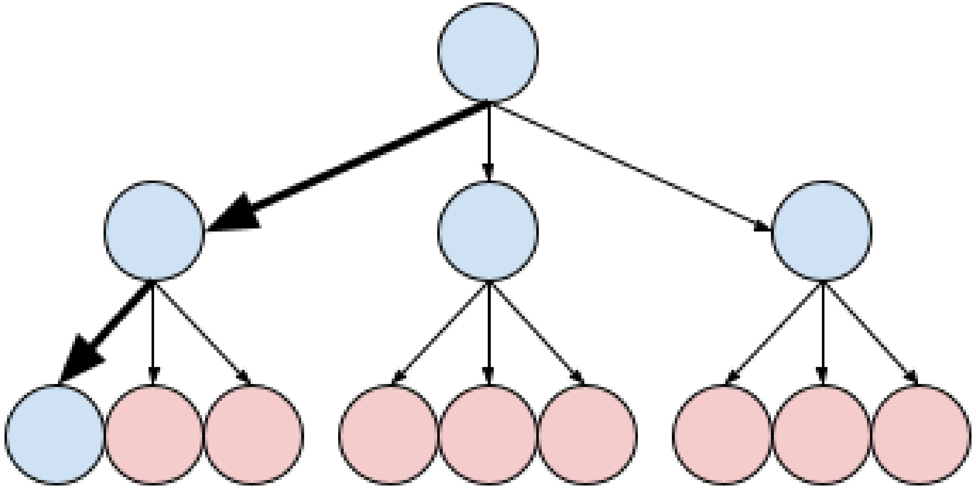
$$\pi_E(s) = \mathcal{U}(A)$$

Alternative: Bias the policy towards states with known large Q values

Exploration Policy

$$\pi_E(s) = \mathcal{U}(A)$$

Alternative: Bias the policy towards states with known large Q values



Exploration Policy

$$\pi_E(s) = \mathcal{U}(A)$$

Alternative: Bias the policy towards states with known large Q values

One approach is the $\boldsymbol{\epsilon}$**-greedy** policy

Exploration Policy

$$\pi_E(s) = \mathcal{U}(A)$$

Alternative: Bias the policy towards states with known large Q values

One approach is the **ϵ-greedy** policy

$$\epsilon \in [0, 1]$$

$$u \sim \mathcal{U}[0, 1]$$

$$\pi_E(s) = \begin{cases} \mathcal{U}(A) \text{ if } u \leq \epsilon \\ \text{argmax}_{a \in A} Q(s, a) \text{ if } u > \epsilon \end{cases}$$

