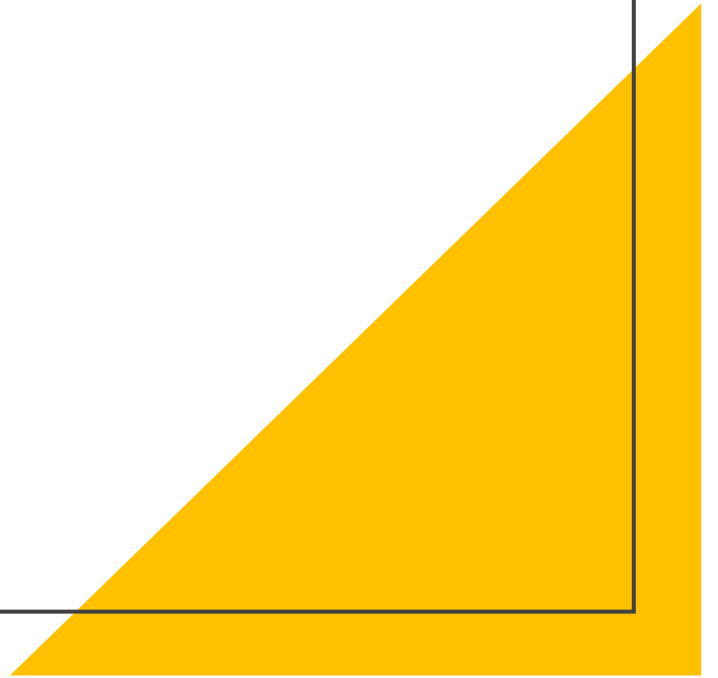


05: Reinforcement Learning

Antorweep Chakravorty



Topics

- Actor Critic
- Advantage Actor-Critic
- Distributed Training
- N-step Distributed Actor Critic

Recap



Value Function



Policy Function

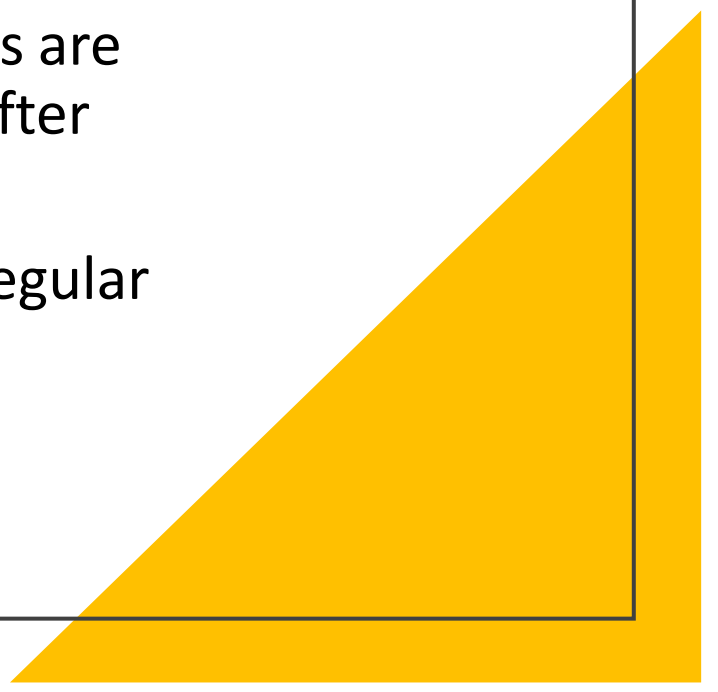
Recap

- WHITEBOARD

Recap

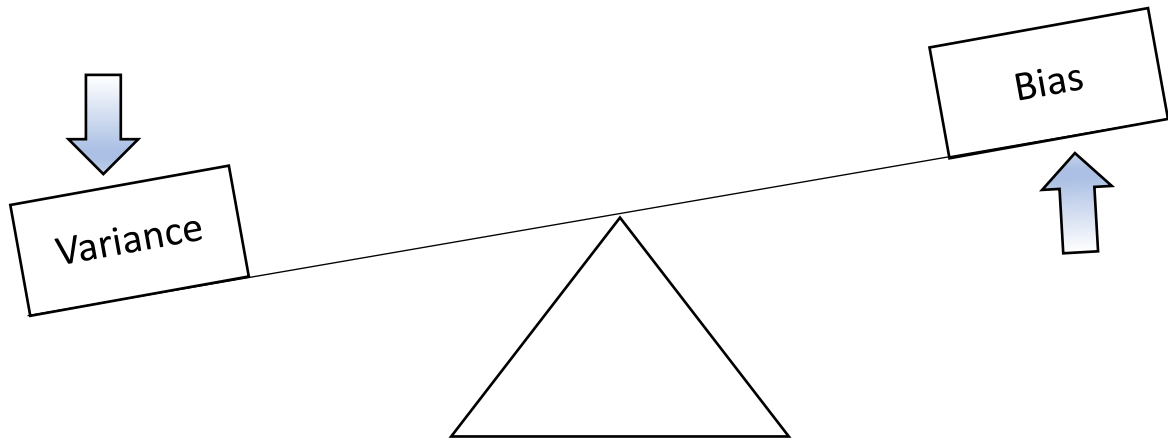
Q-Learning	REINFORCE
Value Function	Policy Function
Learns directly from the environment -> rewards	Learns by positively and negatively reinforcing actions
Predicts rewards (aka values)	Predicts actions
Models the value (expected reward) of an action given a state	Models the policy that determines the probability of choosing an action given a state
All information relevant for modeling the value is present in the input state	Samples a full episode
Intuitive way for solving Markov Decision Process (MDP)	It falls under the umbrella of Monte Carlo approaches
An online learning algorithm	An episodic learning algorithm
Can effectively handle sparse rewards by bootstrapping to predict future states based on the current.	Has a sensitive loss function that depends on a dense reward matrix after the end of an episode
Has no temporal dependence.	Learning is temporally dependent on the sequences of state actions in an episode.

