MODULE 11: INTRODUCTION TO DRL

BA713 - Machine Learning & AI

CONTENTS

CLASSES OF LEARNING PROBLEMS

Supervised Learning

- Given features + labels
- Goal: learn to identify or map the features to labels
- **Example:** Identify oranges from apples

Unsupervised Learning

- Given features only, no labels
- Goal: learn underlying relationships and representation
- **Example:** Apples look similar to oranges, so they can be clustered in the same group as fruits

Reinforcement Learning

- Given state and action pairs
- Goal: maximize future rewards using trail and error
- **Example:** Eat apples or oranges (food) because it can keep you alive

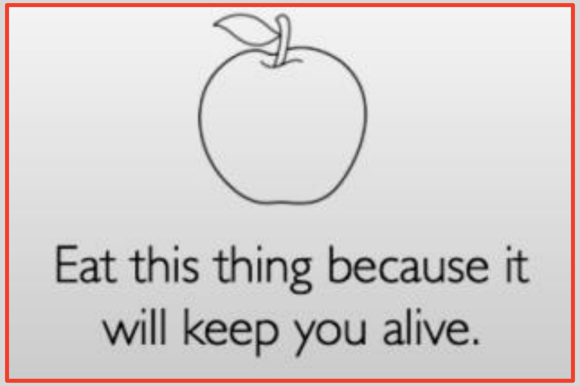


Image Source: introtodeeplearning.com

BACKGROUND

- Reinforcement Learning is the task of learning through trial and error with a final goal to take an action

Deep Reinforcement Learning (DRL)→

- Deep Learning (DL) + Reinforcement Learning (RL)
- Ability of a Deep Neural Network to represent and understand the environment
- Ability of RL to take actions based on its understanding
- unstructured environment, large amounts of data, discover patterns
- Recognition problem

Source: https://towardsdatascience.com/drl-01-a-gentle-introduction-to-deep-reinforcement-learning-405b79866bf4

REINFORCEMENT LEARNING

- **Learning by interacting with the environment**
- How a child or infant learns
- Interactions with the environment→ useful information
- Different from supervised learning where the algorithm is told what actions to take
- trial and error
- Discover the actions with the highest reward by itself
- Decision making problem

Examples:

- learning to drive a car, aware of environment, take actions
- The tic-tac-toe game

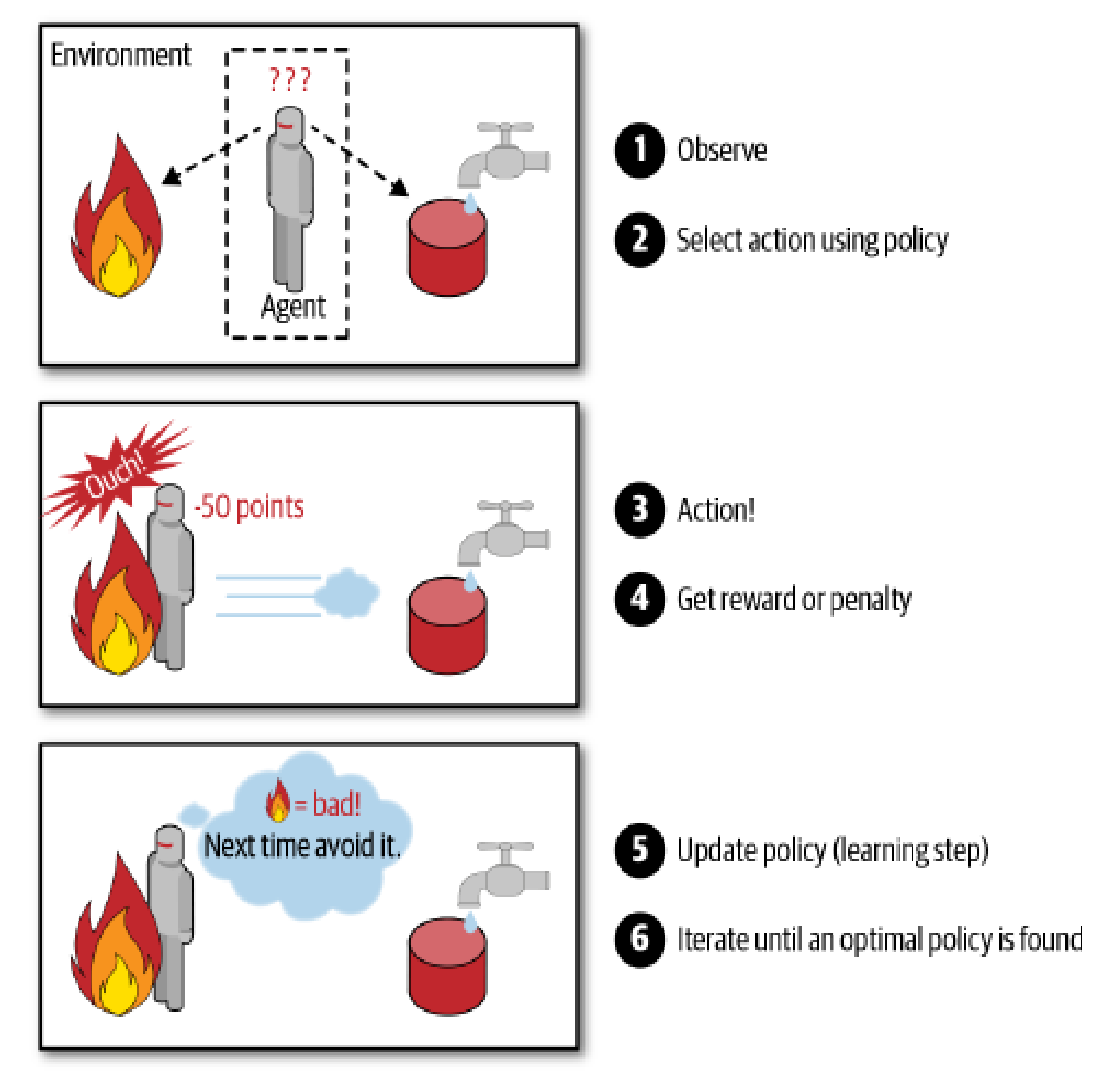


Figure Source: Hands-On Machine Learning with Scikit-Learn, Keras, and TensorFlow