Exploration Policy

$$\pi_E(s) = \mathcal{U}(A)$$

Alternative: Bias the policy towards states with known large Q values

One approach is the **ϵ-greedy** policy

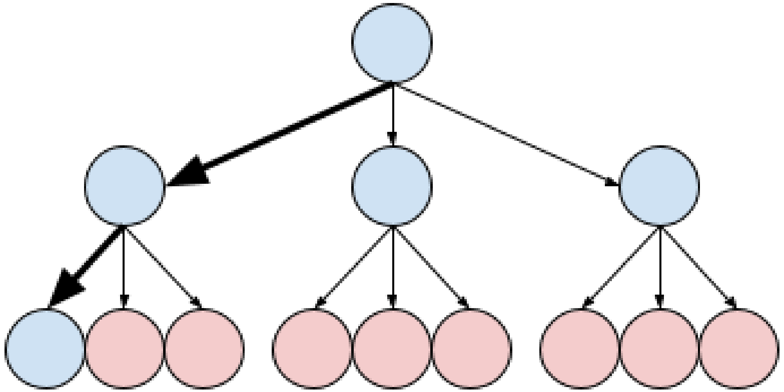
$$\epsilon \in [0, 1]$$

$$u \sim \mathcal{U}[0, 1]$$

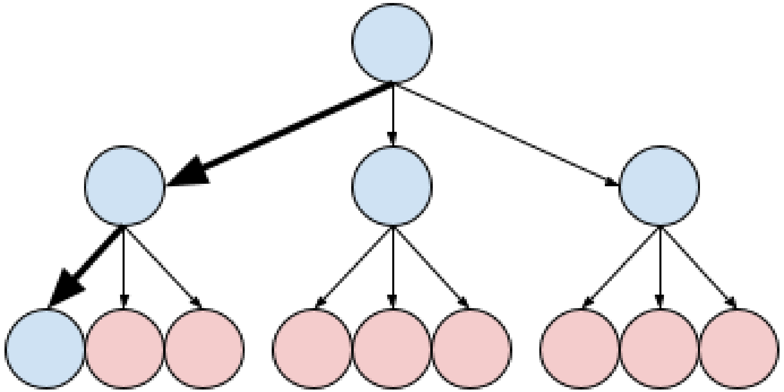
$$\pi_E(s) = \begin{cases} \mathcal{U}(A) \text{ if } u \leq \epsilon \\ \text{argmax}_{a \in A} Q(s, a) \text{ if } u > \epsilon \end{cases}$$

This approach is simple and works surprisingly well in practice

Exploration Policy

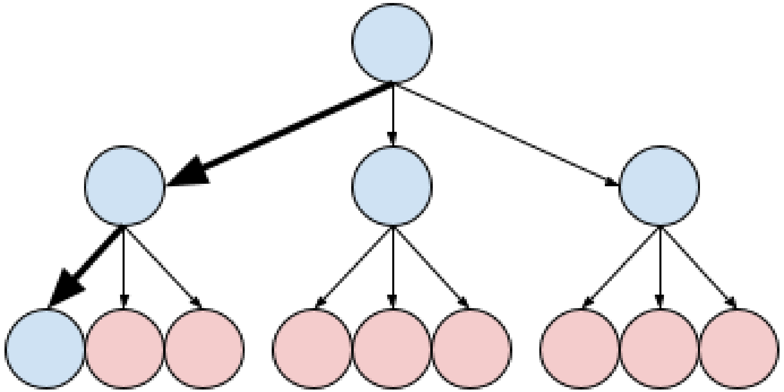


Exploration Policy



Explore promising areas with large Q values more often and as $t \to \infty$, explore all state/action tuples

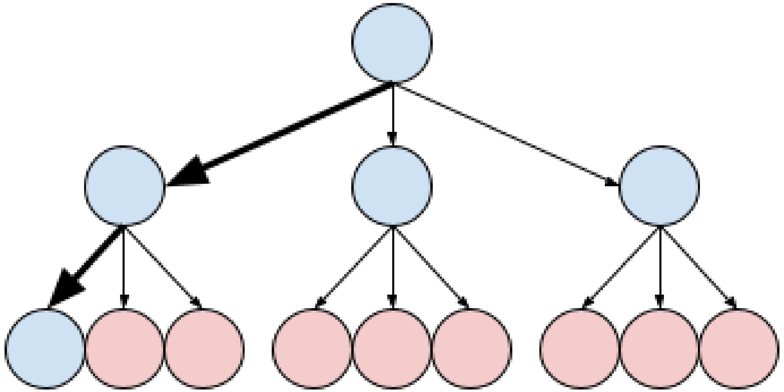
Exploration Policy



Explore promising areas with large Q values more often and as $t \to \infty$, explore all state/action tuples

Question: Any downsides to the ε-greedy approach?

Exploration Policy



Question: Any downsides to the ε-greedy approach?

Answer: Not independently distributed. The state/action distribution will be biased by the Q function. Seems to be ignored in practice?

Exploration Policy

Summary: Maintain two policies

Exploration Policy

Summary: Maintain two policies

π: The policy that approximates π_*

Exploration Policy

Summary: Maintain two policies

π: The policy that approximates π_*

π_E: A stochastic policy used for exploring the environment and collecting data

Let us make a small change to the pseudocode

```
        env = LunarLander()
        Q = nn.Module(env.state_space, env.action_space)
        theta = Q.init(seed=0)
==>     pi = policy(Q, theta)

        for update in range(num_updates):
          collected_data = collect_training_data(env, pi)
          dataset += collected_data
          train_data = dataset.sample()
          theta = train(Q, theta, train_data)
          metrics = evaluate(env, pi)
```

