RL TECHNICAL GUIDE

Abstract (Authored by Tyler Laurie)

Basic definitions of actor-critic networks along with research papers and how they apply to actor-critic algorithms

Table of Contents

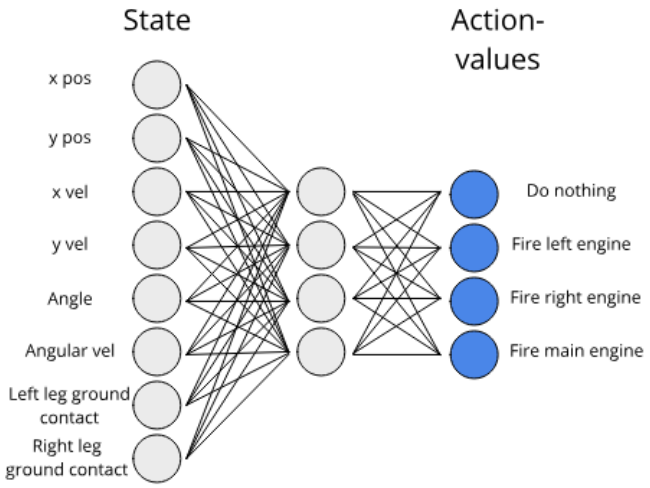
General outlines of critic-actor methods (PPO):

Actor

- The actor network refers to the neural network that is made up of different weights and biases that map states to actions. The below shows an actor network with the states as inputs into the network. As the state is passed through the network the network will output a command based on

the previously learned state and action that was taken. The policy needs a way to update how well it is choosing such actions which brings in the Generalized Advantage Estimator (GAE).

- The basis of the advantage function (from GAE) is to update the policy based on the action that was taken. This is done by choosing an action and observing what the reward is from that action and comparing it to the mean of actions that has been taken over time. In other words, the advantage function measures whether the action taken is better or worse than the policy's default behavior which has been learned overtime. This is primarily done by measuring the difference between the value network's prediction in the discounted expected reward vs the actual discounted reward (target value – this will be explained later in the Bellman update)
  - I.e. if the drone is on the ground and the actions it can take are (where I denotes motor on and 0 denotes motor off for simplicity):
    - Motor 1: I Motor 2: 0 Motor 3: 0 Motor 4: 0
    - Motor 1: I Motor 2: I Motor 3: 0 Motor 4: 0
    - Motor 1: I Motor 2: I Motor 3: I Motor 4: 0
    - Motor 1: I Motor 2: I Motor 3: I Motor 4: I
  - If overtime the drone learns to do action 4 (all motors on) then if, for example, it takes action 2 and only turns on two motors this action will have a lesser reward compared to that of action 4. This would tell the policy to lessen this weight/bias towards this action and decrease the likelihood of taking this action in the future.
- Updating the actor is essentially moving its probability mass towards higher quality actions by using gradient descent on the policy's loss function
  - The policy's loss function is a function of the surrogate loss (action loss) and the value loss function which will be further explained later in this guide

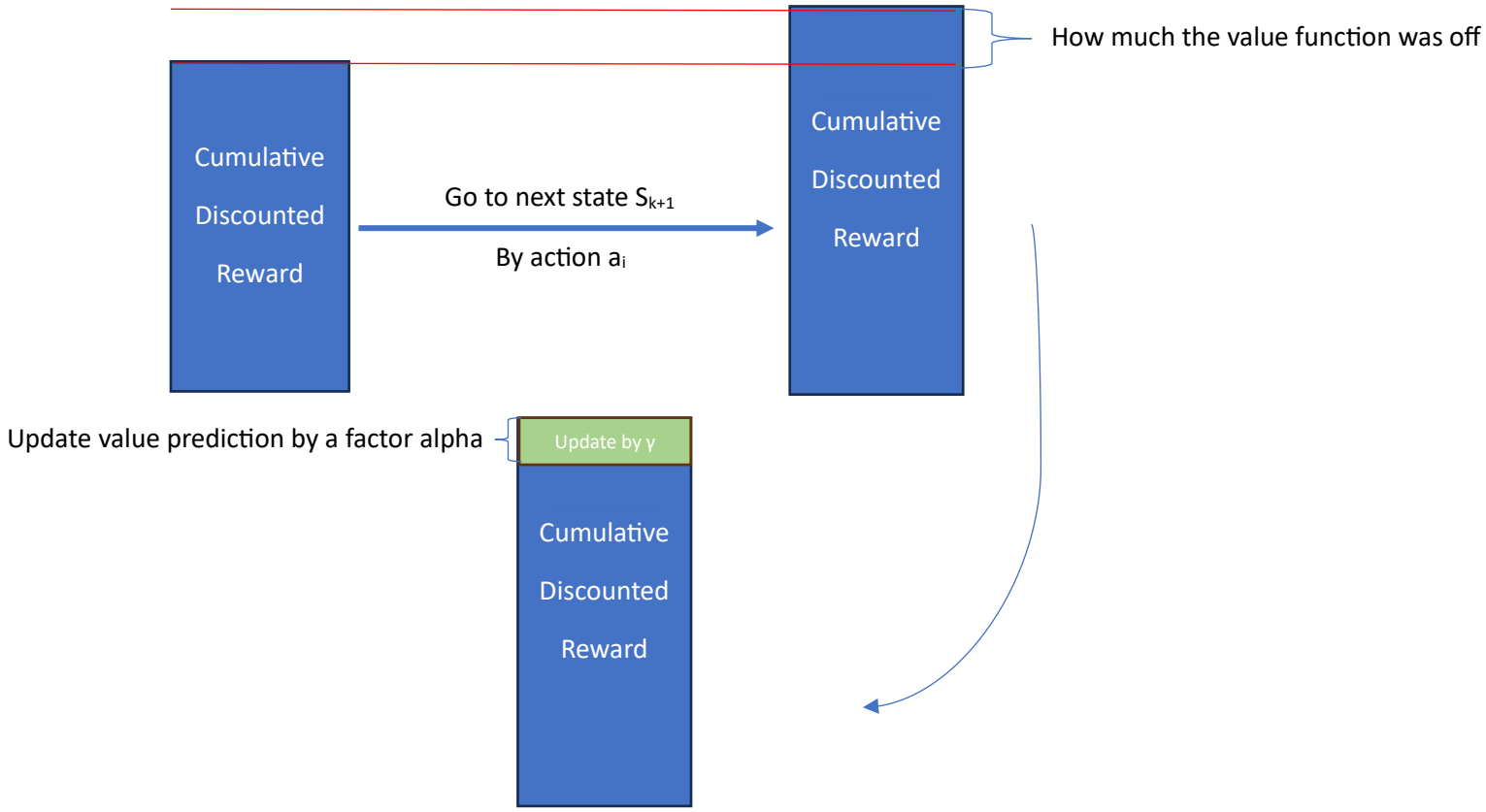


- In a technical sense, the GAE is using a discounted estimation of k number of advantage functions into the future to gather information on how the current action taken could lead to a plethora of different states and actions taken in the future.

