

Lecture Notes for **Neural Networks and Machine Learning**



Introduction to
Reinforcement Learning



Logistics and Agenda

- Logistics
 - Lab Four: Cleaning up GANs
- Agenda
 - Pytorch
 - Basics of Reinforcement Learning
 - Markov Processes
 - Reinforcement Learning Categorization
 - OpenAI Gym



Basics of Pytorch

When you're the only one of your friends
who uses PyTorch instead of TensorFlow



why I am more successful
than you.



Chip Huyen @chipro · 17h

Sometimes I feel like R. Nobody's favorite
but functional and pretty good with data.

9




23

237



Wait, why are we switching to Pytorch?

- Well, its good to know more than just Tensorflow
- Pytorch has some distinct advantages:
 - No need to setup a static computation graph—graph can be dynamic (like eager execution)
 - ♦ Lazy computations still happen on dynamic graph
 - Integration with numpy code on the fly is much easier and faster (compared to TF)
 - ♦ Can tradeoff computations with numpy easily, though not necessarily with autograd
- Also, all the book examples are in Pytorch... so this is nice for following along with the examples

	Keras 	TensorFlow 	PyTorch 
Level of API	high-level API ¹	Both high & low level APIs	Lower-level API ²
Speed	Slow	High	High
Architecture	Simple, more readable and concise	Not very easy to use	Complex ³
Debugging	No need to debug	Difficult to debugging	Good debugging capabilities
Dataset Compatibility	Slow & Small	Fast speed & large	Fast speed & large datasets
Popularity Rank	1	2	3
Uniqueness	Multiple back-end support	Object Detection Functionality	Flexibility & Short Training Duration
Created By	Not a library on its own	Created by Google	Created by Facebook ⁴
Ease of use	User-friendly	Incomprehensive API	Integrated with Python language
Computational graphs used	Static graphs	Static graphs	Dynamic computation graphs ⁵



Pytorch General Flow Training Flow

- Inherit from `torch.nn.Module`
- Define `__init__` and `forward`
- Run epochs in a loop with explicit calls to:
 - Loss creation (for batch)
 - Backward calculation of gradient for batch
 - Step of optimizer for batch
 - Gives a great deal of flexibility to design and optimization process
- Lots of different pythonic ways to carry this out
 - Your book likes to setup steps of model through iterators (**yield** the batch, loss, etc.)



A Simple Definition (much like Keras!)

```
1 import torch
2 import torch.nn as nn
3
4 class OurModule(nn.Module):
5     def __init__(self, num_inputs, num_classes, dropout_prob=0.3):
6         super(OurModule, self).__init__()
7         self.pipe = nn.Sequential(
8             nn.Linear(num_inputs, 5),
9             nn.ReLU(),
10            nn.Linear(5, 20),
11            nn.ReLU(),
12            nn.Linear(20, num_classes),
13            nn.Dropout(p=dropout_prob),
14            nn.Softmax(dim=1)
15        )
16
17    def forward(self, x):
18        return self.pipe(x)
19
20 if __name__ == "__main__":
21     net = OurModule(num_inputs=2, num_classes=3)
22     print(net)
23     v = torch.FloatTensor([[2, 3]])
24     out = net(v)
25     print(out)
26     print("Cuda's availability is %s" % torch.cuda.is_available())
27     if torch.cuda.is_available():
28         print("Data from cuda: %s" % out.to('cuda'))
```

Sequential Definitions

Common Functions



The MNIST Example (not so like Keras)

Functional Definitions

```
import torch
import torch.nn as nn
import torch.nn.functional as F
import torch.optim as optim

class Net(nn.Module):
    def __init__(self):
        super(Net, self).__init__()
        self.conv1 = nn.Conv2d(1, 20, 5, 1)
        self.conv2 = nn.Conv2d(20, 50, 5, 1)
        self.fc1 = nn.Linear(4*4*50, 500)
        self.fc2 = nn.Linear(500, 10)

    def forward(self, x):
        x = F.relu(self.conv1(x))
        x = F.max_pool2d(x, 2, 2)
        x = F.relu(self.conv2(x))
        x = F.max_pool2d(x, 2, 2)
        x = x.view(-1, 4*4*50)
        x = F.relu(self.fc1(x))
        x = self.fc2(x)
        return F.log_softmax(x, dim=1)
```

Definitions

```
torch.nn.Conv2d(in_channels, out_channels, kernel_size,
                 stride=1, padding=0, dilation=1, groups=1, bias=True)
```

```
view(*shape) → Tensor    reshape without copy
```

Training One Epoch

```
def train(args, model, device, train_loader, optimizer, epoch):
    model.train()
    for batch_idx, (data, target) in enumerate(train_loader):
        data, target = data.to(device), target.to(device)
        optimizer.zero_grad()
        output = model(data)
        loss = F.nll_loss(output, target)
        loss.backward()
        optimizer.step()
```

Training Multiple Epochs

```
model = Net().to("cpu")
optimizer = optim.SGD(model.parameters())

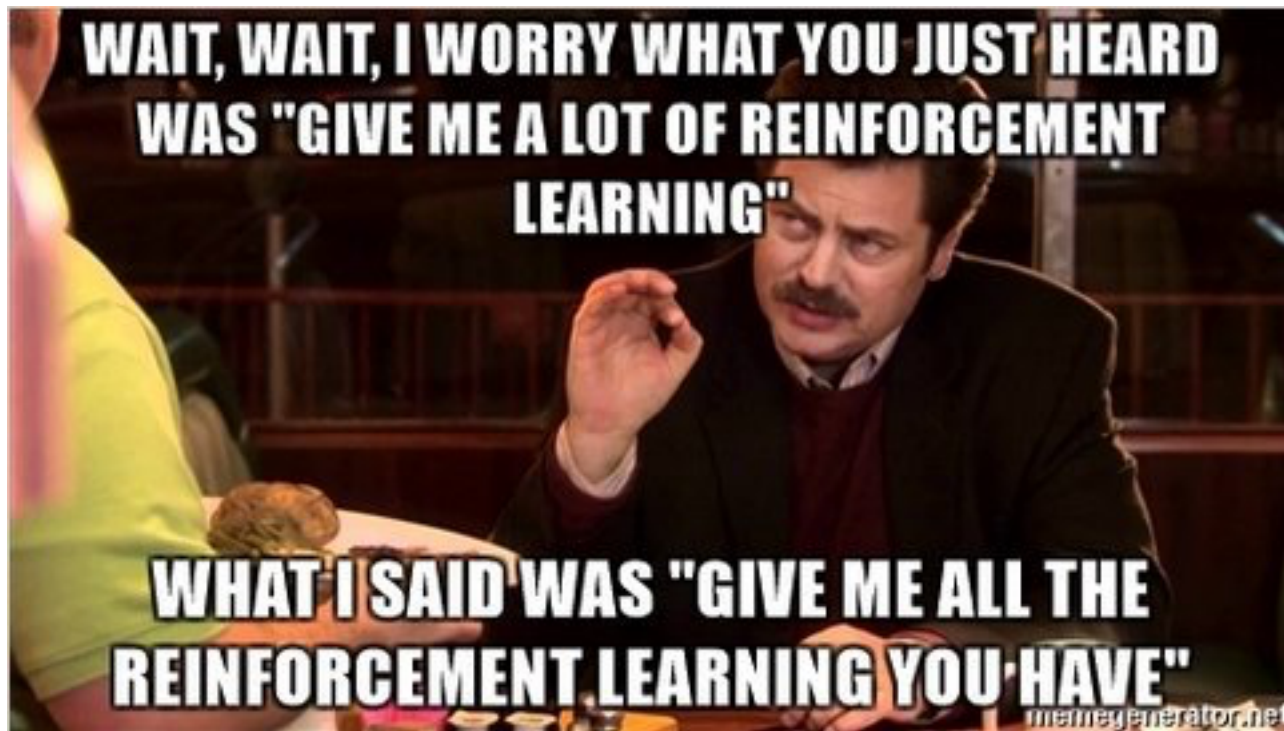
for epoch in range(1, args.epochs + 1):
    train(args, model, "cpu", train_loader, optimizer, epoch)
```