Now we have two policies, one for collection and one for evaluation

```
        env = LunarLander()
        Q = nn.Module(env.state_space, env.action_space)
        theta = Q.init(seed=0)
==>     pi, pi_e = max_q(Q, theta), e_greedy(Q, theta)

        for update in range(num_updates):
==>       collected_data = collect_training_data(env, pi_e)
          dataset += collected_data
          train_data = dataset.sample()
          theta = train(Q, theta, train_data)
          metrics = evaluate(env, pi)
```

Let’s move onto data collection

```
      env = LunarLander()
      Q = nn.Module(env.state_space, env.action_space)
      theta = Q.init(seed=0)
      pi, pi_e = max_q(Q, theta), e_greedy(Q, theta)

      for update in range(num_updates):
==>     collected_data = collect_training_data(env, pi_e)
        dataset += collected_data
        train_data = dataset.sample()
        theta = train(Q, theta, train_data)
        metrics = evaluate(env, pi)
```

Collectors

We must interact with the MDP to collect training data

Collectors

We must interact with the MDP to collect training data

Recall the Q learning objective

$$\min_{\theta} \left(Q(\underline{s}, \underline{a}, \theta) - \left(\underline{r} + \gamma \cdot \underset{\{a' \in A\}}{\mathrm{argmax}}\, Q(\underline{s'}, a', \theta) \right) \right)^2$$

Collectors

We must interact with the MDP to collect training data

Recall the Q learning objective

$$\min_{\theta} \left(Q(\underline{s}, \underline{a}, \theta) - \left(\underline{r} + \gamma \cdot \underset{\{a' \in A\}}{\text{argmax}} \, Q(\underline{s'}, a', \theta) \right) \right)^2$$

Many algorithms train using a **transition tuple** (s, a, r, s', d)

Collectors

Collecting transitions correctly is deceptively tricky (off by one errors)

Collectors

Collecting transitions correctly is deceptively tricky (off by one errors)

```
states, next_states, rewards, actions, dones = [], [], ...
s, _ = env.reset(seed=0)
d = False
while not d:
  a = pi_e(s, theta)
  next_s, r, trunc, term, _ = env.step(action) # r = R(s')
  d = trunc or term
  states.append(s), next_states.append(next_s), rewards...
  s = next_s
# n+1 states total, but each list should be len n
episode = (states, next_states, rewards, actions, dones)
return episode
```

Replay Buffers

We call the dataset a **replay buffer** ($\mathcal{D}$)

Replay Buffers

We call the dataset a **replay buffer ($\mathcal{D}$)**

$$\mathcal{D} = \begin{bmatrix} (s_0, a_0, r_0, s'_0, d_0) \\ (s_1, a_1, r_1, s'_1, d_1) \\ \vdots \end{bmatrix}$$

Replay Buffers

We call the dataset a **replay buffer ($\mathcal{D}$)**

$$\mathcal{D} = \begin{bmatrix} (s_0, a_0, r_0, s_0', d_0) \\ (s_1, a_1, r_1, s_1', d_1) \\ \vdots \end{bmatrix}$$

```
buffer = []
transitions = [(s_0, a_0, r_0, next_s_0), ...]
buffer += transitions
```

Replay Buffers

We call the dataset a **replay buffer ($\mathcal{D}$)**

$$\mathcal{D} = \begin{bmatrix} (s_0, a_0, r_0, s_0', d_0) \\ (s_1, a_1, r_1, s_1', d_1) \\ \vdots \end{bmatrix}$$

```
buffer = []
transitions = [(s_0, a_0, r_0, next_s_0), ...]
buffer += transitions
```

Note: We often enforce a max size of $\mathcal{D}$ using a ring buffer

Replay Buffers

We populated the dataset, now let's sample from it

```
    env = LunarLander()
    Q = nn.Module(env.state_space, env.action_space)
    theta = Q.init(seed=0)
    pi, pi_e = max_q(Q, theta), e_greedy(Q, theta)

    for update in range(num_updates):
      collected_data = collect_training_data(env, pi_e)
==>   dataset += collected_data
      train_data = dataset.sample()
      theta = train(Q, theta, train_data)
      metrics = evaluate(env, pi)
```