Critic

- The critic network refers to a neural network that is predicting the cumulative discounted reward of the state-action pairs. This is the **value function** that is learned by the critic and uses the mean squared error as the loss function to optimize.

- The critic network is updated by the Temporal Difference error method which does the following:
    - At a state S predict the return for a given action which includes the reward and the discounted cumulative future reward
    - Step into the next state with this action
    - Retrieve the reward and the discounted cumulative rewards with the old policy
    - Calculate the difference between the predicted value and the actual value of the above



- The method of TD estimation replaces that of a monte carlo approach which entails sampling every action for the next state along with all subsequent states and actions therein after. The TD estimation can be projected for the next state only (TD(0)), the next two states (TD(1)), or for many states (TD(λ)) which can converge at large amount of states to that of monte carlo. The benefit of having the hyperparameter lambda is to be able to emphasize different segments of training.
    - For example, if having an agent that learns from more recent events then a TD(0) method would be used as compared to a method that would focus on the past experiences of the agent would be TD(1) which becomes more akin to the monte carlo method.
    - TD(λ) where λ is a hyperparameter [0,1] allows for emphasis of each method.

Critic influencing the actor

- A main idea of the actor-critic dynamic is that the critic network can help to guide the actor network in policy guidance while also learning the discounted value of a given state to aid in learning dynamics
- The loss function of the overall algorithm = policy_loss + value_loss

- The influence by the critic is done using the advantage function which measures the difference between the critic's cumulative reward prediction of a current state and what the actual cumulative reward was at this state which is given by the equation:
  - $rt+\gamma V(st+1)-V(st)$
  - Where rt is the actual reward of taking an action into a state
  - $V(st+1)$ is the **predicted** cumulative reward of the next state
  - $V(st)$ is the **actual** cumulative reward of the current state

Flow of data in actor-critic network

State and Action:

- The agent observes a state (S) and selects an action based on its policy
- Trajectory for Critic Network from a distributional point of view
  - This replaces the simple mean projection with a value distribution to aid in a more robust prediction of the actual value
  - A certain number of future states (a trajectory) are considered for evaluating the value distribution. This trajectory is influenced by the chosen action and the policy.
- Critic Network Calculates Value Distribution:
  - The critic network estimates the value distribution at each state, which is modeled as a mixture of Gaussian distributions
  - Each Gaussian is characterized by a mean, a variance, and a weight
- Network Outputs for Mixture Components:
  - The mean can be a direct output from the neural network
  - The variance is typically output from the network and passed through a Softplus activation function to ensure it's positive
  - The weights are output from the network and passed through a Softmax activation layer to ensure they sum up to one and form a valid probability distribution
    - Example: For 3 Gaussians, the network would output 9 values (3 means, 3 variances, 3 weights), assuming a simple architecture where the output layer directly represents these parameters.
- Network Architecture:
  - Network architecture example: takes an input observation (e.g., a 25-dimensional array from the drone), processes it through layers (e.g., two layers with 64 units each), and outputs 9 values to represent the parameters of the 3 Gaussians for the value distribution
- Comparison of Predicted and Actual Distributions:
  - This comparison is used to compute a loss that measures the discrepancy between these distributions
- Policy and Critic Update:
  - The policy is updated using a method like the Generalized Advantage Estimation (GAE), which helps in balancing bias and variance in the advantage estimates
  - The critic is updated by minimizing a loss function, such as the Cramér distance, which measures the distance between the predicted and empirical distributions which is found as the "target distribution" via the Bellman Operator
    - The Bellman Operator has the critic network predict the value of the next state and then the agent takes an action into the next state
    - The agent then sees the actual value of that next state which now becomes the "target distribution"

        - This then updates recursively for every state the agent encounters
- Loss Function for Critic:
    - The Cramér distance is a suitable choice for comparing distributions as it is sensitive to the difference in shapes between the predicted and empirical distributions, leading to more effective learning of the value function.

Revisiting Gaussian Mixture Critics in Off-Policy RL

TLDR:

- Improves critic performance by better representation learning
- Initial scaling of gaussian components impacts performance while number of gaussians and location initialization does not (initialize at the origin)
- The use of cross-entropy as a loss function proves to not be a norm and therefore does not guarantee contraction for the Bellman operator
    - Plan to try to use the 2-Wasserstein and Cramèr distances instead

Technical points:

- The Wasserstein metric is not used due to the Bellman Operator not being a gamma-contraction which means the resultant value distributions are not guaranteed to be minimized in distance from each other
- Uses mixture of gaussians to parameterize the distribution Z_π because MoG can be sampled from and has a differentiable log-probability density function
- Neural network Z_θ to approximate $Z_\pi^{(s,a)}$ and use $Z_{\theta(bar)}$ as a target network to estimate bootstrap targets to stabilize Q-learning
    - Z replaces the value function V
- Utilizing a distributional loss approach by applying the Bellman operator to the distribution
    - The Bellman operator is a means of shifting the distribution and / or reshaping the distribution so as to compare the returns of an action compared to its successor actions in a specific state (i.e. how much different is the distribution compared to what it previously was at this state)
- Use of the cross-entropy loss to measure the distance between the model's predicted returns given an action and the actual returns given an action using unobserved next actions according to the policy
    - $-(1/B)*\sum H(E[_{a'\sim\pi}[r+\gamma Z_{\theta^-}{}^{(s',a')}]; Z_\theta^{(s,a))}]$
        - $Z_\theta^{(s,a))}$ is the model's prediction
        - $r+\gamma Z_{\theta^-}{}^{(s',a')}$ is the actual discounted rewards given by the critic
    - Other papers highly recommend Cramèr distance instead since it creates unbiased sample gradients whereas the Wasserstein creates biased sample gradients
        - The computation of the Wasserstein distance involves solving an optimization problem that can introduce biases in the gradients, especially when approximations or constraints are applied. In practice, to enforce the Lipschitz constraint required for the Wasserstein methods such as weight clipping, or gradient penalty are employed. These methods can inadvertently introduce biases in the gradients because they affect the underlying function space, potentially limiting the diversity of generated samples or skewing the learned distribution in a way that does not accurately reflect the target distribution.