Utils

```
from torchvision import datasets, transforms
train_loader = torch.utils.data.DataLoader(
    datasets.MNIST('../data', train=True, download=True,
                   transform=transforms.Compose([
                       transforms.ToTensor(),      mean, std
                       transforms.Normalize((0.1307,), (0.3081,))
                   ])),
    batch_size=args.batch_size, shuffle=True, **kwargs)
```



Reinforcement Learning Basics



History of RL from Two Paths

- **Optimal Control**

- Model processes via Markov property
- Optimal paths through states calculated through dynamic programming

- **Animal Behavioral Learning (psychology)**

- Animals learn by trial and error
- Formalized by Thorndike, 1911. Strengthen through pleasure and weaken through pain
- Motivation for many pioneering Researchers: Claude Shannon, J. Deutsch, Marvin Minsky, F. Rosenblatt, Widrow, Hoff



Riddles in the Dark

- Hybrid of **Supervised** and **Unsupervised** Learning
- **Reinforcement** Learning
 - Possibly specific labels given, but not without supervision
 - Uses many techniques from supervised learning, but applied towards a different objective
 - Rewards (positive and negative) are possible to assess behavior in an environment
 - Not specific to Machine Learning community



RL Landscape

- **Agent**

- Interacts with the environment. Your model guides the Agent's decisions

- **Environment**

- Anything that is not the agent

- **Observations**

- What the agent knows about the environment

- **Actions**

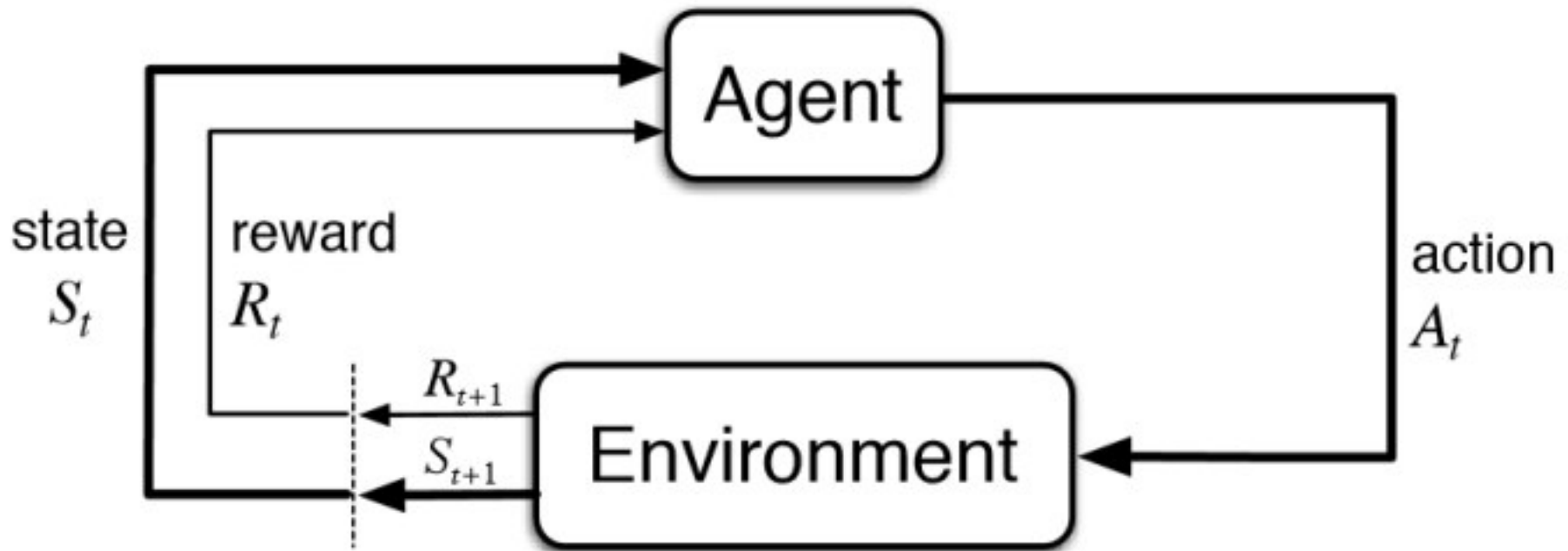
- What an agent can perform with the given environment

- **Rewards**

- Local measure of success
- Can compound local rewards over time



Generic Reinforcement Learning



RL Parameters in Psychology

- One model for some human behavior,
such as **learning a new topic**
- Agent: **You**
- Environment (be specific):
- Observations:
- Actions:
- Reward:

Conclusion: The **complexity** of the process is entirely from the **design** and **assumptions** made in creating the **environment, obs, actions**



Markov Building Blocks



SPACEBAR INSERTS ITS BEST GUESS.
SO IF I TYPE "THE EMPI" AND
HIT SPACE THREE TIMES, IT TYPES
"THE EMPIRE STRIKES BACK."



I GUESS IT FILLS IN YOUR MOST
LIKELY FIRST WORD, THEN THE
WORD THAT USUALLY FOLLOWS IT...

SO IT BUILDS UP YOUR
"TYPICAL" SENTENCE.
COOL! LET'S SEE YOURS!