Episodic and Online Learning

- Episodic Learning: Knowledge of the environment is gathered by sampling a full episode. Updates to model parameters are made at irregular intervals depending on the outcome after an episode ends.
 - Online Learning: Parameters of models are updated at regular intervals or in an *online* fashion. Does not need to have knowledge of the environment to make the updates.
- 
- A large yellow right-angled triangle is positioned in the bottom right corner of the slide, with its hypotenuse facing the top-left.

Bootstrapping

- Bootstrapping is a process of making a prediction from a prediction
- However, if the first prediction is bad, the second prediction maybe even worse.
- Bootstrapping introduces a source of **bias**. Bias is a systematic deviation from the truth.
- On the other hand, making predictions from predictions introduces a kind of self-consistency that results in lower **variance**. Variance is a lack of precision in predictions.

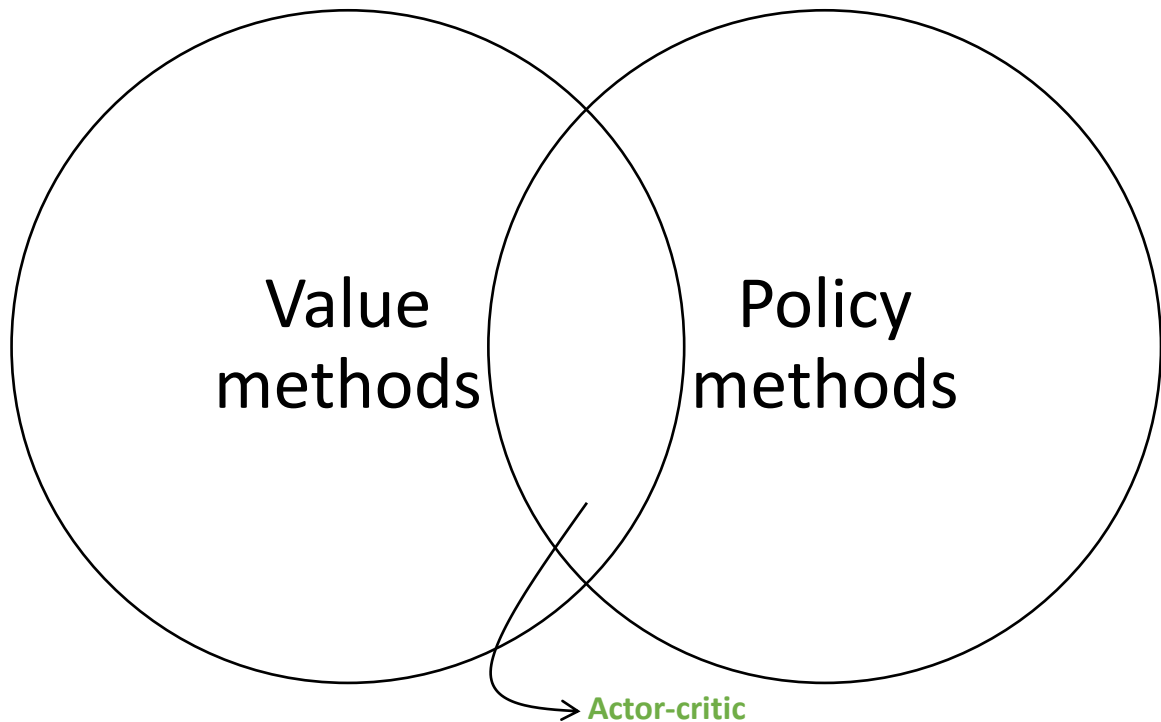


Bias vs Variance

- Key concept to all machine learning: **bias-variance tradeoff**
- Inversely proportional: Reducing bias increases variance and vice-versa
- Leads to the problem of **over and under fitting**