Source: https://towardsdatascience.com/drl-01-a-gentle-introduction-to-deep-reinforcement-learning-405b79866bf4

KEY COMPONENTS OF REINFORCEMENT LEARNING

1) Agent→

- Learning System OR a computer program that takes actions
- Responsibilities of an Agent:

✓ **observes the environment**
✓ **select and perform actions**
✓ **get rewards in return (or penalties)**

- Agent learns by itself to get the most reward over time

2)Environment→

- The surroundings in which the Agent works/operates
- representation of a "problem"
- Based on Agent's decisions→ gives responses or consequences
- Responses→ rewards/penalties

3) Action→

- A move that an agent can make in the environment
- Action space→ represents a set of possible actions that an agent can make in the environment

Source: Hands-On Machine Learning with Scikit-Learn, Keras, and TensorFlow

KEY COMPONENTS OF REINFORCEMENT LEARNING

4) Observations→

- Observations of the environment after the agent takes its actions

5) State→

- Situation which the agent perceives

6) Reward→

- Feedback from the environment that measures the success or failure of the agent's actions.
- Reward for positive action (success)
- Penalty for a negative action (failure)

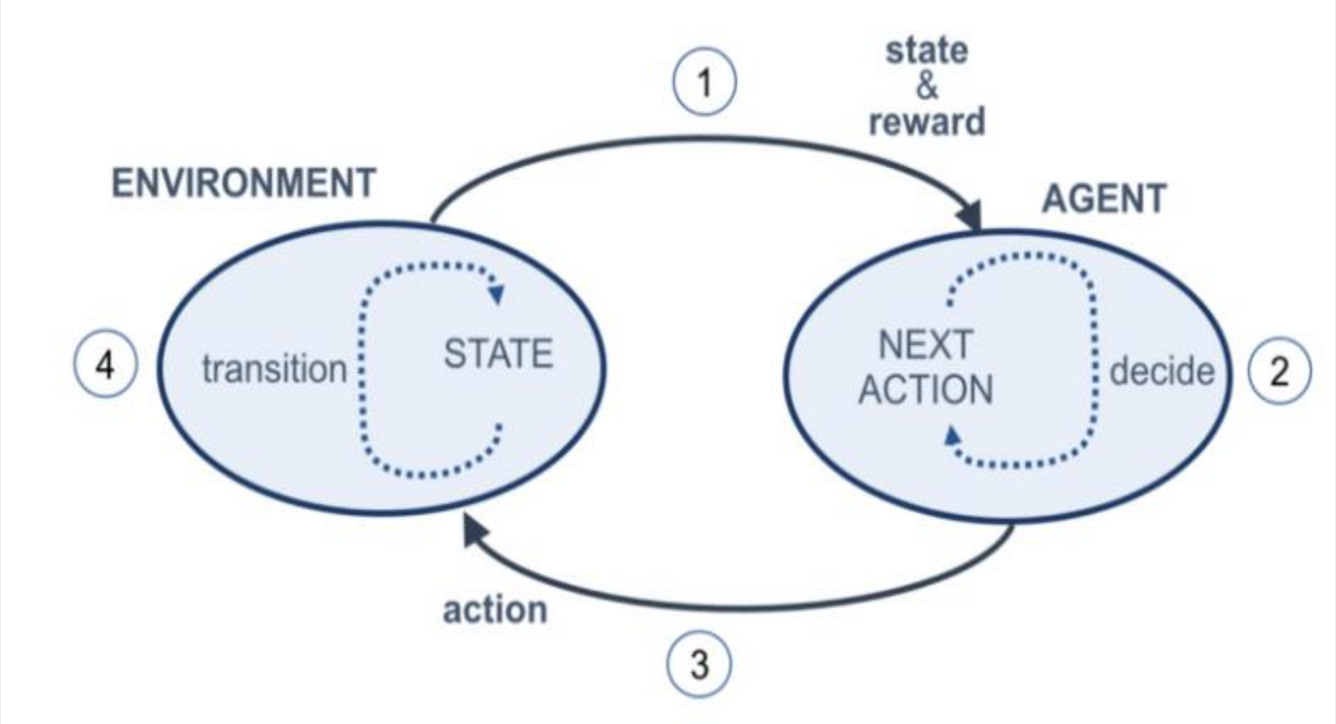
7) Policy→

- best learning strategy
- an action agent takes when in a given situation

Figure Source: introtodeeplearning.com

REINFORCEMENT LEARNING CYCLE

1. Agent observes the environment, takes action, and receives state and reward
2. Agent uses this information to take the next action
3. Agent sends the new action to the environment
4. Environment state changes based on previous state and action taken by the Agent
5. Repeat the cycle



Source: https://towardsdatascience.com/drl-01-a-gentle-introduction-to-deep-reinforcement-learning-405b79866bf4

EXPLORATION VERSUS EXPLOITATION

- trade-off between “exploration” and “exploitation”

- An Agent needs to exploit its past actions to obtain as much positive reward as possible
- At the same time, explore new actions to increase the possibility of more rewards in future

Source: https://towardsdatascience.com/drl-01-a-gentle-introduction-to-deep-reinforcement-learning-405b79866bf4