Markov Processes

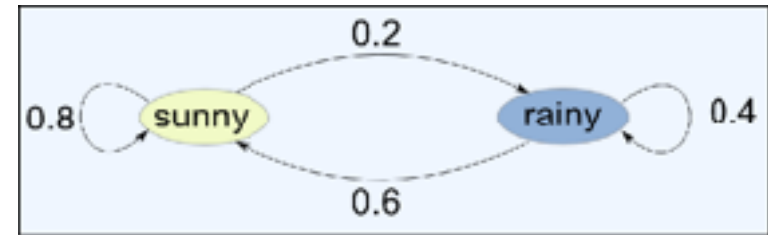
- **Definition:** Any process that can be explained (or simplified) through a sequential set of states that depend only on the previous state
- **Practical Meaning:** For N states, there will be the probability of transition to any other state, encoded through an $N \times N$ transition matrix
- State sequences are not deterministic, they are sampled from the distributions
- Despite **simplicity**, they can model a number of real processes with **enough** precision

Previous State	Next State				
	0.1	0.2	0.1	0.6	0.0
	0.9	0.0	0.1	0.0	0.0
	0.0	0.4	0.0	0.4	0.2
	0.0	0.4	0.2	0.0	0.4
	0.0	0.0	0.6	0.0	0.4



MP Example from Lapan

	Sunny'	Rainy'
Sunny	0.8	0.2
Rainy	0.6	0.4



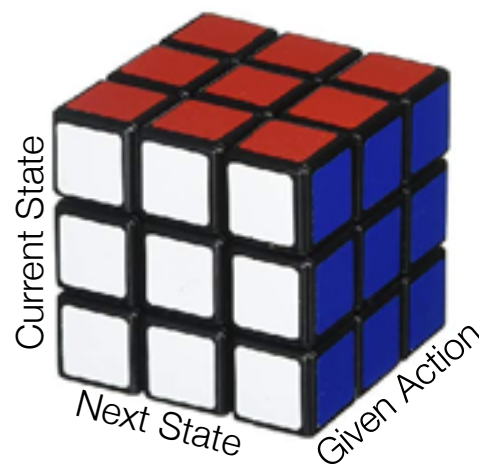
Sun+Summer	
Rainy+Summer	
Sun+Fall	
Rainy+Fall	
Sun+Else	
Rainy+Else	

Adding One Complexity Variable Can Have Large Effect on State Space



Markov Decision Processes

- **New:** Now any state to state transition can be altered by an action
- **Definition:** An MDP consists of:
 - Env. States, s_t
 - Actions for each state, $a(s_t)$
 - Reward function for each state, $r(s_t)$
 - A transition model, $P(s_{t+1}, s_t \mid a)$
a matrix of probabilities
 - ♦ Not **guaranteed** next state by given action



Markov Reward Process

- **Total reward** is given by sum of all **future** rewards

$$G_t = R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} \dots = \sum_k \gamma^k R_{t+k+1}$$

- Gamma defines far- and short-sightedness
 - Common values are **0** (short), **0.9**, **0.99**, and **1** (far)
- This reward calculation can be used to estimate the “Value” of each state based upon the average total reward a state should give, $V(s) = \mathbf{E}[G \mid s_t=s]$
- Typically, this value must be estimated from the model over fixed sequences, otherwise all values can go to infinity (will return to this)

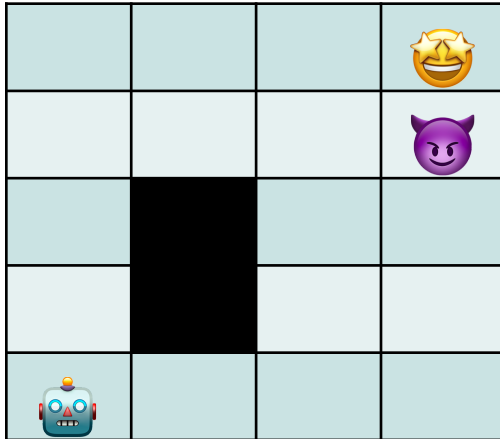


MDPs and MRPs

- How do we select a good action given a current state?
- If gamma is not 0, this can get really complicated as we need to look at all possible future actions
- Instead of defining what is optimal, let's instead setup a comparison of different rules (**policies**)
- A **policy** is defined as $\pi(a, s) = P(a_t = a \mid s_t = s)$
 - Given the current state, we have a certain probability of selecting each action
 - Action selection is **probabilistic**, but easy to define **deterministic** actions (*set one action to 1.0, all others to 0.0*)
- Try different policies, select one with best average reward

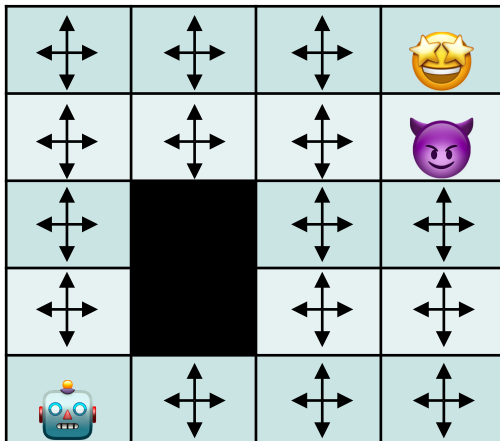


An Illustrative Example: Grid World

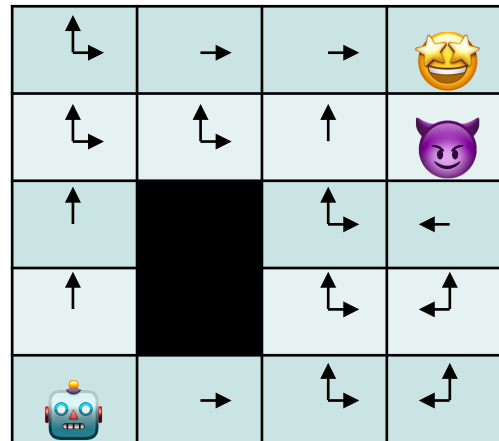


- **State:** Every square in grid
- **Action:** Move to make (l,r,u,d), with probability
- **Reward:** Goal, Death
- **Policy:** Given state, where should we move?
- **Optimal Policy:**

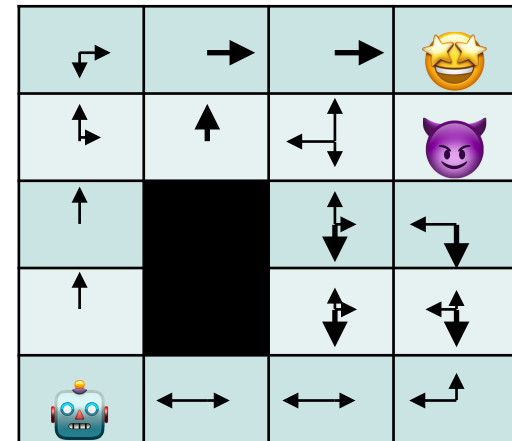
$$\pi^* = \arg \max_{\pi} \mathbf{E} \left[\sum_k \gamma^k R_{t+k+1} \mid \pi \right]$$