Replay Buffers

We populated the dataset, now let's sample from it

```
      env = LunarLander()
      Q = nn.Module(env.state_space, env.action_space)
      theta = Q.init(seed=0)
      pi, pi_e = max_q(Q, theta), e_greedy(Q, theta)

      for update in range(num_updates):
        collected_data = collect_training_data(env, pi_e)
        dataset += collected_data
==>     train_data = dataset.sample()
        theta = train(Q, theta, train_data)
        metrics = evaluate(env, pi)
```

Replay Buffers

We sample training data from $\mathcal{D}$

Replay Buffers

We sample training data from $\mathcal{D}$

We will call the train data our training batch $\mathcal{B}$

$$\mathcal{B} \sim [\mathcal{U}(\mathcal{D}), ..., \mathcal{U}(\mathcal{D})]$$

Replay Buffers

We sample training data from $\mathcal{D}$

We will call the train data our training batch $\mathcal{B}$

$$\mathcal{B} \sim [\mathcal{U}(\mathcal{D}), ..., \mathcal{U}(\mathcal{D})]$$

$$\mathcal{B} = \left[\left(s_j, a_j, r_j, s'_j, d_j\right), ..., (s_k, a_k, r_k, s'_k, d_k)\right]$$

Replay Buffers

We sample training data from $\mathcal{D}$

We will call the train data our training batch $\mathcal{B}$

$$\mathcal{B} \sim [\mathcal{U}(\mathcal{D}), ..., \mathcal{U}(\mathcal{D})]$$

$$\mathcal{B} = [(s_j, a_j, r_j, s'_j, d_j), ..., (s_k, a_k, r_k, s'_k, d_k)]$$

Randomly sampling old data helps mitigate correlations between data, improving training stability

Replay Buffers

Randomly sampling old data helps mitigate correlations between data, improving training stability

Replay Buffers

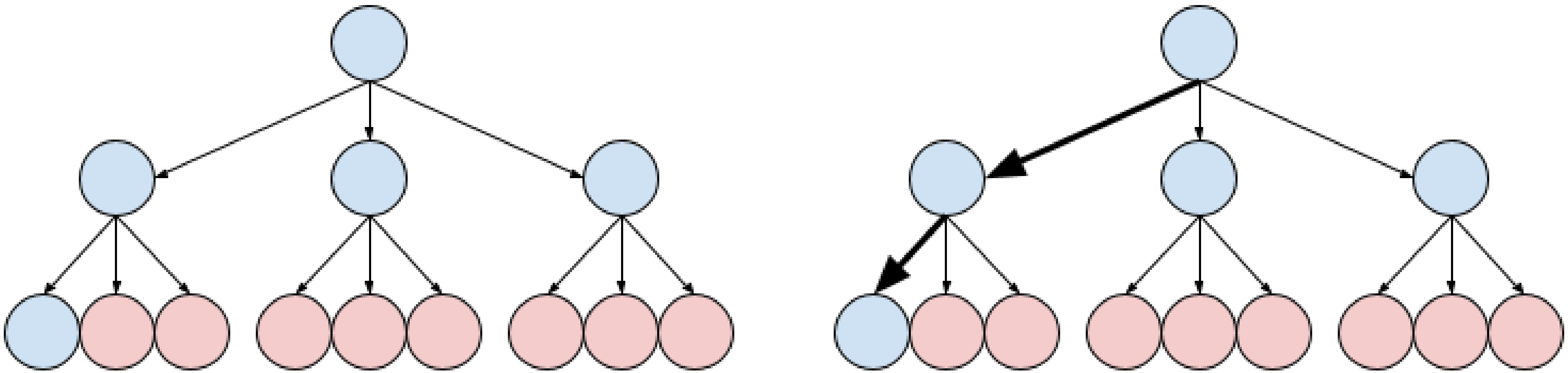
Randomly sampling old data helps mitigate correlations between data, improving training stability

Biased towards many prior policies instead of one

Replay Buffers

Randomly sampling old data helps mitigate correlations between data, improving training stability

Biased towards many prior policies instead of one



Loss Function

```
        env = LunarLander()
        Q = nn.Module(env.state_space, env.action_space)
        theta = Q.init(seed=0)
        pi, pi_e = max_q(Q, theta), e_greedy(Q, theta)

        for update in range(num_updates):
          collected_data = collect_training_data(env, pi_e)
          dataset += collected_data
==>       train_data = dataset.sample()
          theta = train(Q, theta, train_data)
          metrics = evaluate(env, pi)
```