        - On the contrary, the Cramèr distance does not require enforcing a Lipschitz constraint like the Wasserstein distance, potentially allowing for a more straightforward and unbiased estimation of gradients. The comparison based on CDFs can be inherently more stable and less prone to the introduction of biases through optimization constraints or penalties.

Main points:

- On simple environments the MoG approach performed the same as C51 (categorical distribution) and the non-distributional approach (using a scalar value of mean squared error)
- Performs better than or similar to its categorical counterpart C51 and outperforms non-distributional in every category of complex environments
- The increase in performance compared to the non-distributional approach is from 1.) the learning of features by the MoG and 2.) the adaptive reweighting of instances (better learning of the value distribution)
- Tested 1,3,5, and 10 MoGs to see that there is a somewhat diminishing return after going from three gaussians on upward. One gaussian to three was the biggest jump in performance while 3,5, and 10 performed similar
- Initial scaling of the mixture components does seem to impact performance where 1 performs the worst and 0.0001 performs the best out of all the tests
    - Where the initial scale of the mixture components refers to the initialization of the means, sigmas, and alphas

How the paper impacts PPO

- Changes the critic's value function to a value distribution
- Instead of using MSE as a loss function for the value function uses the cross-entropy loss to measure the distance between the predicted value distribution and the target (actual) value distribution
- Changes the loss function of the critic network that involves the cross-entropy loss and the distributions as mentioned above

Future work:

- Using the 2-Wasserstein and Cramèr distances as loss functions

Referenced papers for further reading:

- GMAC: A Distributional Perspective on Actor-Critic Framework
    - Uses PPO and the Cramèr distance

GMAC: A Distributional Perspective on Actor-Critic Framework

TLDR:

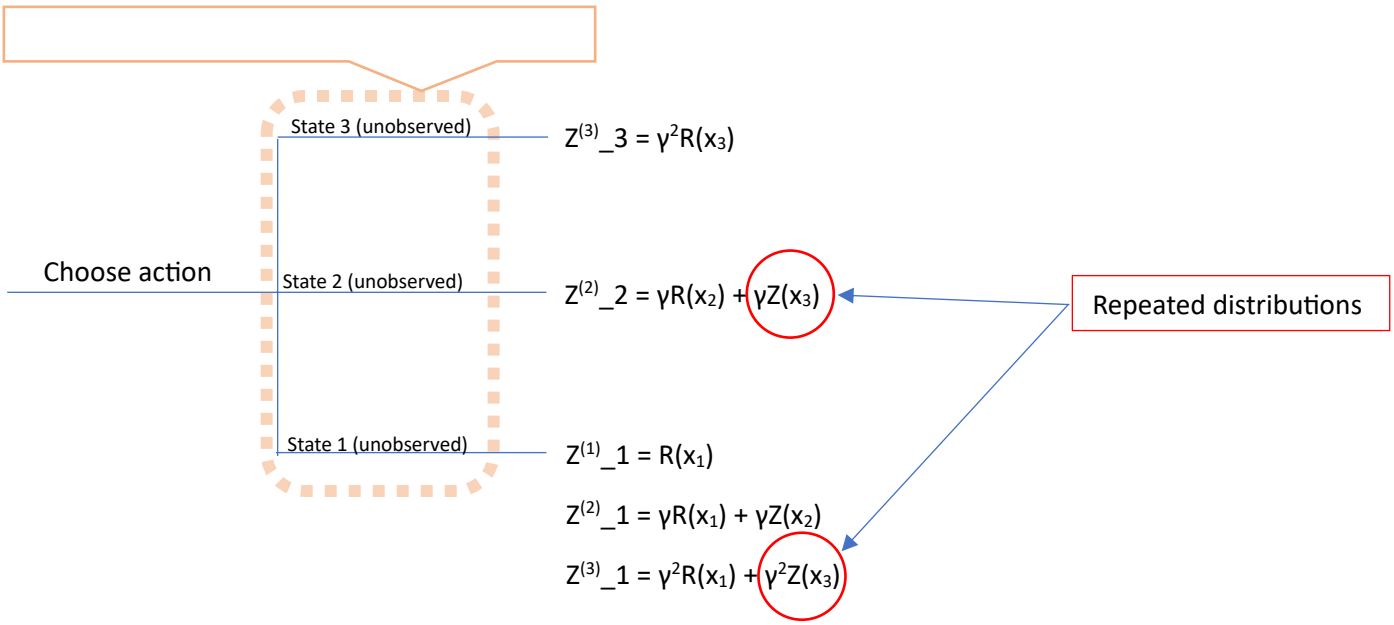
- Improves critic performance by better representation learning; their sample replacement methodology is to have a "n-step return" method but with distributions instead of a trajectory of expected values which is akin to the temporal difference training method

Technical points:

- The value distribution is learned through the distributional Bellman Operator ($\mathcal{T}^{\pi}$)