Random Policy



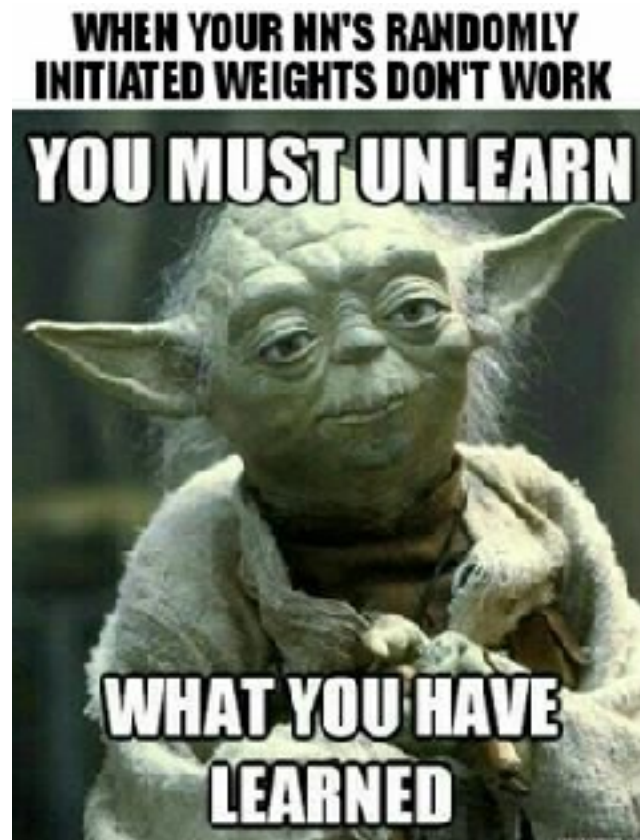
Another Policy



Another Policy



RL Categorization



Various Taxonomies

- Model-based versus Model-free
- Policy-based versus Value-based
- On-Policy, Off-Policy

- On-policy
 - We must interact with environment to learn a policy
- Off-policy
 - Can learn also from historical data or humans



Model-based versus Model-free

- Model Based
 - Predict the next observation and reward based on an understanding of the rules in environment
 - Often look a number of moves ahead (like in chess or similar game)
 - Hard to construct in complex environments
- Model Free
 - Don't care what the problem is
 - Directly try to connect observations to actions (or values from which an action can be inferred)
- Mixed: Sure, like Alpha-Go

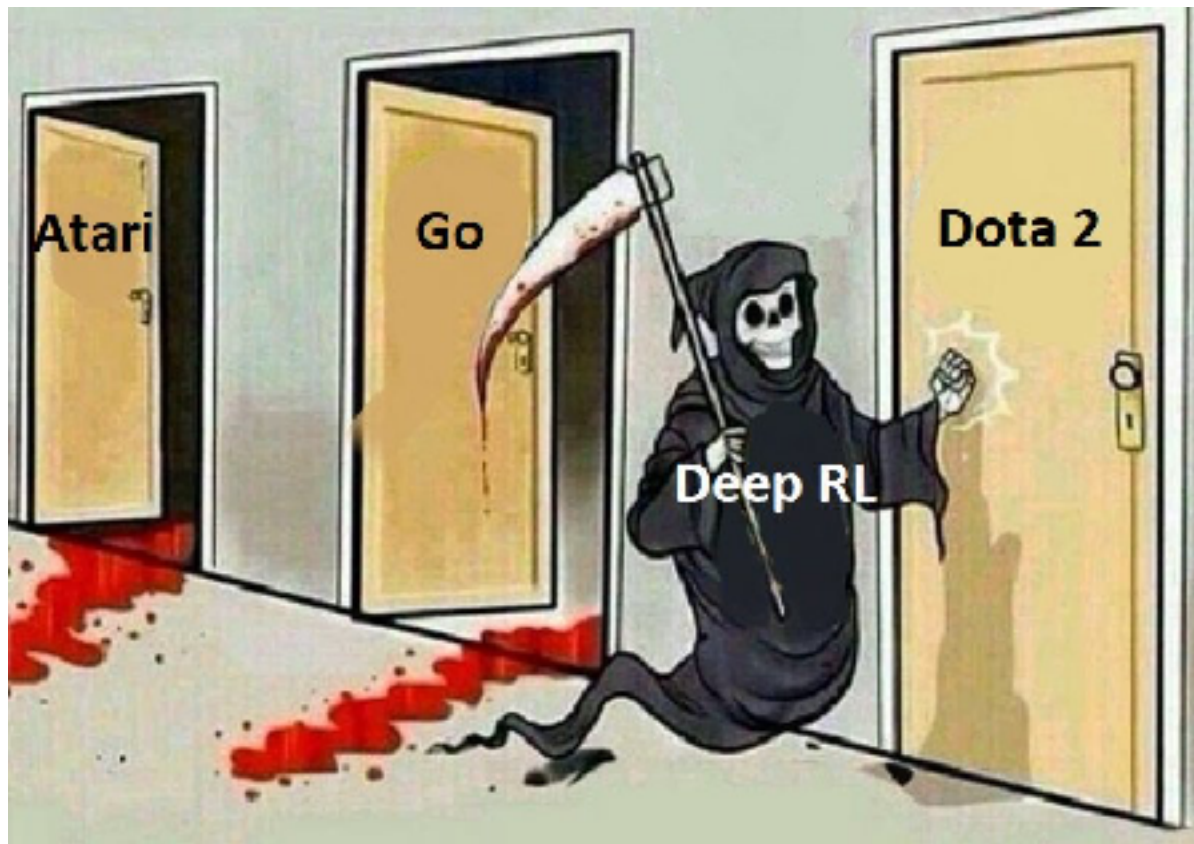


Policy Based versus Value Based

- Policy Based
 - Directly approximate the policy of the agent
 - Policy is typically a probability distribution of actions that we sample from for next action
- Value Based
 - Calculate an intermediate value function for all possible actions
 - Choose the best action based on value function



OpenAI Gym



Object Oriented RL

- Basics:
 - Define object instance for Agent() and the Env()
 - Define what observations will return
 - Run env.step(action)
 - Get new observations and reward from env
- **action_space** and **observation_space**
 - Possible actions to execute, Observations to get
 - Discrete or continuous?
 - Can actions be given simultaneously?