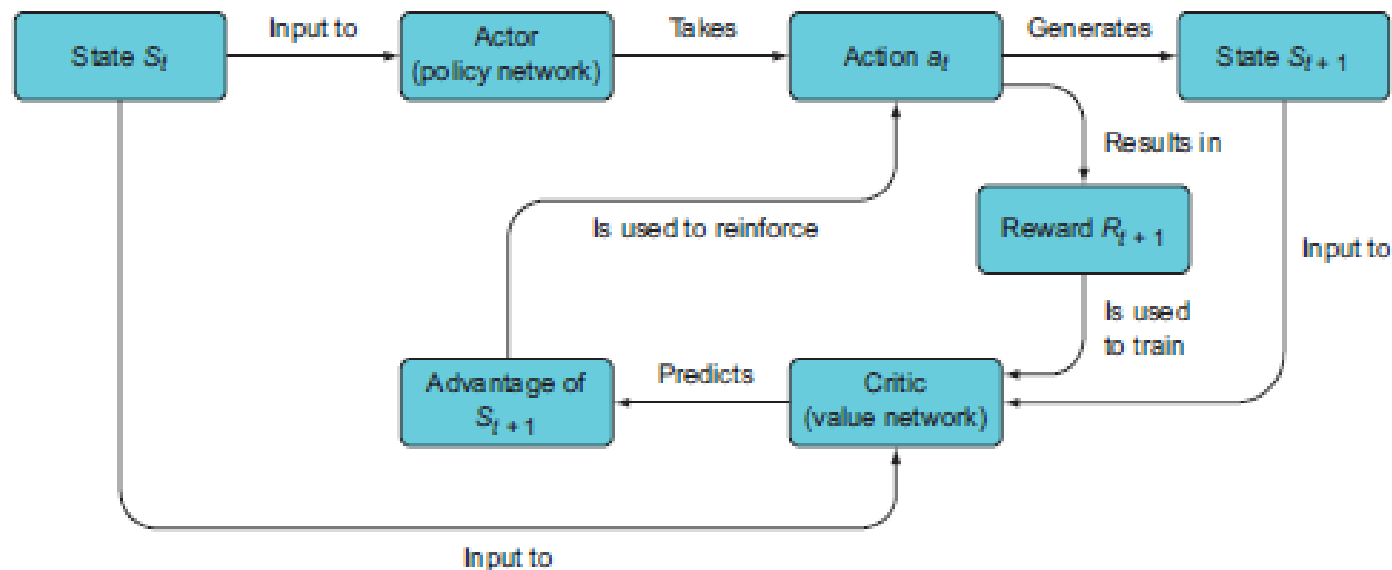
Overfitting and Underfitting

- WHITEBOARD



Combining value and policy function

$$Loss = -\log \pi(a | S) \times \underbrace{(R - V_{\pi}(S))}_{\text{advantage}}$$



Actor- Critic (A2C)

Actor-Critic (A2C)

- The actor and critic are in competition with one another
- The critic is incentivized to model the returns as best as it can (and the returns depend on what the actor does)
- However, the actor is incentivized to beat the expectations of the critic
- If the actor improves faster than the critic, the critic's loss will be high and vice-versa
- Such form of learning that demonstrates an adversarial relationship is called **adversarial training**
- Adversarial training is a powerful technique in many areas of machine learning as well
 - e.g.: **Generative Adversarial Networks** (GANs) in ML are an unsupervised method for generating realistic-appearing synthetic samples of data from training data set using a pair of models that functions similarly to an actor and critic