    - The Bellman Operator is only a fancy way of saying the value distribution is moved such that the predicted distribution becomes the actual distribution by continuous updates that maximize the possible value by sampling states and their respective actions
        - The Bellman Operator, in this case, **only updates** the predicted value distribution
        - This is merely a way for the critic network to update its predictions where the Bellman target is the actual value distribution
    - Instead of sampling all possible next states and actions Z(X', A') which would lead to infeasibility issues, this paper instead approximates the next state value distribution Z(X')
        - This makes sense since the PPO algorithm uses the next state's value estimate and not the state-action value estimate
- Transferring a scalar reward to a distributional reward provides more information for the neural network to learn on (behavior and characteristics of the value distribution)
    - i.e. the preservation of multimodality
- Uses mixture of gaussians to parameterize the distribution Z_π because MoG can be sampled from and has a differentiable log-probability density function
- Uses a "Sample-Replacement" algorithm which has the basis of turning the Temporal Difference method from a scalarized value to a distribution
    - This is done by transferring the target value (current reward + discounted reward) to the target distributional value (current reward + discounted distributional reward)
        - The discounted distributional reward is a little difficult in that taking the expectation of the future rewards does not give a scalar as is normal, but instead gives the expectation of multiple distributions
        - Therefore, this is handled by taking a weighted sum of the multiple distributions, since the expectation is a linear operator, to give the final cdf of the discounted value
        - This gives the appropriate TD(λ) of the distribution
        - To mitigate taking the infinite trajectory of each state, the truncated sum is often used given a desired trajectory length of N
    - Reduce the need to compute N^2 different distributions by first estimating the probability distribution
- The Cramèr distance comes into play for telling how far apart the empirical distribution and the approximated distribution is
    - Since the Cramèr gives an unbiased sample gradient we can use stochastic gradient descent to minimize the distance between the two
    - The squared Cramèr distance is equal to one half of energy distance
- Clarification point:
    - What this sample-replacement methodology does is increase efficiency by computing all of the distributions beforehand while updating only a percentage of them as the Bellman Operator is being performed
        - This prevents the looping through of each distribution and creating distirbutions that have already been used – as seen below the $Z(x_i)$ are repeated
            - 1-step return: $Z^{(1)}\_1 = R(x_1)$
            - 2-step return: $Z^{(2)}\_1 = R(x_1) + \gamma Z(x_2)$
                - $\gamma Z(x_2) = R(x_1) + Z(x_2)$
            - 3-step return: $Z^{(3)}\_1 = R(x_1) + \gamma Z(x_3)$
                - $\gamma Z(x_3) = R(x_1) + Z(x_2) + Z(x_3)$

- 4-step return: $Z^{(4)}\_1 = R(x_1) + \gamma Z(x_4)$
  - $\gamma Z(x_4) = R(x_1) + Z(x_2) + Z(x_3) + Z(x_4)$



- Solution for above stuck point
  - They come up with a trajectory of states sampled from the policy (let's say 3)
  - For each of these three states there will be distributions calculated as follows:
    - State 1 will have three distributions: S1 (itself), S1 → S2, S1 → S3
    - State 2 will have two distributions: S2 (itself), S2 → S3
    - State 3 will have one distribution: S3 (itself)
- Consider the parameterized model of the value distribution as a MoG (mixture of Gaussians)
  - $Z_\theta(x_t) \sim \sum^K w_i(x_t)\, N(z|\mu_i(x_t), \sigma_i(x_t)^2)$, where $\sum^K w_i(x_t) = 1$
  - The energy distance has the following form (related to Cramèr)
    - $E(U, V) = 2\delta(U, V) - \delta(U, U_0) - \delta(V, V_0)$
      - Where $\delta(U, V) = \sum\_(i, j)\, w_{ui}\, w_{vj}\, E[|Z_{ij}|]$, -- which is the distance between the target distribution and the predicted distribution
      - $\delta(U, U_0) - \delta(V, V_0)$ is the internal dispersion of each distribution
        - The energy distance $\delta(U,U0)$ between U and its reference distribution U0 quantifies how much U deviates from this typical or ideal state. It takes into account both the differences in location and spread between U and U0.

      And $Z_{ij} \sim N(z|\mu_{ui} - \mu_{vj}, \sigma^2_{ui} + \sigma^2_{vj})$
  - Analytical approximation of the empirical distribution can be done as follows
    - $Z_t(\lambda) = (1/m) \sum_{i=1}^{m} N(z|\mu\_nk, \sigma^2\_nk)$
- For producing the MoG distribution itself
  - Mean was left as the output of the network
  - Variance output was run through a softplus layer

- Weights were run through a softmax layer (so their sum is 1)

Main points:

- Compare the empirical approximation (given by an analytical/observed mixture of Gaussians) to the predicted value distribution given by the SR(λ) which is the same ideology of TD(λ)
- To calculate how different the actual to the predicted distribution is they use the modified Cramèr distance and work to minimize this as the loss function
- Advantage function is remained unchanged - expectation of the value distribution
- Able to capture multimodality within the value distributions
- Reduce computational cost by using the sample-replacement method

How the paper impacts PPO

- Changes the critic's value function to a value distribution
- Changes the loss function to
  - L = Difference between predicted and actual mixture of Gaussian distributions – difference between predicted distributions – difference between actual distributions

Referenced papers for further reading:

- Statistics and Samples in Distributional RL
  - Demonstrates how to take in a collection of values for certain statistics and return a probability distribution with those statistic values

Scaling Multi-Agent RL with Selective Parameter Sharing

TLDR:

- Use of Selective Parameter Sharing (SePS) to share parameters across agents that may benefit from parameter training based on their goals and abilities
- They claim that using SePS increases sample efficiency, leads to a convergence of higher returns and quicker convergence overall

Technical points:

Main points:

- Sharing parameters effectiveness is highly dependent on the environment
- Instead of sharing parameters across all agents, they developed a method that aids in identifying select agents that will benefit from parameter sharing based on their abilities and goals
- Have a single neural network represent the roles of different agent functions proves to be a bottle neck such as in the case of waiters and cooks in a restaurant
  - Agents in certain roles forget information that is a part of other agent's roles and therefore inhibit the learning of other agents
- Break the agents into partitioned sets where $\mathcal{K} < \mathcal{N}$ but without knowing $\mathcal{K}$ nor the partitioning
  - Where $\mathcal{K}$ is the number of sets and $\mathcal{N}$ is the number of agents
  - Each agent in a cluster k uses the same updates as the shared policy π_k

Future work:

Referenced paper for further reading:

The Surprising Effectiveness of PPO in Cooperative Multi-Agent Games

TLDR:

- (Not finished)

Technical points:

Main points:

- Not to create a new algorithm, but to show with simple modifications PPO can achieve strong performance in a wide variety of cooperative multi-agent settings
- Identify and analyze five hyperparameters that govern PPO's performance
- MAPPO (Multi-agent PPO) refers to PPO with a central critic
    - IPPO (Independent PPO) with local inputs for both the actor and critic
- All agents tested are strongly homogenous agents (identical observation space and action space across all agents)