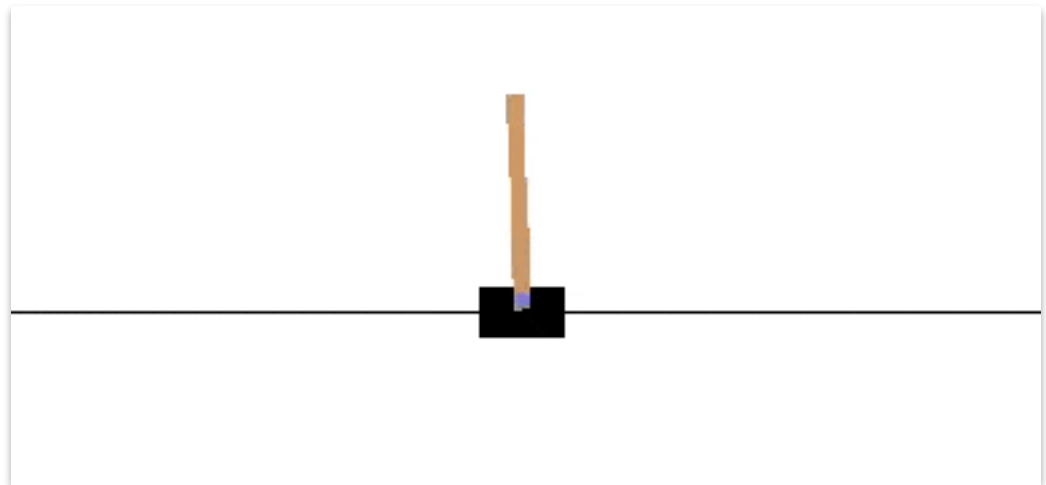
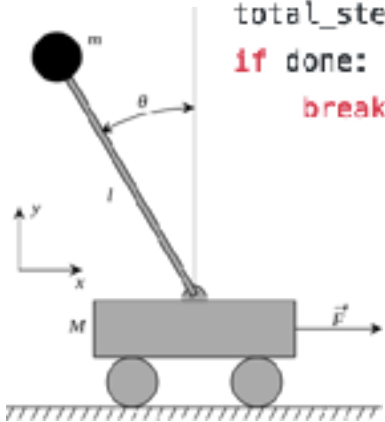
Basics of Cartpole

```
import gym

if __name__ == "__main__":
    env = gym.make("CartPole-v0")

    total_reward = 0.0
    total_steps = 0
    obs = env.reset()

    while True:
        action = env.action_space.sample()
        obs, reward, done, _ = env.step(action)
        total_reward += reward
        total_steps += 1
        if done:
            break
```



Action Space: One input, $[0, 1]$ pull left or pull right

Obs Space: Dynamic state variables (continuous and four dimensional)

End: When more than 15 degrees off or too far from center

Reward: +1 for each time step



Wrapping the Environment

- When you want some extra action, observation, reward processing
- Expose function with `ActionWrapper`, `RewardWrapper`, `ObservationWrapper`

```
class RandomActionWrapper(gym.ActionWrapper):
    def __init__(self, env, epsilon=0.1):
        super(RandomActionWrapper, self).__init__(env)
        self.epsilon = epsilon

    def action(self, action):
        if random.random() < self.epsilon:
            print("Random!")
            return self.env.action_space.sample()
        return action
```

```
if __name__ == "__main__":
    env = RandomActionWrapper(gym.make("CartPole-v0"))

    obs = env.reset()
    total_reward = 0.0

    while True:
        obs, reward, done, _ = env.step(0)
        total_reward += reward
        if done:
            break
```

Might return different action than user supplied
with small probability

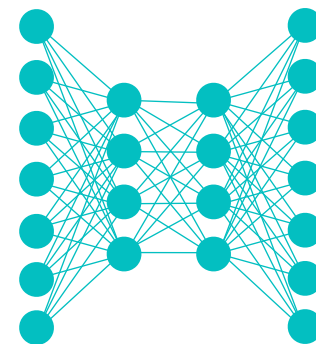


Lecture Notes for **Neural Networks and Machine Learning**

Intro to Reinforcement Learning



Next Time:
CrossEntropy and Q-Learning
Reading: Lamar CH4-CH6

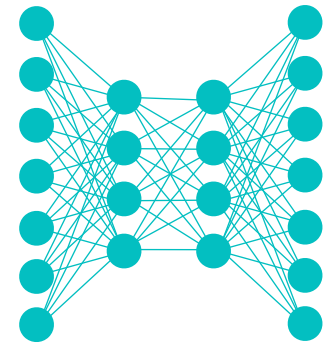




Lecture Notes for **Neural Networks and Machine Learning**



Cross Entropy and
Value Iteration

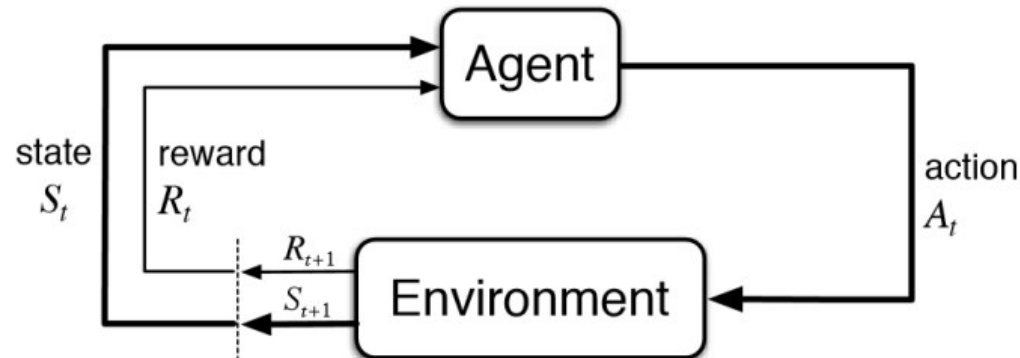


Logistics and Agenda

- Logistics
 - Paper for Class
- Agenda
 - The Cross Entropy Method
 - Value Iteration
 - Q-Learning



Last Time



- **State:** Every square in grid
- **Action:** Move to make (l,r,u,d), with probability
- **Reward:** Goal, Death
- **Policy:** Given state, where should we move?
- **Optimal Policy:**

$$\pi^* = \arg \max_{\pi} \mathbb{E} \left[\sum_k \gamma^k R_{t+k+1} \mid \pi \right]$$



Random Policy



Another Policy



Another Policy

```

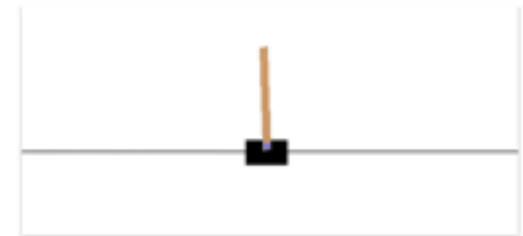
import gym

if __name__ == "__main__":
    env = gym.make("CartPole-v0")

    total_reward = 0.0
    total_steps = 0
    obs = env.reset()

    while True:
        action = env.action_space.sample()
        obs, reward, done, _ = env.step(action)
        total_reward += reward
        total_steps += 1
        if done:
            break