✗ Actor-Critic in a naïve online way

```
# Calculate the probs of each action for a state
policy = self.actor(torch.from_numpy(state_).float())
# Choose an action based on their probs.
action = np.random.choice(self.numActionSpace, p=policy.detach().numpy())
# Execute an action in the environment
nextState_, reward, done, _ = self.env.step(action)

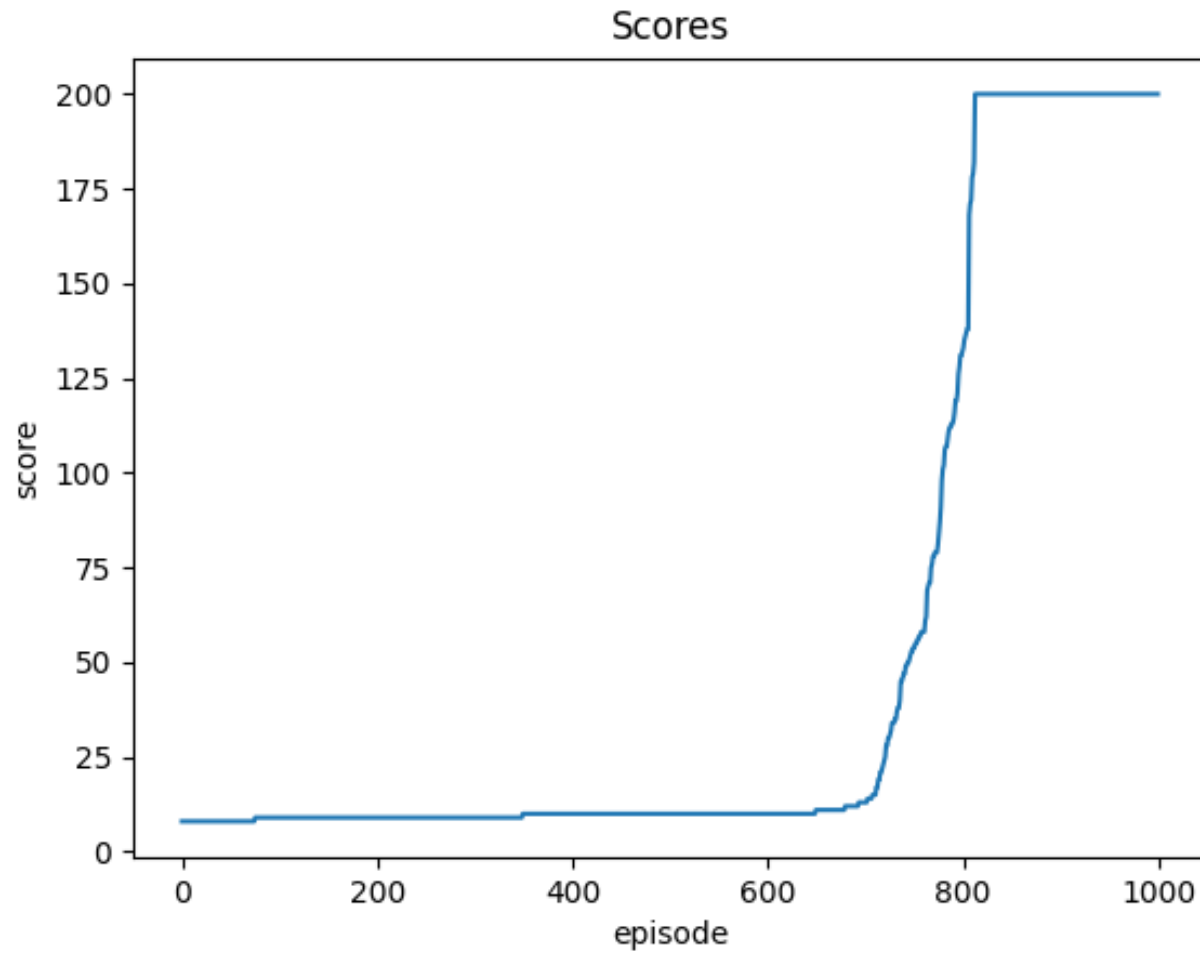
# Calculate the value for a state
value = self.critic(torch.from_numpy(state_).float())
# Calculate the value for the next state. Since we are doing online training we do not have the return for the next
state,
# therefore we bootstrap / predict the value for the next step
nextValue = torch.Tensor([0.0]) if done else self.critic(torch.from_numpy(nextState_).float())

# Value Loss
criticLoss = self.criticLossFun(value, reward + gamma * nextValue.detach())
# Policy Loss
advantage = reward + gamma * nextValue.item() - value.item()
actorLoss = self.actorLossFun(policy[action], advantage)

...

### Backpropagate the actor and critic model
```