Loss Function

```
        env = LunarLander()
        Q = nn.Module(env.state_space, env.action_space)
        theta = Q.init(seed=0)
        pi, pi_e = max_q(Q, theta), e_greedy(Q, theta)

        for update in range(num_updates):
          collected_data = collect_training_data(env, pi_e)
          dataset += collected_data
          train_data = dataset.sample()
==>       theta = train(Q, theta, train_data)
          metrics = evaluate(env, pi)
```

Loss Function

We make a few modifications to the Q learning objective to improve performance

Loss Function

We make a few modifications to the Q learning objective to improve performance

1. Early termination using d

Loss Function

We make a few modifications to the Q learning objective to improve performance

1. Early termination using d
2. Target networks

Loss Function

We make a few modifications to the Q learning objective to improve performance

1. **Early termination using d**
2. Target networks

Loss Function

Recall the standard Q learning objective

$$\min_{\theta} \left(Q(s, a, \theta) - \left(r + \gamma \cdot \max_{\{a' \in A\}} Q(s', a', \theta) \right) \right)^2$$

Loss Function

Recall the standard Q learning objective

$$\min_{\theta} \left(Q(s, a, \theta) - \left(r + \gamma \cdot \max_{\{a' \in A\}} Q(s', a', \theta) \right) \right)^2$$

Rather than learn to output 0 at terminal states, we modify the objective

$$\min_{\theta} \left(Q(s, a, \theta) - \left(r + \neg \boldsymbol{d} \cdot \gamma \cdot \max_{\{a' \in A\}} Q(s', a', \theta) \right) \right)^2$$

Loss Function

Recall the standard Q learning objective

$$\min_{\theta} \left(Q(s, a, \theta) - \left(r + \gamma \cdot \max_{\{a' \in A\}} Q(s', a', \theta) \right) \right)^2$$

Rather than learn to output 0 at terminal states, we modify the objective

$$\min_{\theta} \left(Q(s, a, \theta) - \left(r + \neg \boldsymbol{d} \cdot \gamma \cdot \max_{\{a' \in A\}} Q(s', a', \theta) \right) \right)^2$$

For terminal transitions, this reduces to

$$\min_{\theta} \left(Q(s, a, \theta) - r \right)^2$$

Loss Function

We make a few modifications to the Q learning objective to improve performance

1. Early termination using d
2. **Target networks**

Loss Function

Traditional Q learning used a table

	$s = 0$	$s = 1$	$s = 2$
$a = 0$	1	3	2
$a = 1$	7	4	5
$a = 2$	1	0	0

Loss Function

Traditional Q learning used a table

	$s = 0$	$s = 1$	$s = 2$
$a = 0$	1	3	2
$a = 1$	7	4	5
$a = 2$	1	0	0

Updates are very well defined

	$s = 0$	$s = 1$	$s = 2$
$a = 0$	**5**	3	2
$a = 1$	7	4	5
$a = 2$	1	0	0

Loss Function

Traditional Q learning used a table

	$s=0$	$s=1$	$s=2$
$a=0$	1	3	2
$a=1$	7	4	5
$a=2$	1	0	0

Updates are very well defined

	$s=0$	$s=1$	$s=2$
$a=0$	**5**	3	2
$a=1$	7	4	5
$a=2$	1	0	0

Neural networks are different. Increasing a single $(s=0, a=0)$ entry will often **perturb** the Q value for all states and actions.

	$s=0$	$s=1$	$s=2$
$a=0$	**5**	**4**	**4**
$a=1$	**9**	**5**	5
$a=2$	**4**	0	0

Loss Function

These perturbations ε ripple through the Q recursion, with the max operator resulting in overestimation

$$Q(s, a, \boldsymbol{\theta}') \leftarrow r + \gamma \max_{a' \in A} [Q(s', a', \theta) + \varepsilon_\theta]$$

Loss Function

These perturbations ε ripple through the Q recursion, with the max operator resulting in overestimation

$$Q(s, a, \boldsymbol{\theta}') \leftarrow r + \gamma \max_{a' \in A} [Q(s', a', \theta) + \varepsilon_\theta]$$

$$Q(s, a, \boldsymbol{\theta}'') \leftarrow r + \gamma \left(\max_{a' \in A} [Q(s', a', \boldsymbol{\theta}') + \varepsilon_\theta + \varepsilon_{\theta'}] \right)$$