```



Action Space: One input, [0, 1] pull left or pull right

Obs Space: Dynamic state variables (continuous and four dimensional)

End: When more than 15 degrees off or too far from center

Reward: +1 for each time step



Cross Entropy Method



Optimize Best Random Models

- Create a random neural network
- Let it interact with the environment (randomly)
 - For some set of episodes (e.g., 20)
 - Use network output to sample from possible actions
 - Run episode to completion
- Calculate reward for each episode
- Keep best episodes (some percentile, e.g., best five)
- For the given best episodes, develop loss function incentivizing the actions taken based upon the input observations



Cross Entropy Method

- Model based or Model Free?
 - Model Free (no assumptions of problem)
- Value or Policy Based?
 - Policy Based (randomly sample actions based on policy)
- On-policy or Off-Policy?
 - On-Policy (need to interact with environment to get better)
- Has some similarity to **Simulated Annealing** Optimization



How to Make this more Mathy?

- If we have all possible policies $p(x)$ and a reward function $H(x)$, then maximize

$$\mathbf{E}_{x \leftarrow p(x)}[H(x)] = \mathbf{E}_{x \leftarrow q(x)}\left[\frac{p(x)}{q(x)}H(x)\right]$$

- We can approximate the distribution by: $\frac{1}{N} \sum_i \frac{p(x_i)}{q(x_i)} H(x_i)$
- Proven that this is optimized when $\text{KL}(q(x) \parallel p(x)H(x))$ is minimized. But its intractable, so we drop terms ... and end up just optimizing (neg) cross entropy of samples

$$\pi_{k+1} = \arg \max_{\pi_k} \mathbf{E}_{z \leftarrow \pi_k} [\mathbf{1}_{R(z) > \psi} \log \pi_k]$$

Performance
Measure



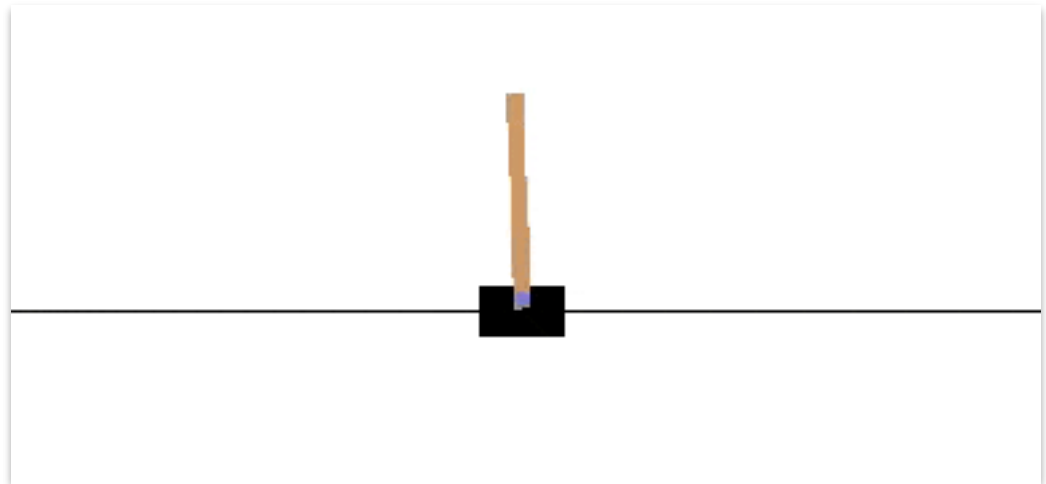
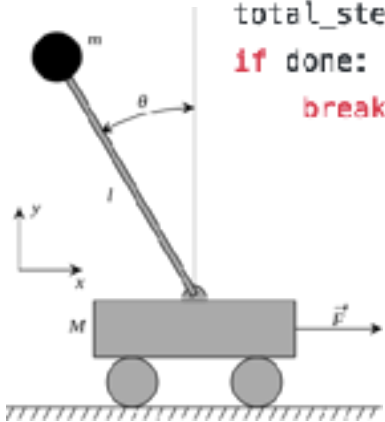
Review: Basics of Cartpole

```
import gym

if __name__ == "__main__":
    env = gym.make("CartPole-v0")

    total_reward = 0.0
    total_steps = 0
    obs = env.reset()

    while True:
        action = env.action_space.sample()
        obs, reward, done, _ = env.step(action)
        total_reward += reward
        total_steps += 1
        if done:
            break
```



Action Space: One input, $[0, 1]$ pull left or pull right

Obs Space: Dynamic state variables (continuous and four dimensional)

End: When more than 15 degrees off or too far from center

Reward: +1 for each time step





Cross Entropy Reinforcement Learning

M. Lapan Implementation for CartPole
and Frozen Lake

Follow Along:

`08a_Basics_Of_Reinforcement_Learning.ipynb`

39



Value Iteration



State Value Review

- Given:
$$G_t = R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} \dots = \sum_k \gamma^k R_{t+k+1}$$
- $V(s) = \mathbf{E}[G \mid s_t=s]$, expected Value of a given state over all future iterations
- **Important:** we can only calculate this exactly if we know:
 - all the rewards for all the states
 - the probabilities of transitioning to a given state from selecting an action
 - likelihood of successful action
 - Most of the time **we know none of this** when we approach the problem, because it assumes a model of the system



The Bellman Equation

- For the case when each action is successful and state is discrete:

$$V_0 = \max_{a \in 1 \dots A} (r_a + \gamma V_a)$$

current value is immediate reward plus value of next state with highest value

- Which feels like cheating because we assume we know V_a ... just go with it for now
- General extension for when actions are probabilistic:

$$V_0 = \max_{a \in A} \mathbf{E}[r_{s,a} + \gamma V_s] = \max_{a \in A} \sum_{s \in S} p_{a,0 \rightarrow s} \cdot (r_{s,a} + \gamma V_s)$$

-probabilities of getting to next state s (current value is immediate reward plus value of next state)

- $p_{a,0 \rightarrow s}$ probability of getting to state s from state 0 , given that you perform action a

- To select action with best value we need reward matrix, $r_{s,a}$ and action transition matrix $p_{a,0 \rightarrow s}$



Defining the Q-Function

$$V_0 = \max_{a \in A} \mathbf{E}[r_{s,a} + \gamma V_s] = \max_{a \in A} \sum_{s \in S} p_{a,0 \rightarrow s} \cdot (r_{s,a} + \gamma V_s)$$

- Define intermediate function Q

$$Q(s, a) = \sum_{s' \in S} p_{a,s \rightarrow s'} \cdot (r_{s,a} + \gamma V_{s'})$$