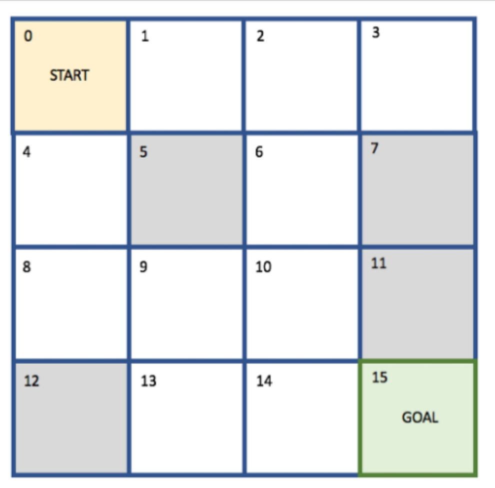
THE FROZEN-LAKE CASE STUDY

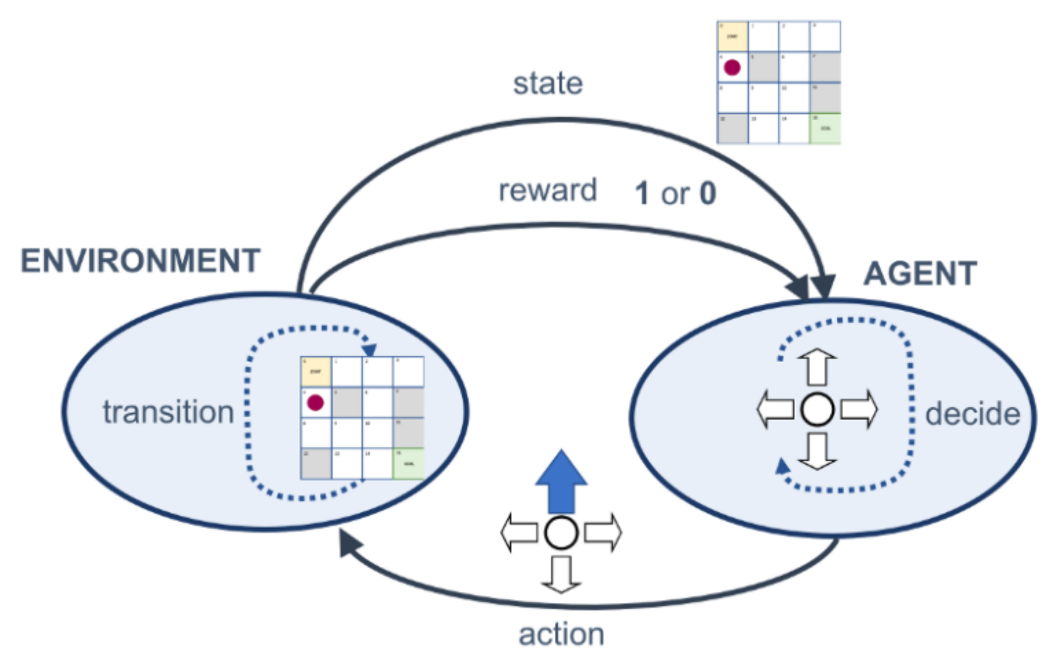
- A simple case study of a very slippery frozen Lake where the agent can skate
- ice skating rink, divided into 16 cells (4x4)
- some of the cells have broken the ice
- The Agent (skater) starts from the top-left position (yellow)
- **Goal:** reach the bottom right box (green) without falling into the 4 holes (grey) on the track

Source: https://towardsdatascience.com/drl-01-a-gentle-introduction-to-deep-reinforcement-learning-405b79866bf4

THE FROZEN-LAKE CASE STUDY

- **Environment:** grid of size 4x4 (has 16 cells)
- **State space:** composed of 16 states (0–15)
- **Start position:** top-left position
- **Goal:** reach the bottom-right position of the grid
- four holes in the fixed cells of the grid
- If Agent falls into the holes→ episode ends, reward= Zero
- If Agent reaches the destination→ episode ends, reward= +1
- **Action space:** four directions movements: up, down, left, and right
- Fence around the lake→ if the Agent tries to move out of the grid world, it will just bounce back to the cell from which it tried to move
- **Behavior of the Environment:** transition function/probabilities
- Lake frozen and the environment is slippery
- Agent’s actions are not always expected!
- there is a 33% chance that it will slip to the right or the left



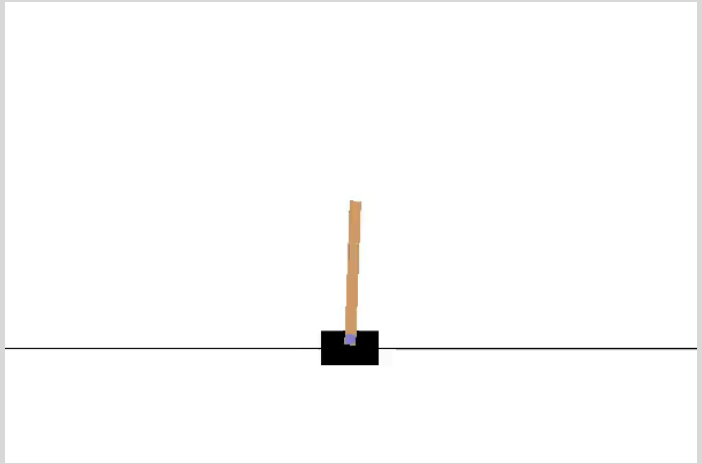


Source: https://towardsdatascience.com/drl-01-a-gentle-introduction-to-deep-reinforcement-learning-405b79866bf4

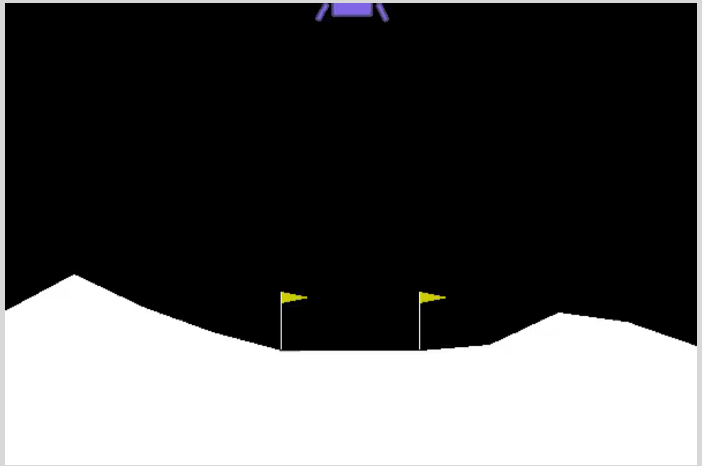
Introduction to OpenAI Gym

- Simulated environment for bootstrap training is needed for DRL
- For example, learn to play chess game, need a chess game simulator
- Make a robot to walk, need a real environment→ has several limitations, need a simulator
- **OpenAI Gym** toolkit for a wide variety of simulated environments for training the Agents using RL, comparing results or developing new RL algorithms
- For example, Atari games, Board games, 2D or 3D physical simulations
- Weblink: https://gym.openai.com/