Execute for each
step in an episode
until the end of the
episode



Actor-Critic
in a naïve
online way

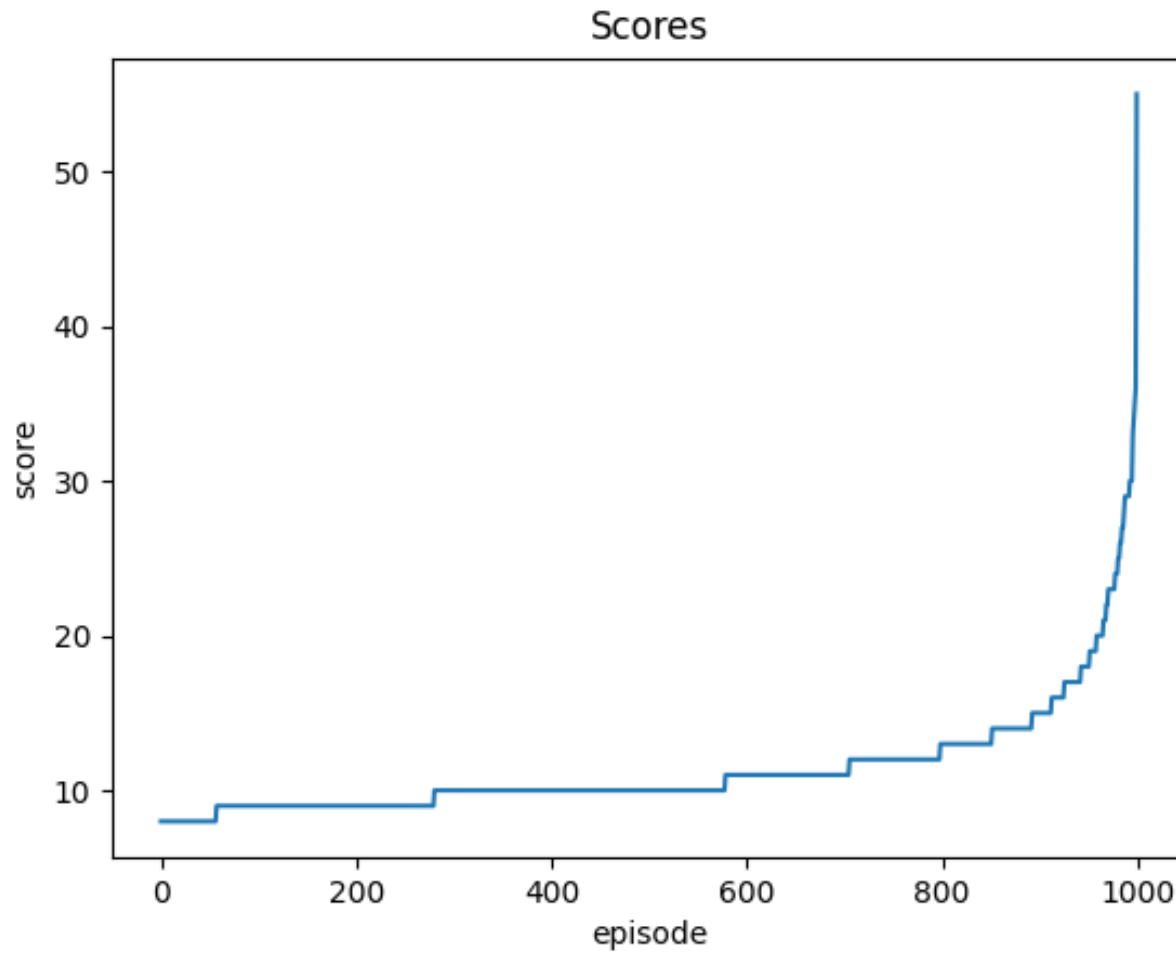
✗ Actor-Critic with experience replay in an episodic manner

```
### For each episode until termination, add episode transitions to a buffer (for policy updates) and experiences to a replay buffer (for value updates).
...
### After the end of an episode, update the actor model
# Store the state info as a batch of states
stateBatch = torch.stack([torch.from_numpy(s).float() for s in iterations[:, 0]])
# Store the action info as a batch of actions
actionBatch = torch.Tensor(list(iterations[:, 1]))
# Feed the state batch to the actor model to calculate the probs of actions for each state in the batch
policy = self.actor(stateBatch)
# Gets the probs of actions actually performed for each state
probs = policy.gather(dim=1, index=actionBatch.long().unsqueeze(dim=1)).squeeze()

# Policy Loss
actorLoss = self.actorLossFun(probs, advantage)
# Backpropagate policy
...
### If the replay buffer is larger than the batch size
# Select a set of random indices to be chosen from the replay buffer
indices = np.random.choice(len(replay), size=batch)
# Extract the experiences from the replay buffer
replay_ = np.array(replay)[indices, :]
# Create a state batch with the excted experiences
stateBatch = torch.stack([torch.from_numpy(s).float()
                           for s in replay[:, 0]])

# Calculate the value for the extracted states
value = self.critic(stateBatch)

# Value Loss
criticLoss = self.criticLossFun(value, torch.Tensor(list(replay[:, 1] + gamma * replay[:, 2])).float())
```



Actor-Critic with
experience
replay in an
episodic manner

Limitations

- Deep learning models need updates to their parameters in a batch (*batch training*).
 - Training with single pieces of data at a time would have too much variance and the parameters would never converge
 - The noise in a batch gets averaged out leading to better updates
- A2C models have temporal dependence. Using a replay buffer and sampling from it loses the temporal dependence.
- > Solution: **Distributed Training**
 - Running multiple copies of the agent in parallel, each with separate instantiation of the environment.
 - A varied set of experiences can be collected from each agent
 - Sample of the gradient's averages would lead to a lower variance
 - Eliminates the need for experience replay and allows us to train in a completely online fashion

Distributed Training

- WHITEBOARD

Multiprocessing vs Multithreading

- Two forms of parallel computations
- Multiprocessing executes processes simultaneously on multiple GPU or CPU cores
- Multithreading executes tasks simultaneously on a single core
- In multiprocessing each process has its own memory address space and may have multiple threads
- In multithreading, multiple tasks get on a core by moving between tasks when one task is waiting for some operations to finish like IO from disk
- **Machine learning** models generally have IO operation to the memory and are limited by computation speed. Therefore, ML models **benefit** from true simultaneous computations with **multiprocessing**

multiprocessing module from python

(also wrapped by pytorch)

Example 1

```
import multiprocessing as mp
...

# Takes an array and squares each element
def square(x):
    return np.square(x)

if __name__ == '__main__':
    x = np.arange(64)

    numProcessors = mp.cpu_count()

    # Set up a multi processing pool with the number less than or equal to the number of cpus available
    pool = mp.Pool(numProcessors)
    # Use the pool map function to apply the square function to each array in the list and returns the results in a list
    squared = pool.map(square, [x[numProcessors*i:numProcessors*i+numProcessors] for i in range(numProcessors)])

    print('Squared values: {}'.format(squared))
```

Outputs:

```
Values: [ 0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20 21 22 23
 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47
 48 49 50 51 52 53 54 55 56 57 58 59 60 61 62 63]
number of cpus: 8
Squared values: [array([ 0,  1,  4,  9, 16, 25, 36, 49]), array([ 64,  81, 100, 121, 144, 169, 196, 225]),
array([256, 289, 324, 361, 400, 441, 484, 529]), array([576, 625, 676, 729, 784, 841, 900, 961]),
array([1024, 1089, 1156, 1225, 1296, 1369, 1444, 1521]), array([1600, 1681, 1764, 1849, 1936, 2025, 2116,
2209]), array([2304, 2401, 2500, 2601, 2704, 2809, 2916, 3025]), array([3136, 3249, 3364, 3481, 3600,
3721, 3844, 3969])]
```

multiprocessing module from python

(also wrapped by pytorch)