Loss Function

These perturbations ε ripple through the Q recursion, with the max operator resulting in overestimation

$$Q(s, a, \boldsymbol{\theta}') \leftarrow r + \gamma \max_{a' \in A}[Q(s', a', \theta) + \varepsilon_\theta]$$

$$Q(s, a, \boldsymbol{\theta}'') \leftarrow r + \gamma \left(\max_{a' \in A}[Q(s', a', \boldsymbol{\theta}') + \varepsilon_\theta + \varepsilon_{\theta'}] \right)$$

$$Q(s, a, \theta''') \leftarrow r + \gamma \left(\max_{a' \in A}[Q(s', a', \boldsymbol{\theta}'') + \varepsilon_\theta + \varepsilon_{\theta'} + \varepsilon_{\theta''}] \right)$$

Loss Function

These perturbations ε ripple through the Q recursion, with the max operator resulting in overestimation

$$Q(s, a, \boldsymbol{\theta}') \leftarrow r + \gamma \max_{a' \in A} [Q(s', a', \theta) + \varepsilon_\theta]$$

$$Q(s, a, \boldsymbol{\theta}'') \leftarrow r + \gamma \left(\max_{a' \in A} [Q(s', a', \boldsymbol{\theta}') + \varepsilon_\theta + \varepsilon_{\theta'}] \right)$$

$$Q(s, a, \theta''') \leftarrow r + \gamma \left(\max_{a' \in A} [Q(s', a', \boldsymbol{\theta}'') + \varepsilon_\theta + \varepsilon_{\theta'} + \varepsilon_{\theta''}] \right)$$

Compounding pertubations combined with the max operator result in exploding Q values (i.e., $Q(\cdot, \cdot) = \infty$)

Loss Function

Compounding pertubations combined with the max operator result in exploding Q values (i.e., $Q(\cdot, \cdot) = \infty$)

Loss Function

Compounding pertubations combined with the max operator result in exploding Q values (i.e., $Q(\cdot, \cdot) = \infty$)

Solution 1: Constrained optimization of neural networks (hard)

Loss Function

Compounding pertubations combined with the max operator result in exploding Q values (i.e., $Q(\cdot, \cdot) = \infty$)

Solution 1: Constrained optimization of neural networks (hard)

Solution 2: Very large batch sizes that cover all (s, a) (intractable)

Loss Function

Compounding pertubations combined with the max operator result in exploding Q values (i.e., $Q(\cdot, \cdot) = \infty$)

Solution 1: Constrained optimization of neural networks (hard)

Solution 2: Very large batch sizes that cover all (s, a) (intractable)

Solution 3: Surrogate **target network** to break recurrence (easy)

Loss Function

Solution 3: Surrogate **target network** to break recurrence (easy)

Initialize target parameters $\psi = \theta$

$$Q(s, a, \theta') \leftarrow r + \gamma \max_{a' \in A} \left[Q(s', a', \psi) + \varepsilon_\psi \right]$$

Loss Function

Solution 3: Surrogate **target network** to break recurrence (easy)

Initialize target parameters $\psi = \theta$

$$Q(s, a, \theta') \leftarrow r + \gamma \max_{a' \in A} \left[Q(s', a', \psi) + \varepsilon_\psi \right]$$

$$Q(s, a, \theta'') \leftarrow r + \gamma \left(\max_{a' \in A} \left[Q(s', a', \psi) + \varepsilon_\psi \right] \right)$$

Loss Function

Solution 3: Surrogate **target network** to break recurrence (easy)

Initialize target parameters $\psi = \theta$

$$Q(s, a, \theta') \leftarrow r + \gamma \max_{a' \in A} \left[Q(s', a', \psi) + \varepsilon_\psi \right]$$

$$Q(s, a, \theta'') \leftarrow r + \gamma \left(\max_{a' \in A} \left[Q(s', a', \psi) + \varepsilon_\psi \right] \right)$$

$$Q(s, a, \theta''') \leftarrow r + \gamma \left(\max_{a' \in A} \left[Q(s', a', \psi) + \varepsilon_\psi \right] \right)$$