Reinforcement Learning

Learning Types

- *Central Q-learning (CQL):* a single central policy (π_c) is trained by observations of all agents and chooses actions for each by selecting joint actions
- *CQL Problems:*
    - Scalarization of zero-sum and general-sum stochastic games
    - Action space grows exponentially with a policy learning joint actions given an increase in actions and / or agents
    - Physically / virtually distributed so hard to communicate from π_c to agents and vice versa
- *Solutions to CQL Problems:*
    - Decompose π_c into multiple policies given the respective state (specific states of learning can have different policies)

    - This is achieved by giving each agent a copy of π_c and the agent will then compute its own action from a joint action
- *Independent learning (IQL):* a policy for each agent while it ignores the other agents
- *IQL Problems:*
    - Each agent learns from its own policy using only its local history and therefore misses out on information that is trained on by other agents
    - No use of agents' information which makes for a dynamic environment that is always changing
    - Transition function state non-stationarity where the other agents update their own joint actions which causes unstable learning
- *Solutions to IQL Problems:*
    - Self-play: all agents use the same policy and compete against itself to get better
    - All agents have the same learning rule which helps reduce some of the non-stationarity problem

MARL Algorithms

Joint action learning (JAL)

- Uses Temporal Difference learning to learn estimates of joint action values
    - Address the IQL problem of non-stationarity of other agents and the CQL problem of reward scalarization
    - Learn joint action value models that estimate the expected returns of joint actions in any given state
    - JAL-GT focuses on viewing the joint action values as a non-repeated normal-form game for a state

- Minimax Q-learning
    - When to use it: in two-agent zero-sum stochastic games where each agent has all relevant information, and both agents have perfect information (i.e. all possible information)
        - Good for competitive environments
        - Minimax is particularly good at making short-term tactical decisions where the immediate consequences of actions are more critical than long-term strategy
        - Bad when not all relevant information is known to all agents which causes exploitation of agents
- Policies learned do not necessarily exploit weaknesses in their opponents, but these policies are robust to exploitation by assuming worst-case opponents during training

- Nash Q-learning
    - When to use it: in a general-sum stochastic game with a finite number of agents
    - Computes a Nash equilibrium solution given a general-sum game
    - Guaranteed to reach an equilibrium point under very restrictive circumstances
        - Must try all combinations of states and joint actions

Agent modelling

- learning explicit models of other agents to predict their actions based on their past chosen actions

- Policy reconstruction
  - aims to learn models of the policies of other agents based on their observations of past actions (supervised learning problem using the past observed state-action pairs)

- Fictitious play
  - Each agent i models the policy of each other agent j as a stationary probability distribution by taking the empirical distribution of j's past actions which means it is *not* conditioned on the state, but rather past actions
  - Each agent i tries to choose their action to maximize the overall reward given the other agents j actions from previous steps

$\pi_j(a_j)$ = (number of times agent j selected action aj) / (all possible next actions)

- JAL-AM
  - Combination of Joint Action Learning (JAL) and Agent Modelling (AM) which combines
    - Aspects of learning the joint action values using temporal difference learning (from JAL)
    - With the agent model of other agents' policies and best responses to select actions for the learning agents and to derive update targets for the joint action values
  - Like fictitious play, but the empirical distribution is now conditioned on the state

$\pi_j(a_j|s)$ = (number of times agent j selected action aj in a given state) / (all possible next actions in a given state)

  - Looks at all possible actions in a state and picks the action that will give maximal value given the joint action value and the other agent's empirical distributions
  - Therefore, this method will give *best-response actions* compared to *best-response policies*

Policy-based learning

- learn policy parameters using gradient-ascent techniques

Regret matching

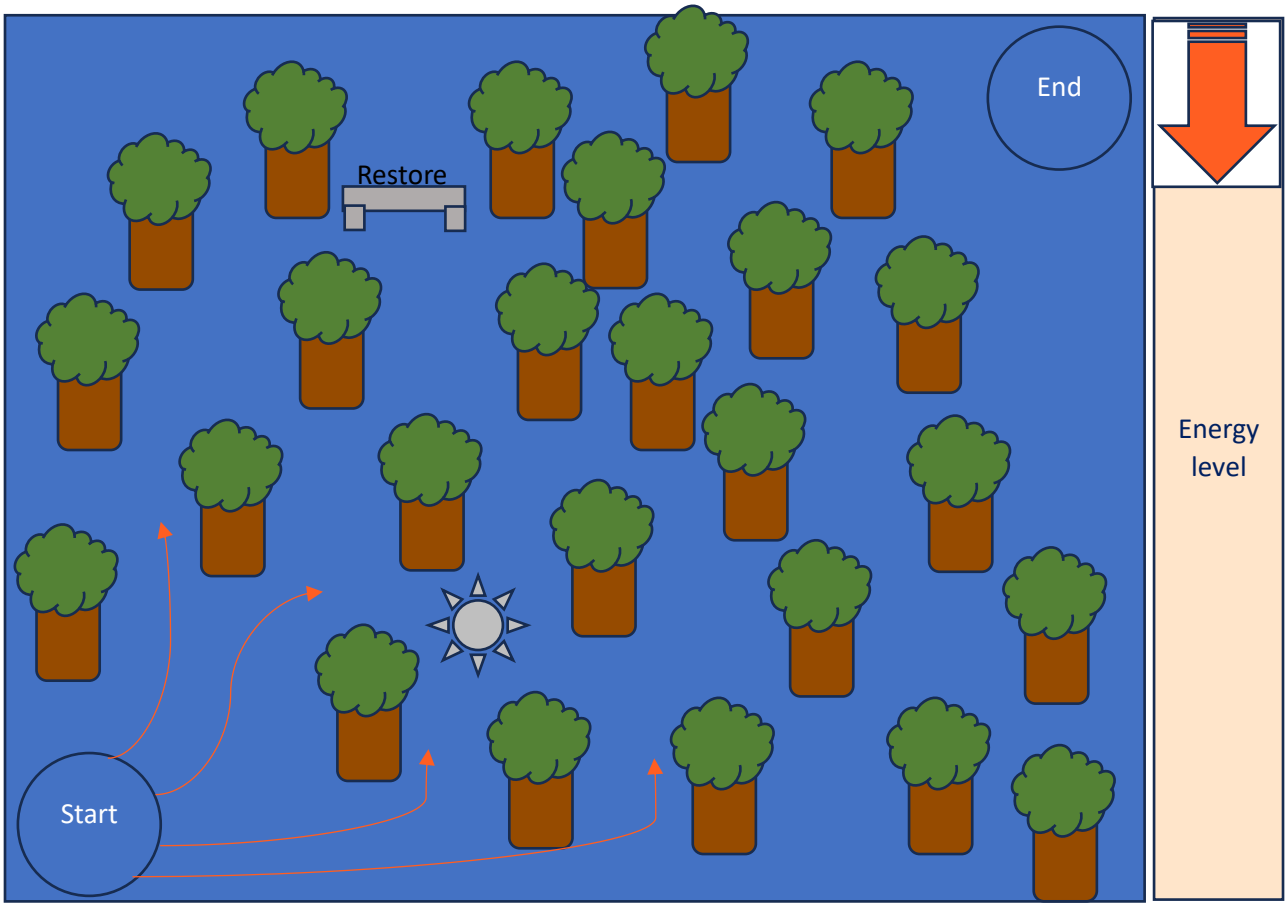
- minimize different notions of regret and can achieve no-regret outcomes