Example 2

```
...
# Takes an array and squares each element
def square(i, x, queue):
    print('in process: {}'.format(i))
    queue.put(np.square(x)) # We store the squared value of each such into an queue

if __name__ == '__main__':
    processes = [] # A list to store the reference to each process
    queue = mp.Queue() # A multiprocessing queue. A data structure that is shared across all processes

    x = np.arange(64)
    numProcessors = mp.cpu_count()
    # Set up a multi processing pool with the number less than or equal to the number of cpus available
    numProcessors = numProcessors

    # Starts as many processes as specified in numProcessors. Each process performs the square function
    # with the process index, data chunk and queue as arguments
    for i in range(numProcessors):
        startIndex = numProcessors*i
        process = mp.Process(target=square, args=(i, x[startIndex:startIndex+numProcessors], queue))
        process.start()
        processes.append(process)

    # Executes after all the processes have completed
    for process in processes:
        process.join() # Wait return until all processes have completed

    # Terminates each process
    for process in processes:
        process.terminate()

    results = []
    while not queue.empty():
        results.append(queue.get()) # Pops a result sequence from the queue and adds it to the results
list
```

Outputs:

```
Values: [ 0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20 21
22 23
 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47
 48 49 50 51 52 53 54 55 56 57 58 59 60 61 62 63]
number of cpus: 8
in process: 1
in process: 0
in process: 2
in process: 3
in process: 5
in process: 4
in process: 6
in process: 7
Squared values: [array([ 64,  81, 100, 121, 144, 169, 196, 225]), array([
0,  1,  4,  9, 16, 25, 36, 49]), array([256, 289, 324, 361, 400, 441, 484,
529]), array([576, 625, 676, 729, 784, 841, 900, 961]), array([1600, 1681,
1764, 1849, 1936, 2025, 2116, 2209]), array([1024, 1089, 1156, 1225, 1296,
1369, 1444, 1521]), array([2304, 2401, 2500, 2601, 2704, 2809, 2916,
3025]), array([3136, 3249, 3364, 3481, 3600, 3721, 3844, 3969])]
```

Distributed Actor-Critic (DA2C)

- Maintain two models: actor and critic
- Create multiple clones of the models to be executed by multiple workers/cores in parallel.
- Each clone should share the same parameters. Updates from one clone are reflected in all other clones.
- Define the hyperparameters
- Each worker executes the specified number of epochs
- While in an epoch
 - Start a new episode by initializing a new state and storing it in s_t
 - Until the end of the episode
 - Compute the value of the state $v(s_t)$ and store it in a list
 - Compute the probabilities of the actions A given state s_t
 - Choose an action a . Store the probability of the action p_a to a list
 - Execute the chosen action a in the environment and receive a new state s_{t+1} and a reward r
 - Store the reward r on a list. Update state s_t as s_{t+1} : $s_t = s_{t+1}$
 - Train
 - Initialize $R = 0$. Loop through the rewards in reverse order to generate returns: $R = r_t + \gamma * R$
 - Minimize the actor loss: $-1 \gamma_t * \log \pi(a|s) * (R - v(s_t))$
 - Minimize the critic loss: $(R - v)^2$