Loss Function

Solution 3: Surrogate **target network** to break recurrence (easy)

Initialize target parameters $\psi = \theta$

$$Q(s, a, \theta') \leftarrow r + \gamma \max_{a' \in A} \left[Q(s', a', \psi) + \varepsilon_\psi \right]$$

$$Q(s, a, \theta'') \leftarrow r + \gamma \left(\max_{a' \in A} \left[Q(s', a', \psi) + \varepsilon_\psi \right] \right)$$

$$Q(s, a, \theta''') \leftarrow r + \gamma \left(\max_{a' \in A} \left[Q(s', a', \psi) + \varepsilon_\psi \right] \right)$$

After a while, set $\psi = \theta$ again

Loss Function

Behold, the combination of **early termination** and **target networks**

$$Q(s, a, \theta) = r + \neg \boldsymbol{d} \cdot \gamma \cdot \max_{a' \in A} Q(s', a', \boldsymbol{\psi})$$

Loss Function

Behold, the combination of **early termination** and **target networks**

$$Q(s, a, \theta) = r + \neg \boldsymbol{d} \cdot \gamma \cdot \max_{a' \in A} Q(s', a', \boldsymbol{\psi})$$

$$Q(s, a, \theta) - \left(r + \neg \boldsymbol{d} \cdot \gamma \cdot \max_{a' \in A} Q(s', a', \boldsymbol{\psi}) \right) = 0$$

Loss Function

Behold, the combination of **early termination** and **target networks**

$$Q(s, a, \theta) = r + \neg \boldsymbol{d} \cdot \gamma \cdot \max_{a' \in A} Q(s', a', \psi)$$

$$Q(s, a, \theta) - \left(r + \neg \boldsymbol{d} \cdot \gamma \cdot \max_{a' \in A} Q(s', a', \psi) \right) = 0$$

The standard objective for DQN is

$$\min_{\theta} \left(Q(s, a, \theta) - \left(r + \neg \boldsymbol{d} \cdot \gamma \cdot \max_{a' \in A} Q(s', a', \psi) \right) \right)^2$$

Loss Function

We need to make a few small updates given our new objective

```
        env = LunarLander()
        Q = nn.Module(env.state_space, env.action_space)
        theta = Q.init(seed=0)
        pi, pi_e = max_q(Q, theta), e_greedy(Q, theta)

        for update in range(num_updates):
          collected_data = collect_training_data(env, pi_e)
          dataset += collected_data
          train_data = dataset.sample()
==>       theta = train(Q, theta, train_data)
          metrics = evaluate(env, pi)
```

Loss Function

Initialize target parameters, and use target params in loss function

```
        env = LunarLander()
        Q = nn.Module(env.state_space, env.action_space)
==>     theta, psi = Q.init(seed=0), Q.init(seed=0)
        pi, pi_e = max_q(Q, theta), e_greedy(Q, theta)

        for update in range(num_updates):
          collected_data = collect_training_data(env, pi_e)
          dataset += collected_data
          train_data = dataset.sample()
==>       theta = train(Q, theta, psi, train_data)
          metrics = evaluate(env, pi)
```

Evaluation

Same as `collect_training_data` but use π not π_E

```
      env = LunarLander()
      Q = nn.Module(env.state_space, env.action_space)
      theta, psi = Q.init(seed=0), Q.init(seed=0)
      pi, pi_e = max_q(Q, theta), e_greedy(Q, theta)

      for update in range(num_updates):
        collected_data = collect_training_data(env, pi_e)
        dataset += collected_data
        train_data = dataset.sample()
        theta = train(Q, theta, psi, train_data)
==>     metrics = evaluate(env, pi)
```

Summary

- Review
- State of the field
- Implement Deep Q Networks (DQN) (Mnih et al.)

Next Time

- Zach, Riccardo, Grace, Dylan, and Saksham will be lecturing
  1. Give us a hint on the topic!
  2. Turn in reports (email is best)
  3. 10 min presentation + 5 min questions and discussion
     - Next lecture is not recorded (reduce anxiety)

- Miniproject handout
  - Moodle says due 22 March 16:00