Neural Networks in MARL

- Neural networks are good generalization techniques which is good for agents in the sense that they can generalize their actions and do not need to go to a given state to learn from it
- Each scenario / game played has a certain value function in which is the "optimal" reward pathway for an agent to act
  - A neural network can change its weights and biases to map closer and closer the actual value function by iteratively going over possible actions in states
  - This ability for the neural network to change its weights allows for it to ultimately to "find" the target value function without us knowing it
  - Imagine the maze below in which a person is trying to fight their way through a forest for a quest

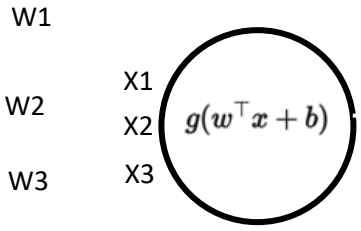
        - As the agent moves within the environment they encounter energy depletion, potential restore points, and potential harm
        - Inherently, there is an "optimal value function" where a person will be able to make their way through this forest and complete the quest with the highest energy level possible and the least amount of harm possible
    - A neural network can estimate this "optimal value function" and generalize its learning much faster compared to tabular methods
        - This is done by learning some function $f(x;\theta)$ for some input $x \in R^{dx}$
            - We take some input value x and multiply it by some value θ which can be a weight in terms of a neural network
            - $\hat{v}(s;\theta) = \theta^T x(s) = \text{sum}(\theta_k * x_k(s))$ where θ denotes parameters and x(s) represents the state features

- We can then compare the target estimation the neural network learned compared to the actual value function (given by TD learning) and minimize this difference as our loss function



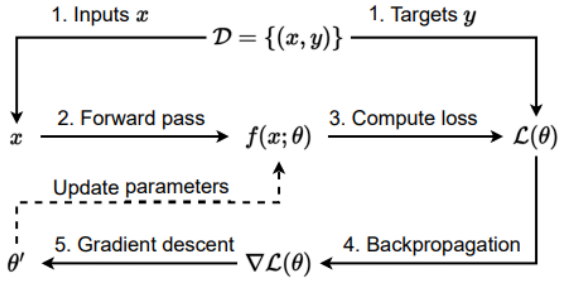
Feedforward Neural Networks

- Also called feedforward networks, fully-connected neural networks, or multilayer perceptrons (MPLs)
- Comprised of weights and biases that are ran through an activation function
  - $f_{k,u}(x;\vartheta^k_u)=g_k(\mathbf{w}^\top x+b)$



- Activation functions are used in neural networks to cause non-linearity
  - i.e. the composition of two linear functions is a linear function

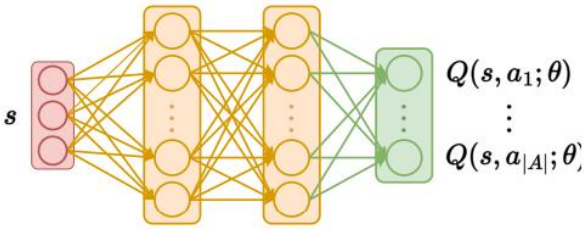
    - Taking a linear function and passing it through a non-linear function gives a non-linear function
- Since there is no closed-form solution to solve the perfect weight / bias of a neuron there exists non-convex gradient-based optimization methods to help with optimizing each weight / bias
    - Gradient-based optimization involves
        - Loss function $\mathcal{L}(\theta)$ – a way to give the network a guide on what to minimize
        - Gradient-based optimizer – which optimization technique to use
        - Backpropagation – a way to compute gradients in the network
            - Source for understanding backpropagation



Deep reinforcement learning

Incorporating neural networks into RL

- The incorporation of neural networks into value function approximation involves a few challenges, but has great benefits
    - In continuous action spaces it will be unlikely that a drone will encounter the same state and therefore these neural networks can generalize for us the value function of unseen states
    - The drawback is now we must implement the correct architecture, a loss function, and a gradient-based optimizer
- We can then transition the value function from a tabular method to a neural network by having the neural network receive the state (s) as an input and output the action-value estimates for every possible action $Q(s,a_n)$
    - For explicitness it will return a set of rewards for possible actions such as:
        - $Q(s,a_1) = 100$, $Q(s,a_2) = 45$, $Q(s,a_3) = 5$, $Q(s,a_4) = 90$



- Loss function in this case is still the squared difference between the target value y_t and the predicted value function estimate $Q(s_t, a_t; \theta)$
    - $\mathcal{L}(\theta) = (y_t - Q(s^t, a^t; \theta))^2$
    - This loss function is minimized by changing the network parameters θ

    - It is important to notice that the neural network estimated value function will be seen twice in this loss function
        - $Q(s_t, a_t; \theta)$ for the current state's reward
        - $y_t = Q(s^{t+1}, a^{'}; \theta)$ for the target value function
    - The target value function must **not** have the backward pass compute and update its gradient (modern deep RL libraries take this into account and give ways so this gradient is blocked) and is only used as an estimation target value function
        - The target value function is not apart of the gradient map since it is not being optimized, but rather it is a function of the actual reward in a system (in which there is no gradient)
- Key point: the network parameters do not change on a per state basis, but rather the connection strength is what dictates different actions per different states
    - i.e. you can freeze the network weights and the drone will still fly on its on because it learns over hundreds / thousands of parameters to generalize different states