CartPole-v1



LunarLander-v2



Source: Hands-On Machine Learning with Scikit-Learn, Keras, and TensorFlow

CartPole-v1 Example

- Create a virtual environment and activate it

```
$ cd $ML_PATH                 # Your ML working directory (e.g., $HOME/ml)
$ source my_env/bin/activate # on Linux or MacOS
$ .\my_env\Scripts\activate  # on Windows
```

- Install OpenAI Gym

```
$ python3 -m pip install -U gym
```

- open up a Python shell or a Jupyter notebook and create an environment with make()
- Here, we've created a CartPole environment. This is a 2D simulation in which a cart can be accelerated left or right in order to balance a pole placed on top of it

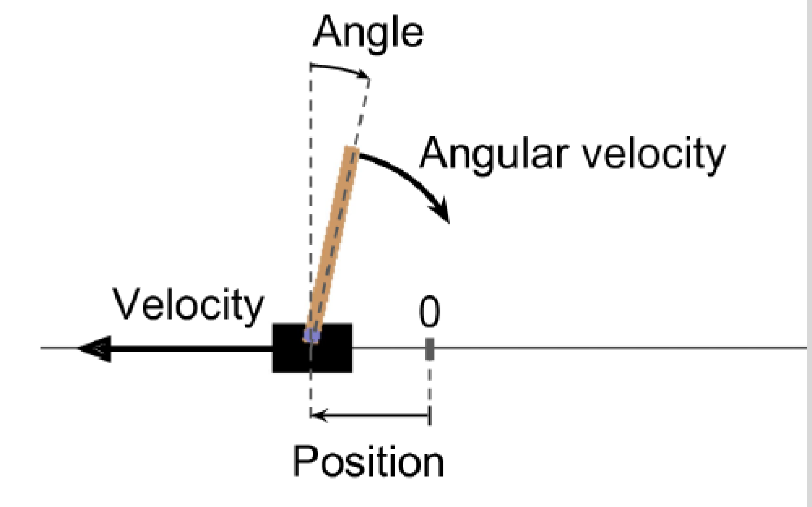
```
>>> import gym
>>> env = gym.make("CartPole-v1")
```


Source: Hands-On Machine Learning with Scikit-Learn, Keras, and TensorFlow

CartPole-v1 Example

- initialize the environment using the reset() method. This returns the first observation.
- For CartPole, each observation is 1D array of 4 floats→ represent cart's position (0.0 = center), its velocity (positive means right), the angle of the pole (0.0 =vertical), and its angular velocity (positive means clockwise)

```
>>> obs = env.reset()
>>> obs
array([-0.01258566, -0.00156614,  0.04207708, -0.00180545])
```



- Display this environment by calling its render() method

```
>>> env.render()
True
```

- If you want render() to return the rendered image as a NumPy array, you can set mode="rgb_array"

```
>>> img = env.render(mode="rgb_array")
>>> img.shape  # height, width, channels (3 = Red, Green, Blue)
(800, 1200, 3)
```


Source: Hands-On Machine Learning with Scikit-Learn, Keras, and TensorFlow

CartPole-v1 Example

- Ask the environment what actions are possible
- Discrete(2) means that the possible actions are integers 0 and 1, which represent accelerating left (0) or right (1).

```
>>> env.action_space
Discrete(2)
```

- Since the pole is leaning toward the right (obs[2] > 0), let's accelerate the cart toward the right
- The step() method executes the given action and returns four values: observation, reward, done, info

```
>>> action = 1  # accelerate right
>>> obs, reward, done, info = env.step(action)
>>> obs
array([-0.01261699,  0.19292789,  0.04204097, -0.28092127])
>>> reward
1.0
>>> done
False
>>> info
{}
```

- **obs [1] > 0,** cart moving right
- **obs [2] >0,** pole tilted towards right
- **obs [3] <0,** angular velocity negative, tilted towards left in next step
- **reward→** always 1, keep episode running
- **done→** True when episode over or pole tilts too much or goes off screen or after 200 steps (when you win)
- **info→** extra information, how many lives the agent has in the game

- Call the close() method to free resources once you are finished

Source: Hands-On Machine Learning with Scikit-Learn, Keras, and TensorFlow

Hardcode simple policy for CartPole-v1 Example

- simple policy that accelerates left when the pole is leaning toward the left and accelerates right when the pole is leaning toward the right
- We will run this policy to see the average rewards it gets over 500 episodes

```python
def basic_policy(obs):
    angle = obs[2]
    return 0 if angle < 0 else 1

totals = []
for episode in range(500):
    episode_rewards = 0
    obs = env.reset()
    for step in range(200):
        action = basic_policy(obs)
        obs, reward, done, info = env.step(action)
        episode_rewards += reward
        if done:
            break
    totals.append(episode_rewards)
```

```python
>>> import numpy as np
>>> np.mean(totals), np.std(totals), np.min(totals), np.max(totals)
(41.718, 8.858356280936096, 24.0, 68.0)
```

- Even with 500 tries, this policy never managed to keep the pole upright for more than 68 consecutive steps
- cart oscillates left and right more and more strongly until the pole tilts too much

Source: Hands-On Machine Learning with Scikit-Learn, Keras, and TensorFlow

Q-Learning

- The 'Q' in Q-learning stands for **Quality**
- **Quality→** represents how useful a given action is in gaining some future reward
- Based on assessing the quality of an action that is taken to move to a state
- Rather than determining the possible value of the state being moved to

Basic steps for Q-Learning:

1. Create a **Q-table or matrix** with shape of **[state, action]** and initial values of zero. The Q-table acts as a reference table for an Agent to select the best action based on Q-value
2. Agent interacts with Environment with an initial state, takes an Action and receives a reward.
3. Agent selects an Action based on max value of that Action for a state (**Exploiting**). Another way is by selecting an Action at random (**Exploring**).
4. Update and store the Q-values after each step or action and end when an episode is done or reaches a terminal point.