Distributed Actor-Critic (DA2C)

```
### While an episode hasn't terminated
# Calculate the value of the state
value = self.critic(torch.from_numpy(state_).float())
# Calculate the probs of each action for the state
policy = self.actor(torch.from_numpy(state_).float())
# Choose an action based on their prob. dist.
action = np.random.choice(self.numActionSpace, p=policy.detach().numpy())
# Execute an action in the environment
state_, reward, done, _ = env.step(action)

### Store the value of the state, the prob. of the chosen action and the reward into a list
...
# Convert to tensors and also reverse the values, probs and rewards
values = torch.concat(values).flip(dims=(0,)).view(-1)
probs = torch.stack(probs).flip(dims=(0,)).view(-1)
rewards = torch.Tensor(rewards).flip(dims=(0,)).view(-1)
...
# Intermediate variable to store the return per step
ret_ = torch.Tensor([0])

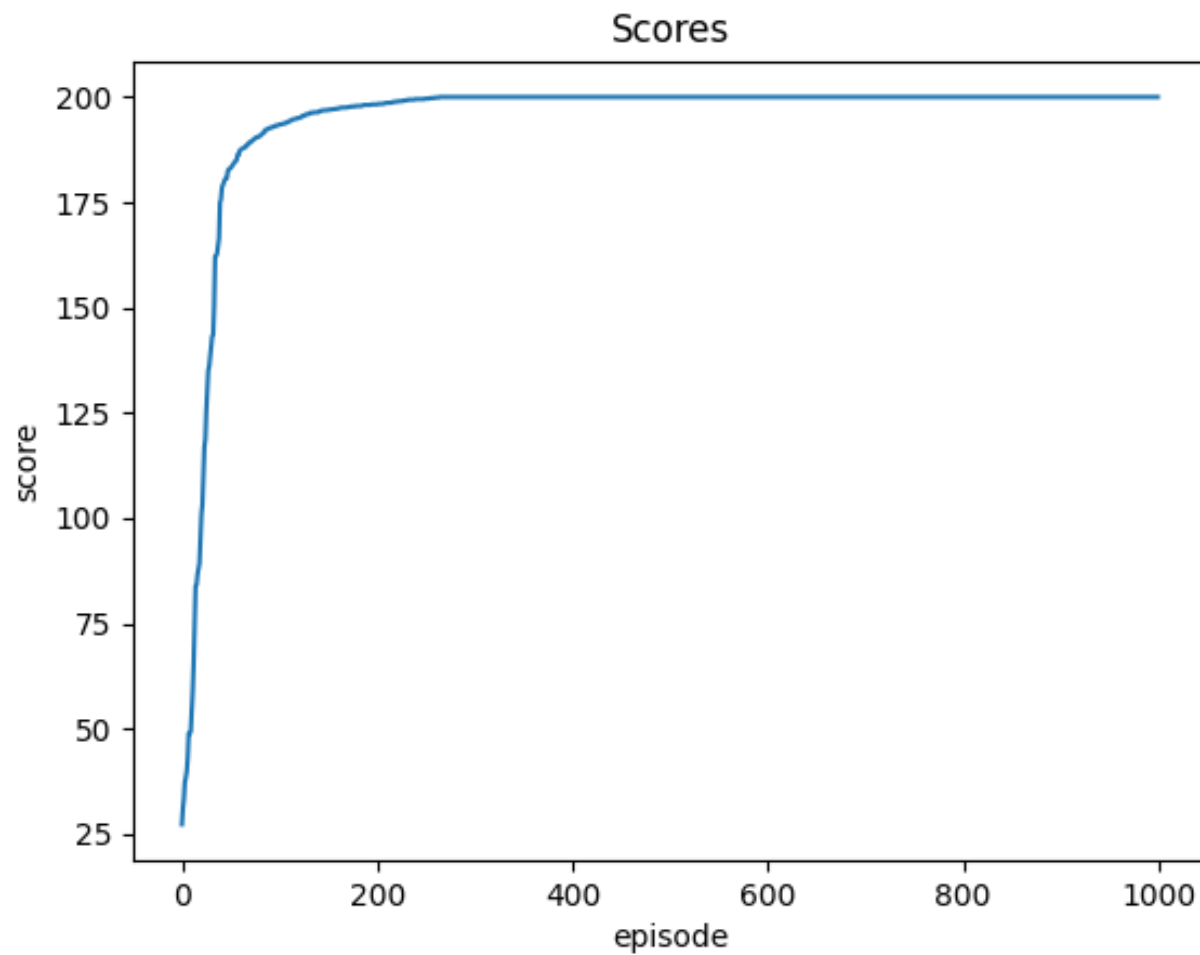
# iterate through the steps
for r in range(rewards.shape[0]):
    ret_ = rewards[r] + params['gamma'] * ret_
    Returns.append(ret_)

# Convert to a tensor
Returns = torch.stack>Returns).view(-1)

# Value Loss
criticLoss = criticLossFun(values, Returns)

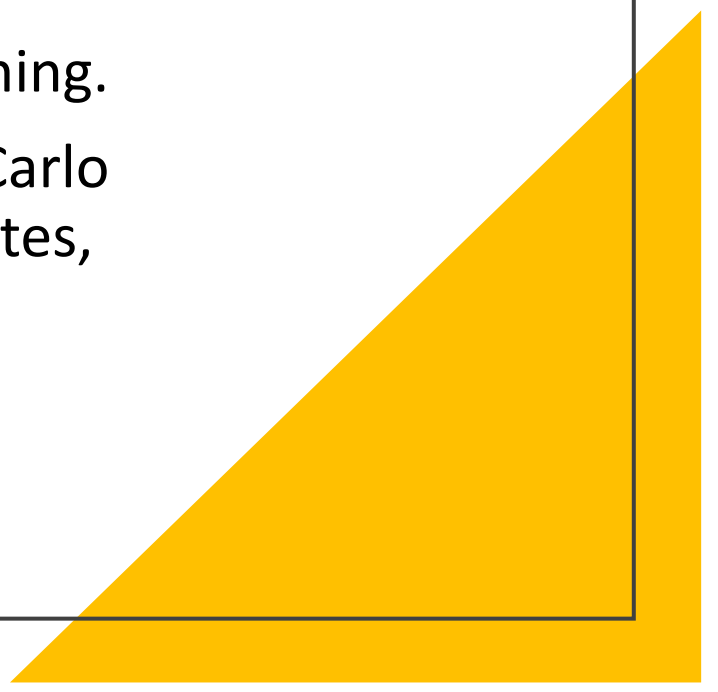
# Policy Loss
advantage = Returns - values.detach()
actorLoss = actorLossFun(probs, advantage)

### Backpropagate
```