Moving target problem

- The moving target problem is a problem in which the policy is adapting based on time
    - As the policy changes during learning the update target becomes non-stationary since the value estimate sued in the target is also changing
    - This is now exacerbated since we are approximating the value function with neural networks
        - All states and actions change continually as the value function is updated
- Combining off-policy training, with bootstrapped target values, and function approximations by the neural network can cause a *deadly triad* problem where the values of random actions can be overestimated and not get corrected which causes divergence
    - On-policy methods can fix this because an agent can visit a state and update its value function overestimation to a more correct (more correct because it is small updates at a time)

- Another way to help fix this problem is to create a neural network that will have the same architecture of the main value function, but instead it will compute the bootstrapped target values
    - The target network is initialized with the same parameters as the value function estimation network and has parameters $\bar{\theta}$
    - $Y^t = r^t + \gamma \max_{a'} Q(s^{t+1}, a'; \bar{\theta})$ where $Q(s^{t+1}, a'; \bar{\theta})$ will be the target network
    - Instead of performing gradient descent on the target network, the parameters are periodically copied from the current value function estimation from the neural network
        - Why: because we don't want the target network to diverge too far from the current network
        - However, this does help with target stability

Correlation of experiences

- The data used to train such networks is assumed to be i.i.d. (identically and independently distributed) which is mainly false in many RL forms
    - Samples are the experiences of the drone given the tuple of state, action, reward, next state which are highly correlated
    - Changes in policy change the distribution from which the experience are pulled from

- Correlations are important in the sense that if an agent learns a certain behavior, it might become specialized in the particular action and its value function parameters learn this behavior
    - If the agent has any change of state (say the drone was always initialized from the right of the objective so it learned how to land from the right and now it has a sudden initialization from the left) then it might fail since it has now changed the direction from which is it trying to land, and the value function will provide incorrect estimations
        - To mitigate this issue we can add a replay buffer $\mathcal{D}$ which takes random experiences and stores them in this buffer
            - $\mathcal{D} = \{(s^t, a^t, r^t, s^{t+1})\}$
        - Then we get a batch $\mathcal{B}$ of this buffer data uniformly and train the value function on this
        - **Key point:** the MSE is computed over the batch and minimized to update the parameters of the value function
        - **Note:** this is an off-policy method since the batch can contain experiences from the policy from recent timesteps as well as earlier timesteps

Common algorithms

DQN (Deep Q-networks) / Rainbow

- Uses a neural network to approximate the value function
- DQN uses a target network and replay buffer to mitigate against the moving target problem and correlation of consecutive samples
    - Both of which are pertinent to effective learning
- Issue with DQN: overestimation of action values since the target network computation uses the maximum action-value estimate over **all** actions in the next state where
    - $\max_a Q(s^{t+1}, a'; \bar{\theta})$ where this maximum action is likely to pick an action that is overestimated
    - This overestimation issue can be fixed by decoupling the greedy action and the value estimation
        - Which in simple terms is the main value function asking, "which action should I take?" and the target value function asking, "how good is that action?"
            - $y^t = r^t + \gamma Q(s^{t+1}, \text{argmax}_a Q(s^{t+1}, a'; \theta); \bar{\theta})$
- Double DQN addresses the problem of value overestimation by the above-mentioned decoupling of the greedy action and the value estimation
- Rainbow incorporates six extensions of DQN: decoupling of value overestimation, prioritized replay, dueling networks, multi-step learning, distributional RL, and noisy nets

Algorithm 12 Deep Q-networks (DQN)

Initialise value network Q with random parameters θ
Initialise target network with parameters $\bar{\theta} = \theta$
Initialise an empty replay buffer $\mathcal{D} = \{\}$
Repeat for every episode:
for time step $t = 0, 1, 2, \ldots$ **do**
  Observe current state s^t
  With probability ϵ: choose random action $a^t \in A$
  Otherwise: choose $a^t \in \arg\max_a Q(s^t, a; \theta)$
  Apply action a^t; observe reward r^t and next state s^{t+1}
  Store transition (s^t, a^t, r^t, s^{t+1}) in replay buffer $\mathcal{D}$
  Sample random mini-batch of B transitions (s^k, a^k, r^k, s^{k+1}) from $\mathcal{D}$
  if s^{k+1} is terminal **then**
    Targets $y^k \leftarrow r^k$
  else
    Targets $y^k \leftarrow r^k + \gamma \max_{a'} Q(s^{k+1}, a'; \bar{\theta})$
  Loss $\mathcal{L}(\theta) \leftarrow \frac{1}{B} \sum_{k=1}^{B} \left(y^k - Q(s^k, a^k; \theta) \right)^2$
  Update parameters θ by minimising the loss $\mathcal{L}(\theta)$
  In a set interval, update target network parameters $\bar{\theta}$

Policy-Gradient Algorithms

- Learning a policy as a parametrized function which can be represented using linear function approximation
- These methods compute gradients with respect to the parameters of their policy and update the learned policy
    - By transferring the policy to a learned neural network this will allow for us to use policies in the continuous action space
- Basis of the policy: receives a state as the input and outputs a scalar output for every action l(s,a) to represent the policy's action preference in a given state
    - These scalars are transformed into a probability distribution using the softmax activation function
        - $\Pi(a|s; \phi) = e^{l(s,a; \phi)} / \text{sum}(e^{l(s,a'; \phi)})$
        - Where this is the current policy's single action / the sum of all actions in a state
        - ϕ is policy parameters and θ is value function parameters
- The policy can be update via a loss function optimization in which the learned parameters increase the "quality" of the policy
    - A measure to quantify a "good policy" is the expected episodic returns which can be represented by the value of the policy in a given state $V^{\pi}(s)$ or the action value $Q^{\pi}(s, a)$

Policy-gradient theorem

- Instead of minimizing a loss function as with many neural networks, the policy-gradient theorem uses a different methodology of maximizing a state-visitation gradient of the policy
- Cannot use a replay buffer to update the policy since its expectation shows that it uses the data is generates, so if it uses old data it will be learning "bad habits" therefore it has to be on-policy