Source: https://towardsdatascience.com/simple-reinforcement-learning-q-learning-fcddc4b6fe56

DEFINING THE Q-FUNCTION

- Total reward R_t is the discounted sum of all rewards obtained from time "t"

$$R_t = r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + \cdots$$

γ: discount factor; $0 < \gamma < 1$

- Q-Function captures the expected total future reward an agent in s state "s" can receive by executing a certain action "a"

$$Q(\underbrace{s_t}_{\text{state}}, \underbrace{a_t}_{\text{action}}) = \mathbb{E}[R_t | s_t, a_t]$$

Source: introtodeeplearning.com

VALUE LEARNING VERSUS POLICY LEARNING

- Agent needs a policy to infer the best action to take at a particular state “s”

Value Learning

Find $Q(s, a)$

$$a = \underset{a}{\operatorname{argmax}}\, Q(s, a)$$

Find the Action that maximizes the Q-function or gives the highest Q-value

Policy Learning

Find $\pi(s)$

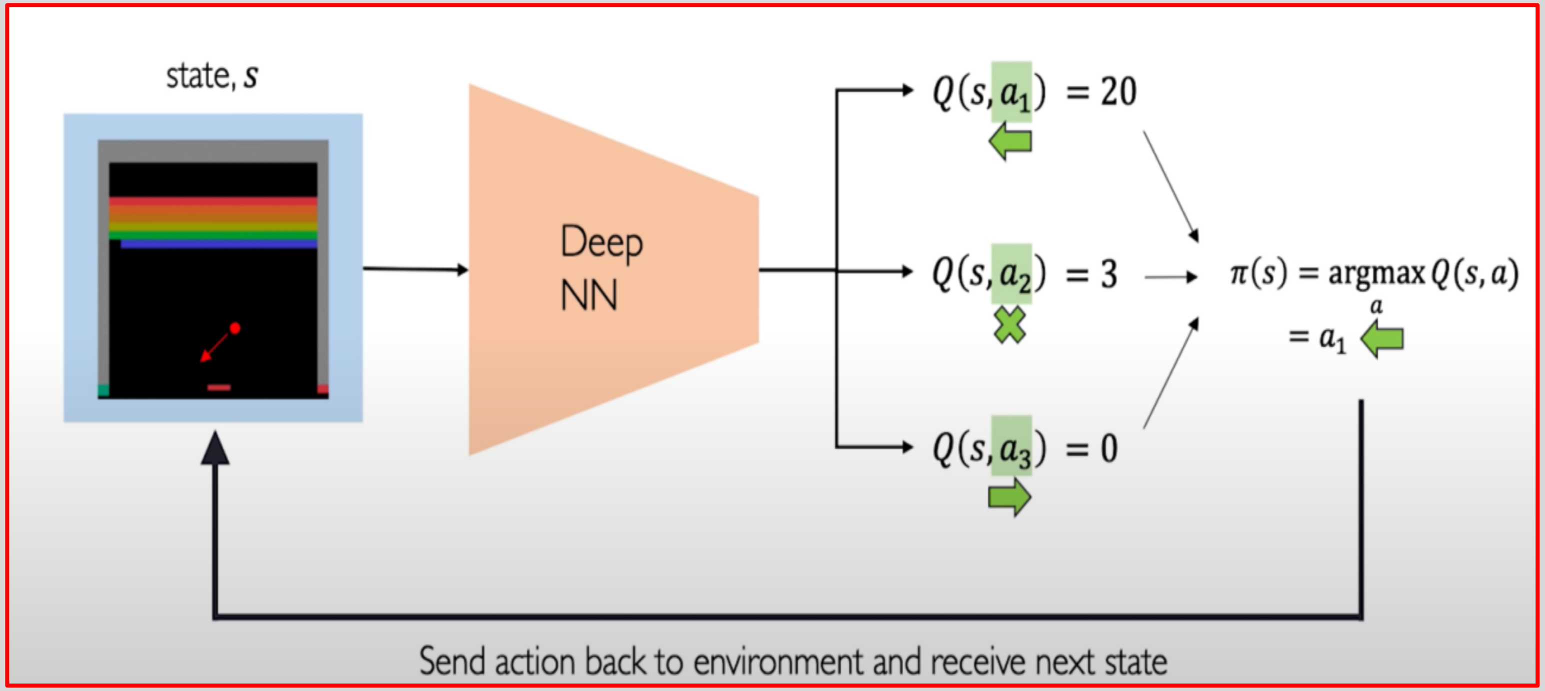
Sample $a \sim \pi(s)$

Directly learns the policy instead of using a Q-function that governs what Action to take

Source: introtodeeplearning.com

DEEP Q-NETWORK (DQN)

- Using Q-learning or Value Learning
- A DNN is used to learn the Q-function and then used to infer the optimal policy
- Example of **Atari Breakout video game**
- Input to DNN is state "**s**" and output Q-value for the three possible Actions "**a**"
- Actions→ move left, right or stay in same place
- To infer the optimal policy, select the action that maximizes the Q-value



Source: introtodeeplearning.com

DISADVANTAGES OF Q-LEARNING (VALUE LEARNING)

Complexity:

- Can model scenarios where the action space is discrete and small→ only few possible actions at each step
- Cannot handle continuous action spaces, for example, Robotic vacuum cleaner can go forward, backward, left and right. Can it go in any other direction not discretized, continuous and infinite space ?

Flexibility:

- Policy computed from the Q function by maximizing the reward
- Cannot learn stochastic policies or random probability distribution