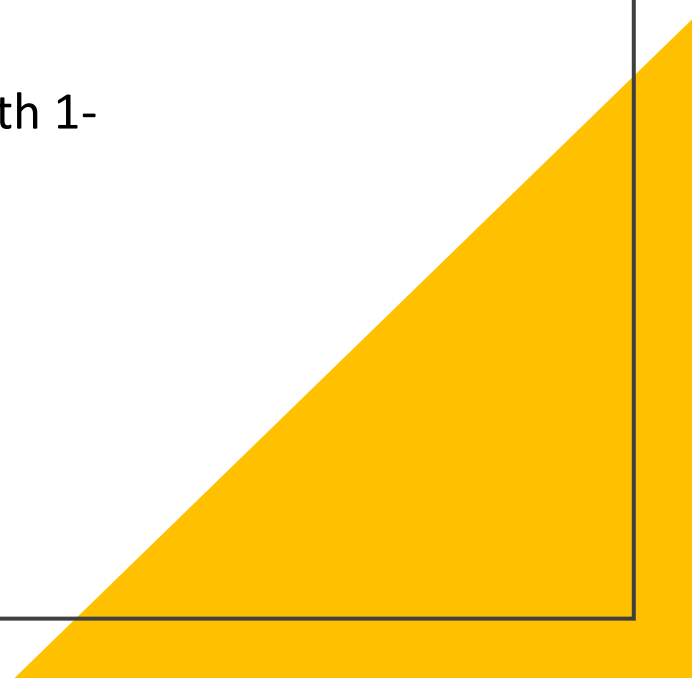


Distributed
Actor-Critic
(DA2C)

Temporal difference (TD) Learning

- Continuous RL problems without a termination state
 - Temporal Difference= Monte Carlo + Dynamic Programming.
 - TD methods sample from the environment, like Monte Carlo methods, and perform updates based on current estimates, like dynamic programming methods.
- 
- A large yellow right-angled triangle is positioned in the bottom right corner of the slide, extending from the bottom edge and the right edge towards the center.

N-Step Distributed Actor-Critic (NDA2C)

- Monte Carlo full episodic learning does not take advantage of bootstrapping since there's nothing to bootstrap
 - Bootstrapping was performed in online learning (e.g.: DQN), but with 1-step learning the bootstrap may introduce bias
 - Since bootstrapping employs prediction from a prediction, so the predictions will be better if we are able to collect more data before making them
 - NDA2C facilitates temporal difference learning
- 
- A large yellow right-angled triangle is positioned in the bottom right corner of the slide, pointing towards the top right.

N-Step Distributed Actor-Critic (NDA2C)

- Maintain two models: actor and critic
- Create multiple clones of the models to be executed by multiple workers/cores in parallel.
- Each clone should share the same parameters. Updates from one clone are reflected in all other clones.
- Define the hyperparameters, including the step size
- Each worker executes the specified number of epochs
- While in an epoch
 - Start a new episode by initializing a new state and storing it in s_t
 - Create an intermediate return variable G
 - Until the end of the episode or the number of steps is reached
 - Compute the value of the state $v(s_t)$ and store it in a list
 - Compute the probabilities of the actions A given state s_t
 - Choose an action a . Store the probability of the action p_a to a list
 - Execute the chosen action a in the environment and receive a new state s_{t+1} and a reward r
 - Store the reward r on a list. Update state s_t as s_{t+1} : $s_t = s_{t+1}$
 - If the episode ends, set G to zero else to current value $v(s_t)$
 - Train
 - Initialize $R = 0$. Loop through the rewards in reverse order to generate returns: $R = r_t + \gamma * R$
 - Minimize the actor loss: $-1 \gamma_t * \log \pi(a|s) * (R - v(s_t))$
 - Minimize the critic loss: $(R - v)^2$

N-Step Distributed Actor-Critic (NDA2C)

```
### While an episode hasn't terminated or the specified n_steps hasn't been reached
# Calculate the value of the state
value = self.critic(torch.from_numpy(state_).float())
# Calculate the probs of each action for the state
policy = self.actor(torch.from_numpy(state_).float())
# Choose an action based on their prob. dist.
action = np.random.choice(self.numActionSpace, p=policy.detach().numpy())
# Execute an action in the environment
state_, reward, done, _ = env.step(action)
### Set the intermediate return
if done:
    ...
    G = torch.Tensor([0])
else:
    ...
    G = value.detach()
### Store the value of the state, the prob. of the chosen action and the reward into a list
...
# Convert to tensors and also reverse the values, probs and rewards
values = torch.concat(values).flip(dims=(0,)).view(-1)
probs = torch.stack(probs).flip(dims=(0,)).view(-1)
rewards = torch.Tensor(rewards).flip(dims=(0,)).view(-1)
...
# Intermediate variable to store the return per step
ret_ = torch.Tensor([0])

# iterate through the steps
for r in range(rewards.shape[0]):
    ret_ = rewards[r] + params['gamma'] * ret_
    Returns.append(ret_)

# Convert to a tensor
Returns = torch.stack>Returns).view(-1)

# Value Loss
criticLoss = criticLossFun(values, Returns)

# Policy Loss
advantage = Returns - values.detach()
actorLoss = actorLossFun(probs, advantage)

### Backpropagate
```

N-Step Distributed Actor-Critic (NDA2C)