- With some nice properties/relations:

$$V_s = \max_{a \in A} Q(s, a)$$

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



Value Iteration (Value Based)

- **Direct:**

- Initialize $V(s)$ to all zeros
- Take a series of random steps
- Perform for each state:
$$V(s) \leftarrow \max_{a \in A} \sum_{s' \in S} p_{a,s \rightarrow s'} \cdot (r_{s,a} + \gamma V(s'))$$
- Repeat until $V(s)$ stops changing

- **Q-Function Variant:**

- Initialize $Q(s,a)$ to all zeros
- Take a series of random steps
- For each state and action:
$$Q(s,a) \leftarrow \sum_{s' \in S} p_{a,s \rightarrow s'} \cdot (r_{s,a} + \gamma \max_{a'} Q(s', a'))$$
- Repeat until Q is not changing

Need to estimate $p_{a,s \rightarrow s'}$
Via observed **Transitions**

This Update Will **Converge to Optimal Policy**





Value Iteration Reinforcement Learning

M. Lapan Implementation for
and Frozen Lake

Follow Along:
`08a_Basics_Of_Reinforcement_Learning.ipynb`



Some Limitations

- Q function can get really big for **large states and action spaces**
- Infinite when the spaces are **continuous**
 - We will solve this by using a neural network to **approximate** the Q function
- Transition matrix, similarly, can get gigantic for large state and action spaces
 - We will solve this by dropping the transition probabilities in Q function update
- This Variant is known as Q-Learning



Atari Paper Presentation

Playing Atari with Deep Reinforcement Learning

Volodymyr Mnih Koray Kavukcuoglu David Silver Alex Graves Ioannis Antonoglou

Daan Wierstra Martin Riedmiller

DeepMind Technologies

`{vlad,koray,david,alex.graves,ioannis,daan,martin.riedmiller} @ deepmind.com`



Lecture Notes for **Neural Networks and Machine Learning**

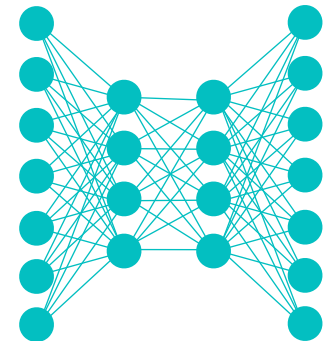
CE and Value Iteration



Next Time:

Deep Q-Learning

Reading: Lapan CH6, CH7

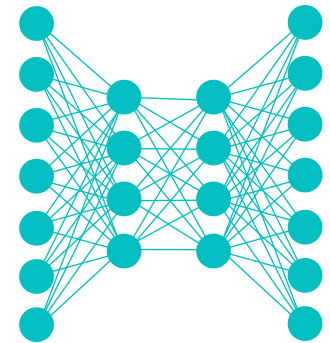




Lecture Notes for **Neural Networks and Machine Learning**



Deep Q-Learning



Last Time

How to Make this more Mathy?

- If we have all possible policies $p(x)$ and a reward function $H(x)$, then maximize

$$\mathbf{E}_{x \sim p(x)}[H(x)] = \mathbf{E}_{x \sim q(x)}\left[\frac{p(x)}{q(x)} H(x)\right]$$

- We can approximate the distribution by: $\frac{1}{N} \sum_i \frac{p(x_i)}{q(x_i)} H(x_i)$
- Proven that this is optimized when $\text{KL}(q(x) \parallel p(x)H(x))$ is minimized. But its intractable, so we drop terms ... and end up just optimizing (neg) cross entropy of samples

$$\pi_{k+1} = \arg \max_{\pi_k} \mathbf{E}_{z \sim \pi_k} [\mathbf{1}_{R(z) > \mu'} \log \pi_k]$$

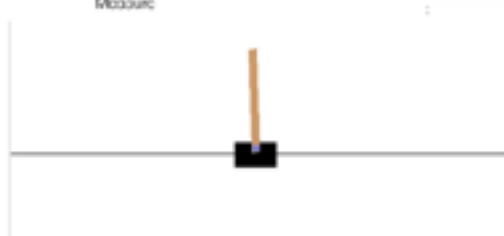
Performance
Measure

```
import gym

if __name__ == "__main__":
    env = gym.make("CartPole-v0")

    total_reward = 0.0
    total_steps = 0
    obs = env.reset()

    while True:
        action = env.action_space.sample()
        obs, reward, done, _ = env.step(action)
        total_reward += reward
        total_steps += 1
        if done:
            break
```

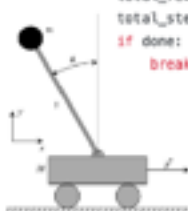


Action Space: One input, [0, 1] pull left or pull right

Obs Space: Dynamic state variables (continuous and four dimensional)

End: When more than 15 degrees off or too far from center

Reward: +1 for each time step



Value Iteration (Value Based)

• Direct:

- Initialize $V(s)$ to all zeros
- Take a series of random steps
- Perform for each state: $V(s) \leftarrow \max_{a \in A} \sum_{s' \in S} P_{s,a,s'} \cdot (r_{s,a} + \gamma V(s'))$
- Repeat until $V(s)$ stops changing

• Q-Function Variant:

- Initialize $Q(s,a)$ to all zeros
- Take a series of random steps
- For each state and action: $Q(s,a) \leftarrow \sum_{s' \in S} P_{s,a,s'} \cdot (r_{s,a} + \gamma \max_{a'} Q(s',a'))$
- Repeat until Q is not changing

Need to estimate $p_{a,s \rightarrow s'}$
Via observed **Transitions**

This Update Will **Converge to Optimal Policy**

Defining the Q-Function

$$V_0 = \max_{a \in A} \mathbf{E}[r_{s,a} + \gamma V_s] = \max_{a \in A} \sum_{s' \in S} P_{s,a,s'} \cdot (r_{s,a} + \gamma V_{s'})$$

- Define intermediate function Q

$$Q(s,a) = \sum_{s' \in S} P_{s,a,s'} \cdot (r_{s,a} + \gamma V_{s'})$$

- With some nice properties/relations:

$$V_s = \max_{a \in A} Q(s,a)$$

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



Q-Learning



Tabular Q-Learning Algorithm

- In update, ignore the transition probability, making use of the iterative nature of Q:

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

- Add momentum to the update equation

$$Q(s, a) \leftarrow (1 - \alpha) \cdot Q(s, a) + \alpha \cdot [r_{s,a} + \gamma \max_{a' \in A} Q(s', a')]$$

- Algorithm:
 - Sample (with rand) from environment, (s, a, r, s')
 - Make **Bellman Update** with Momentum
 - Repeat until convergence





Tabular Q-Learning Reinforcement Learning

M. Lapan Implementation for
and Frozen Lake

Follow Along:

`08a_Basics_Of_Reinforcement_Learning.ipynb`

54



Q-Learning with a Neural Network

- Want to approximate $Q(s,a)$ when the state space is potentially large. Given s_t , we want the network to give us a row of actions that we can choose from:
[$Q(s_t,a_1)$, $Q(s_t,a_2)$, $Q(s_t,a_3)$, ... $Q(s_t,a_A)$]
- This allows us to make a loss function which incentivizes the actual Q-function behavior we desire from a sampled tuple (s, a, r, s')

$$\mathcal{L} = \left[\underset{\substack{\text{from current network} \\ \text{params}}}{Q(s, a)} - \left[r_{s,a} + \gamma \underset{\substack{\text{from older network params} \\ \text{(better stability)}}}{\max_{a' \in A} Q^*(s', a')} \right] \right]^2$$

Periodically Update
Params of Q^* from Q

$$\mathcal{L} = \left[Q(s, a) - r_{s,a} \right]^2$$

if no next state (env is done)



But we need more power!