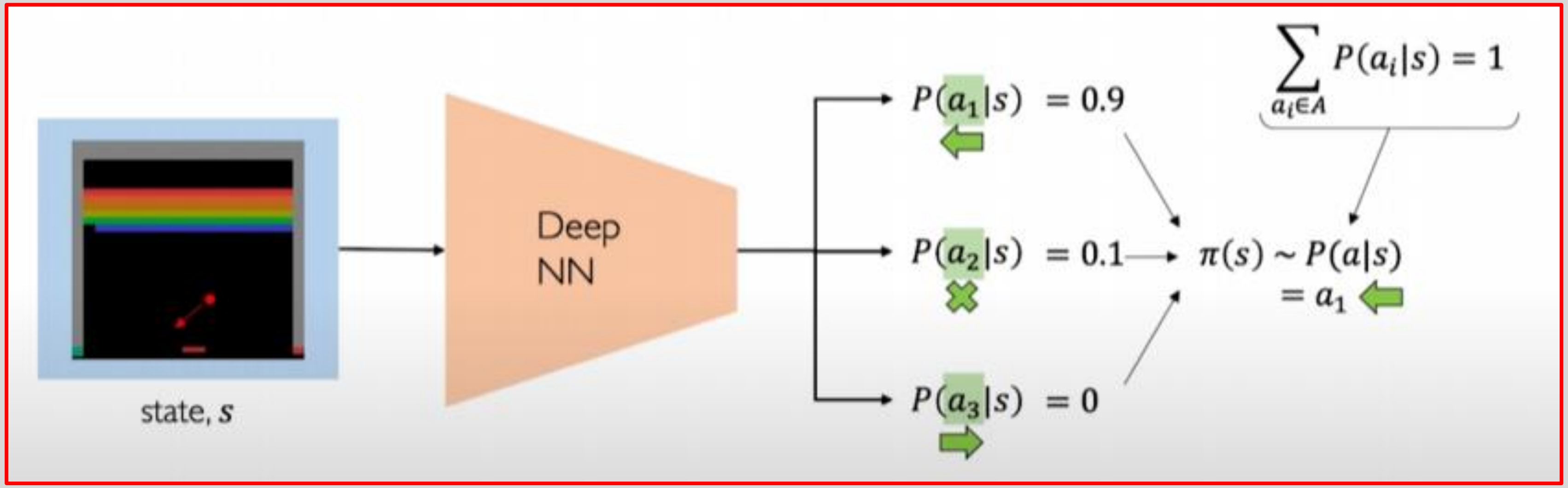
Source: introtodeeplearning.com

POLICY GRADIENT (LEARNING) METHODS

- Address the problems in value or Q-Learning methods
- Directly optimize the policy $\boldsymbol{\pi}(\boldsymbol{s})$
- Sample from the probability distribution to predict an Action to take
- Total probability must sum to 1.

Advantages:

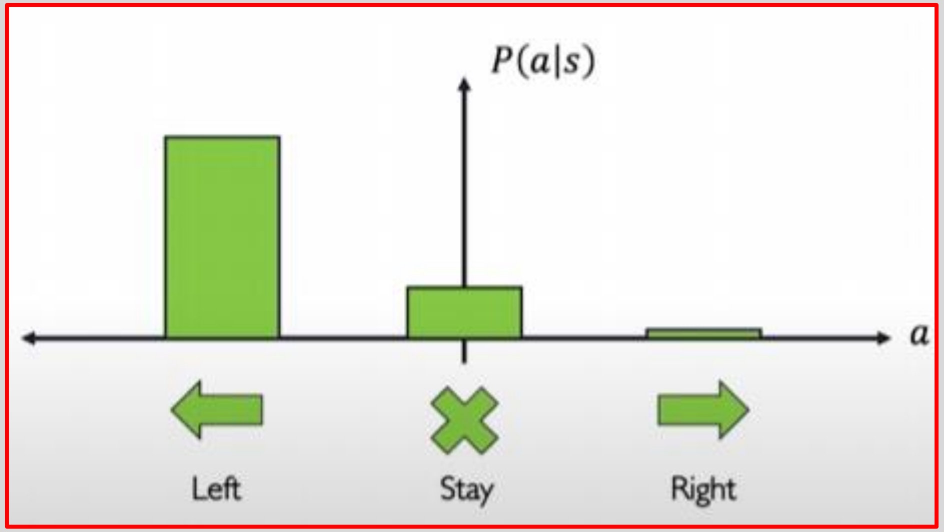
- Directly optimizing a Policy
- Handle continuous Action spaces



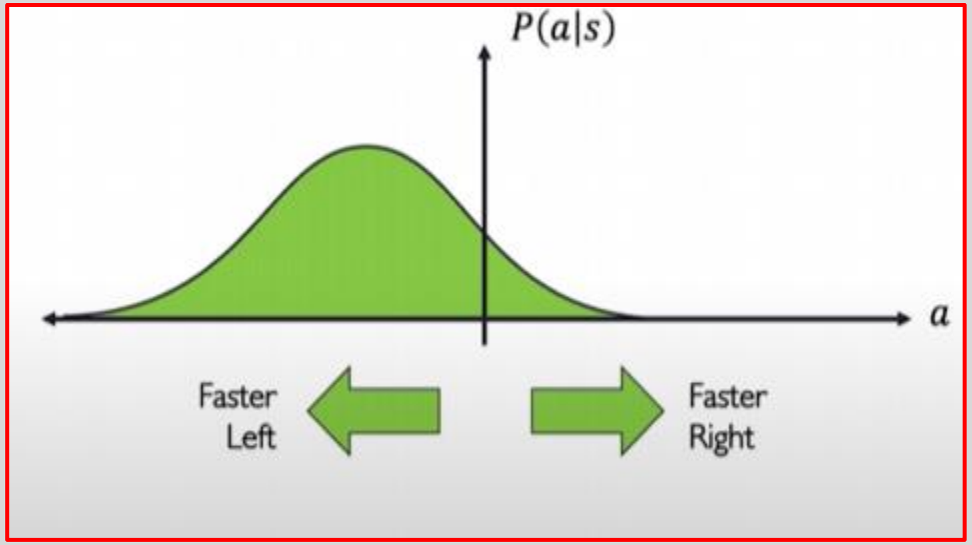
Source: introtodeeplearning.com

DISCRETE ACTION SPACE VERSUS CONTINUOUS

Value Learning (Q-Learning)



Policy Gradients



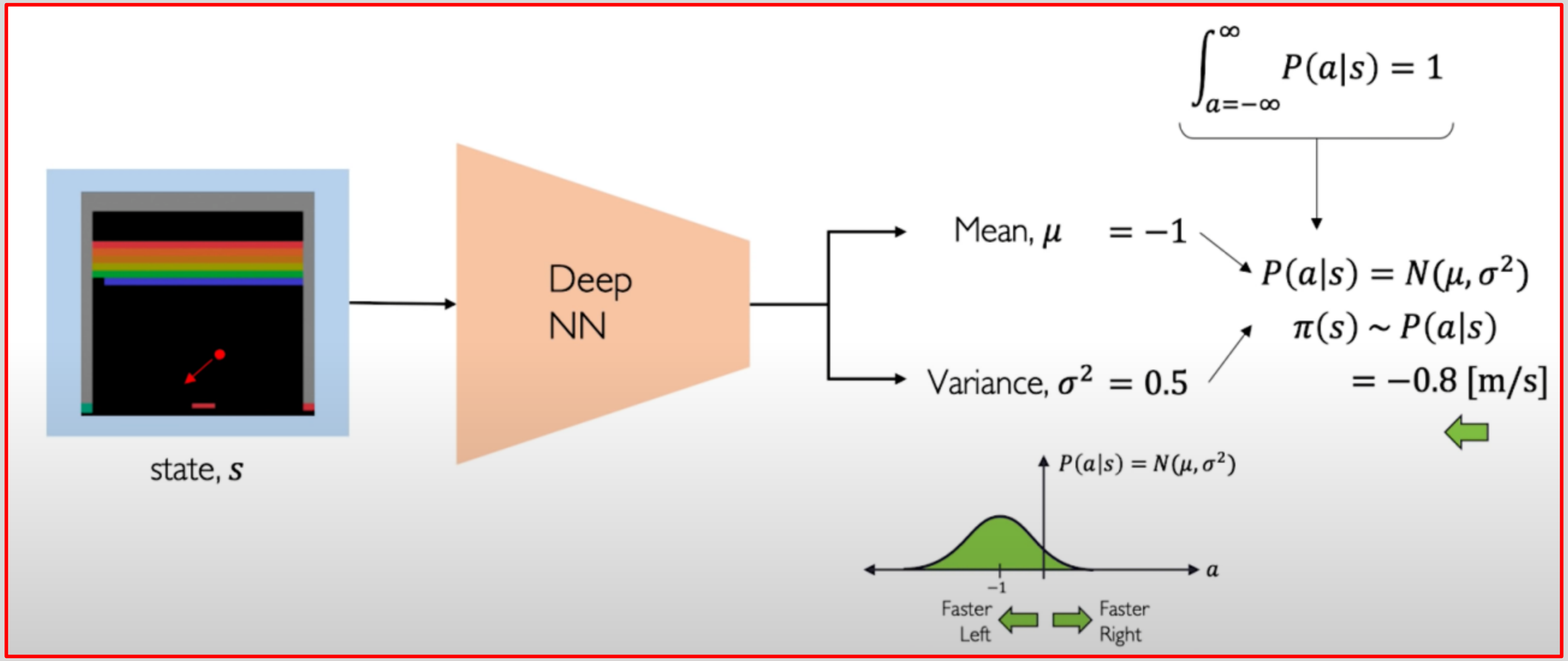
Discrete Action space shows only the directions to move

Continuous Action space shows not only the directions to move but also how fast to move. This can have numerous possibilities. Direction is represented by right (+) and left (-). Represented as Gaussian distribution here. This figure shows that the probability of moving faster towards left is much greater than the right. We can also identify the exact mean value which is the highest point in the middle.

Source: introtodeeplearning.com

POLICY GRADIENTS-MAIN IDEA

- Model the continuous action space with Policy Gradient method
- Instead of predicting the probability of an Action, given a possible State, there will be an infinite number of Actions in this case
- Output distribution is Gaussian with a mean and variance value→ only two outputs



Source: introtodeeplearning.com

IMPLEMENTING A DEEP Q-NETWORK (DQN)

- Build a DQN for the CartPole environment

```
env = gym.make("CartPole-v0")
input_shape = [4] # == env.observation_space.shape
n_outputs = 2 # == env.action_space.n

model = keras.models.Sequential([
    keras.layers.Dense(32, activation="elu", input_shape=input_shape),
    keras.layers.Dense(32, activation="elu"),
    keras.layers.Dense(n_outputs)
])
```

- pick the action with the largest predicted Q-Value using greedy policy

```
def epsilon_greedy_policy(state, epsilon=0):
    if np.random.rand() < epsilon:
        return np.random.randint(2)
    else:
        Q_values = model.predict(state[np.newaxis])
        return np.argmax(Q_values[0])
```

Source: Hands-On Machine Learning with Scikit-Learn, Keras, and TensorFlow

IMPLEMENTING A DEEP Q-NETWORK (DQN)

- Store all the experiences in a replay buffer (or replay memory), and sample a random training batch from it at each training iteration. Use deque list for this

```
from collections import deque

replay_buffer = deque(maxlen=2000)
```

- Each experience will be composed of five elements: a state, the action the agent took, the resulting reward, the next state it reached, and finally a Boolean indicating whether the episode ended at that point (done).
- function to sample a random batch of experiences from the replay buffer

```
def sample_experiences(batch_size):
    indices = np.random.randint(len(replay_buffer), size=batch_size)
    batch = [replay_buffer[index] for index in indices]
    states, actions, rewards, next_states, dones = [
        np.array([experience[field_index] for experience in batch])
        for field_index in range(5)]
    return states, actions, rewards, next_states, dones
```

Source: Hands-On Machine Learning with Scikit-Learn, Keras, and TensorFlow

IMPLEMENTING A DEEP Q-NETWORK (DQN)

- create a function that will play a single step using the ε-greedy policy, then store the resulting experience in the replay buffer

```
def play_one_step(env, state, epsilon):
    action = epsilon_greedy_policy(state, epsilon)
    next_state, reward, done, info = env.step(action)
    replay_buffer.append((state, action, reward, next_state, done))
    return next_state, reward, done, info
```

- define some hyperparameters, and we create the optimizer and the loss function

```
batch_size = 32
discount_factor = 0.95
optimizer = keras.optimizers.Adam(lr=1e-3)
loss_fn = keras.losses.mean_squared_error
```


Source: Hands-On Machine Learning with Scikit-Learn, Keras, and TensorFlow

IMPLEMENTING A DEEP Q-NETWORK (DQN)