- We need to do some **random actions** before following the policy or else we won't learn
- Also, we need to follow the policy more and more during training to get to better places in the environment
- **Epsilon-Greedy** Approach:
 - Start randomly doing actions with prob ϵ
 - Slowly make ϵ smaller as training progresses
- And also we need to have larger amounts of uncorrelated training batches so we will again use **experience replay**





Deep Q-Learning Reinforcement Learning

M. Lapan Implementation for
Frozen Lake

And with Atari!

Follow Along:

`08a_Basics_Of_Reinforcement_Learning.ipynb`



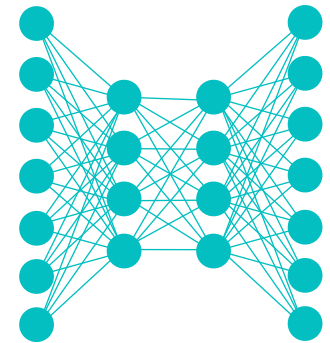
Course Retrospective



Lecture Notes for **Neural Networks and Machine Learning**

Deep Q Learning

Next Time:
Class Retrospective
Reading: None

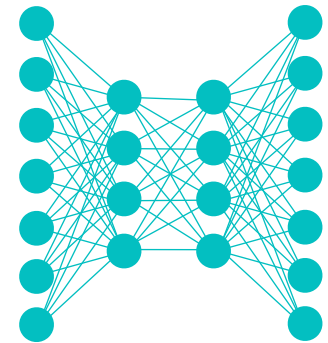




Lecture Notes for **Neural Networks and Machine Learning**



Course Retrospective



Last Time

```
import gym

if __name__ == "__main__":
    env = gym.make("CartPole-v0")

    total_reward = 0.0
    total_steps = 0
    obs = env.reset()

    while True:
        action = env.action_space.sample()
        obs, reward, done, _ = env.step(action)
        total_reward += reward
        total_steps += 1
        if done:
            break
```



Action Space: One input, [0, 1] pull left or pull right

Obs Space: Dynamic state variables (continuous and four dimensional)

End: When more than 15 degrees off or too far from center

Reward: +1 for each time step



$$V_0 = \max_{a \in A} \mathbf{E}[r_{s,a} + \gamma V_s] = \max_{a \in A} \sum_{s' \in S} p_{a,s \rightarrow s'} \cdot (r_{s,a} + \gamma V_{s'})$$

- Define intermediate function Q

$$Q(s, a) = \sum_{s' \in S} p_{a,s \rightarrow s'} \cdot (r_{s,a} + \gamma V_{s'})$$

- With some nice properties/relations:

$$V_s = \max_{a \in A} Q(s, a)$$

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

- Want to approximate $Q(s, a)$ when the state space is potentially large. Given s_t , we want the network to give us a row of actions that we can choose from:
[$Q(s_t, a_1)$, $Q(s_t, a_2)$, $Q(s_t, a_3)$, ... $Q(s_t, a_A)$]
- This allows us to make a loss function which incentivizes the actual Q-function behavior we desire from a sampled tuple (s, a, r, s')

$$\mathcal{L} = \left[\underbrace{Q(s, a)}_{\text{from current network params}} - \underbrace{\left[r_{s,a} + \gamma \max_{a' \in A} Q^*(s', a') \right]}_{\substack{\text{from older network params} \\ \text{(better stability)}}} \right]^2$$

Periodically Update Params of Q^* from Q

$$\mathcal{L} = [Q(s, a) - r_{s,a}]^2$$

if no next state (env is done)



Deep Q-Learning Reinforcement Learning

M. Lapan Implementation for Frozen Lake

And with Atari!



Logistics and Agenda

- Logistics
 - Final Paper Due during Finals
- Agenda
 - Deep Q Learning Code Review
 - Paper Presentation
 - Class Retrospective





Deep Q-Learning Reinforcement Learning

M. Lapan Implementation for
Frozen Lake

And with Atari!

Follow Along:

`08a_Basics_Of_Reinforcement_Learning.ipynb`



Paper Presentation

Dota 2 with Large Scale Deep Reinforcement Learning

OpenAI, *

Christopher Berner, Greg Brockman, Brooke Chan, Vicki Cheung,
Przemysław "Psyho" Dębiak, Christy Dennison, David Farhi, Quirin Fischer,
Shariq Hashme, Chris Hesse, Rafal Józefowicz, Scott Gray, Catherine Olsson,
Jakub Pachocki, Michael Petrov, Henrique Pondé de Oliveira Pinto, Jonathan Raiman,
Tim Salimans, Jeremy Schlatter, Jonas Schneider, Szymon Sidor, Ilya Sutskever, Jie Tang,
Filip Wolski, Susan Zhang

December 13, 2019



Course Retrospective

- Ethics: ConceptNet NumberBatch
- CNN Visualization: Filters, Heatmaps, Grad-CAM, DeepDream
- CNN Fully Convolutional: R-CNN, YOLO, Mask-RCNN, YOLACT, DeepLabV3
- Style Transfer
- Multi-task and Multi-modal
- GANs: Goodfellow to Wasserstein to LS to Cycle
- RL: Just the basics (wanted to get to World Models)
- What was good, **bad**, **ugly**? What could be **changed**?



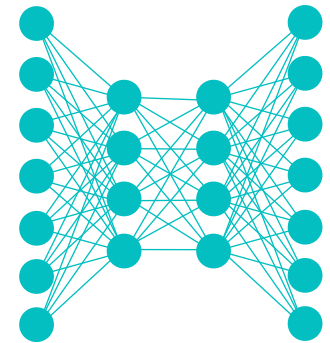
Lecture Notes for **Neural Networks and Machine Learning**

Course Retrospective



Next Time:
None!

Reading: None



Backup slides



Title Between Topics



Example Slide





Title

Subtitle

Follow Along: Notebook Name