- **Create a function training_step() that will sample a batch of experiences from the replay buffer and train the DQN by performing a single Gradient Descent step on this batch**

```
def training_step(batch_size):
    experiences = sample_experiences(batch_size)
    states, actions, rewards, next_states, dones = experiences
    next_Q_values = model.predict(next_states)
    max_next_Q_values = np.max(next_Q_values, axis=1)
    target_Q_values = (rewards +
                       (1 - dones) * discount_factor * max_next_Q_values)
    mask = tf.one_hot(actions, n_outputs)
    with tf.GradientTape() as tape:
        all_Q_values = model(states)
        Q_values = tf.reduce_sum(all_Q_values * mask, axis=1, keepdims=True)
        loss = tf.reduce_mean(loss_fn(target_Q_values, Q_values))
    grads = tape.gradient(loss, model.trainable_variables)
    optimizer.apply_gradients(zip(grads, model.trainable_variables))
```

Sampling a batch of experiences

Predict Q-value for each possible action

Compute Target Q-value

Mask the Q-values not needed

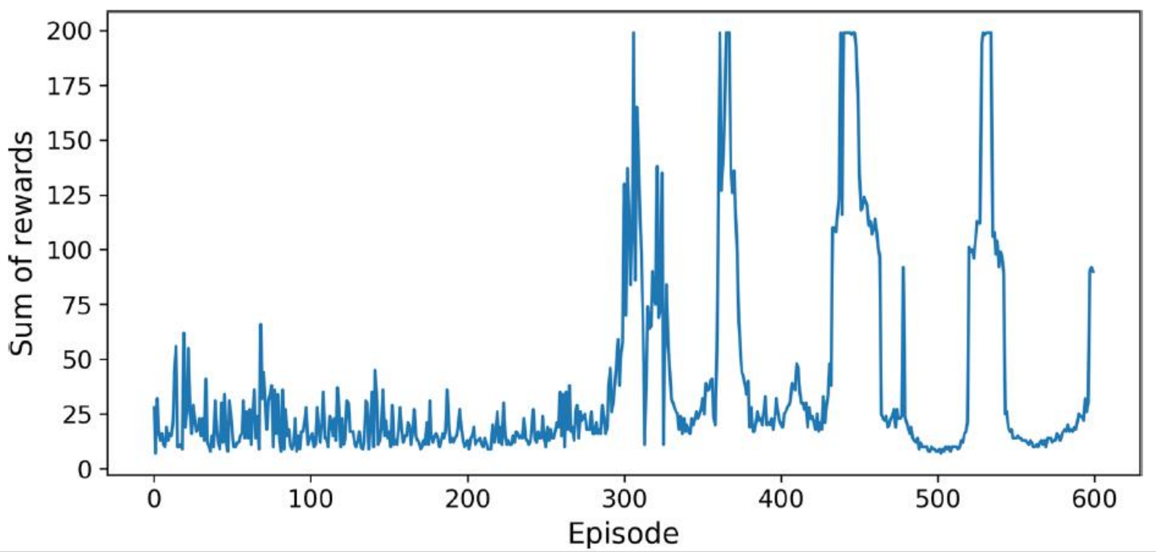
Compute the MSE between target and predicted Q-values and then apply Gradient Descent to minimize the loss

Source: Hands-On Machine Learning with Scikit-Learn, Keras, and TensorFlow

IMPLEMENTING A DEEP Q-NETWORK (DQN)

- Run the model for 600 episodes each for a maximum of 200 steps
- call the play_one_step() function, which will use the ε-greedy policy to pick an action, then execute it and record the experience in the replay buffer
- if we are past the 50th episode, we call the training_step() function to train the model on one batch sampled from the replay buffer

```python
for episode in range(600):
    obs = env.reset()
    for step in range(200):
        epsilon = max(1 - episode / 500, 0.01)
        obs, reward, done, info = play_one_step(env, obs, epsilon)
        if done:
            break
    if episode > 50:
        training_step(batch_size)
```



Learning curve of the Deep Q-Learning algorithm shows catastrophic forgetting. As the agent explores the environment, it updates its policy, but what it learns in one part of the environment may break what it learned earlier in other parts of the environment

Source: Hands-On Machine Learning with Scikit-Learn, Keras, and TensorFlow

ADVANTAGES AND DISADVANTAGES OF DRL

Advantages:

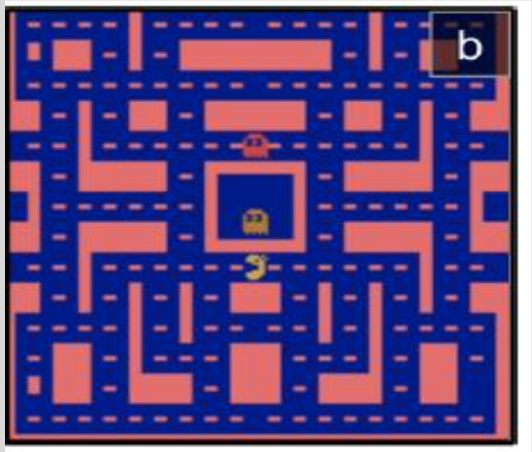
- Used to solve complex problems that cannot be solved by conventional techniques
- achieve long-term results which are very difficult to achieve using other algorithms
- learning model is very similar to the learning of human beings

Disadvantages:

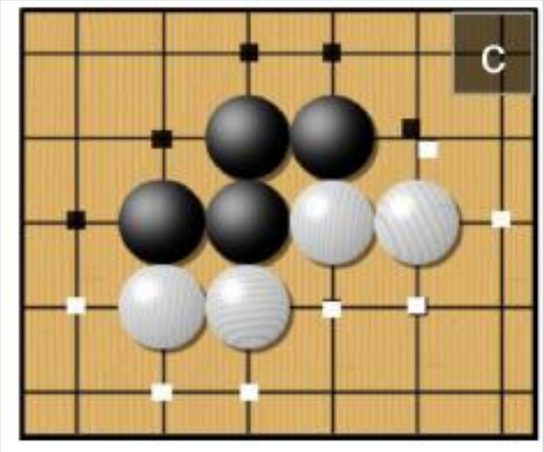
- Notoriously difficult→ training instabilities
- huge sensitivity to the choice of hyperparameter values and random seeds
- Not preferable for solving simple problems
- Needs a lot of data and a lot of computation

SOME APPLICATIONS OF DRL

(a) Robotics



(b) Ms. Pac-Man



(c) Go player

(d) Thermostat

(e) Automatic Trader

THANKS!

Do you have any questions?