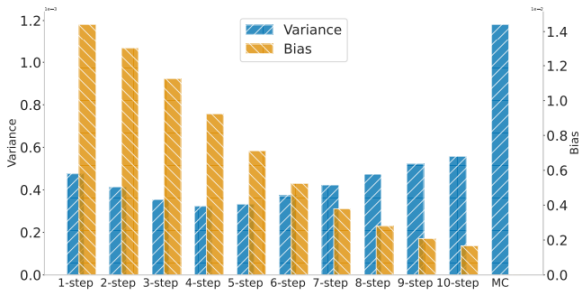
$$\nabla_{\phi} J(\phi) = \mathbb{E}_{a \sim \pi(\cdot|s;\phi)} \left[Q^{\pi}(s, a) \frac{\nabla_{\phi} \pi(a \mid s; \phi)}{\pi(a \mid s; \phi)} \right]$$

- The numerator represents the gradient of the policy pointing in the direction in parameter space that most increases the probability of repeating action a on future visits to state s
- The denominator is a normalization factor to correct for the data distribution by the policy

Actor-Critic Algorithms

- Family of policy gradient algorithms which train a parameterized policy (actor) and a parameterized value function (critic)

    - Compared to monte carlo methods, these methods use bootstrapped value estimates to optimize policy gradient algorithms
- Bootstrapped returns is an important part of these algorithms as it has two primary benefits
    - Estimate episodic returns from a single step rather than from a history of episodes
    - This method also provides *less variance* as compared to monte carlo methods since the bootstrapped returns **only depend on the current state, action, reward, and next state** but does induce an *increase in bias* however this tradeoff has shown to increase training stability
        - Whereas monte carlo methods depend on the history of the episode
    - The below graph represents the tradeoff between bias variance as once projects the returns from 1 single bootstrapped step to monte carlo
        - The takeaway is that a balance between methods can have moderately low levels of each bias and variance



A2C: Advantage Actor-Critic

- Uses the advantage function to compute estimates of which direction for the policy gradient to go
- The advantage function is given by $A^{\pi}(s,a) = Q^{\pi}(s,a) - V^{\pi}(s) = r^{t} + \gamma V(s^{t+1}) - V(s^{t})$
    - Where Q is action-value and V is state-value functions with respect to the policy
    - For the policy π, the action value $Q^{\pi}(s,a) = r^{t} + \gamma V(s^{t+1})$ is the expected returns of π when first applying action a in state s and afterwards following π
        - →i.e. returns(action→new state→trajectory of policy)
        - The immediate rewards of the policy's action
    - The state value $V^{\pi}(s) = V(s^{t})$ is the expected return when following the policy π already in a given state without following a predetermined action
        - →i.e. returns(next state→trajectory of policy)
        - The bootstrapped rewards of a state
    - The advantage function asks the question: how much better or worse is an action compared to that of an average action taken by the policy?
        - → returns(action→new state→trajectory of policy) - returns(next state→trajectory of policy)
        - Positive advantage: increase the probability the policy will choose the action again
        - Negative advantage: decrease the probability the policy will choose the action again
    - ***Key point about distributional RL:*** You do not have to update this to be the value distribution since this is for policy optimization and the main point of distributional RL is for the critic network to learn its value distribution better

        - Now this *could be* implemented but I am not sure how you could accurately compare the action value $Q^{\pi}(s,a)$ to that of the distribution without having to have $Q^{\pi}(s,a)$ as a distribution as well
            - One thought is to only return the mean of the value distribution (which is already being done)
            - Another thought is to make a mixture of Gaussians, weight them, make a final representative distribution and take the mean (I think this will just give the same mean as before)
            - Or take the mode of the mixture of Gaussians instead of the mean to handle multimodality
    - We can reduce the variance of the advantage function by introducing the n-step returns (as above between 1-step bootstrapped methods and monte carlo)
    - ***Main point:*** We now have a loss function we can optimize with regards to the advantage function (we can minimize it by making it negative)
        - $\mathcal{L}(\phi)$: $-Adv(s^t,a^t)\log\pi(a^t \mid s^t; \phi)$ to update actors parameters
        - The loss of the critic network can also be calculated from part of this by
            - $\mathcal{L}(\theta)$: $(y^t - V(s^t; \theta))^2$ to update critic parameters
            - Where y^t is the bootstrapped target $y^t = r^t + \gamma V(s^{t+1}; \theta))$
- An additional element of this algorithm is the addition of the cross-entropy term which is a measure of the policy's uncertainty
    - If all actions have a uniform distribution, then this is maximal uncertainty
    - Maximizing the entropy provides a regularization term that discourages premature convergence to a suboptimal close-to-deterministic policy and hence incentivizes exploration

PPO: Proximal Policy Optimization

- The problem with A2C is any gradient step (even with small learning rates) can lead to significant changes of the policy and could reduce the expected performance of the policy
    - This can be mitigated by creating "trust regions" which create regions for the actor's parameters that will not lead a significant reduction in performance
- PPO builds on the idea of TRPO (as seen below) by having trust regions for policy optimization introduced by sampling weight ρ which is a surrogate loss function
- These sampling weights are a measure of how good the current policy is compared to that of the old policy i.e. the expected returns of the current policy in the numerator and the expected returns of the old policy in the denominator
    - This helps to show how much better/worse the new policy is and updates accordingly
    - $P(s,a) = \pi(a|s;\phi) / \pi_{\beta}(a|s)$
        - $\pi(a|s;\phi)$ is the policy that we wish to optimize parameterize by ϕ
        - $\pi_{\beta}(a|s)$ is the behavior policy which was used to select action a in state s (or the "old" policy)
    - This method allows for the policy to be updated multiple times using the same data
- In code the loss function for the policy can be seen as:
    - pg_loss1 = -mb_advantages * ratio
    - pg_loss2 = -mb_advantages * torch.clamp(ratio, 1 - args.clip_coef, 1 + args.clip_coef)
    - pg_loss = torch.max(pg_loss1, pg_loss2).mean()

TRPO: Trust region policy optimization

- Constrains each optimization step of the policy to a certain region to reduce the risk of any degradation in quality of the policy and therefore create gradual and safe quality of steps to improve the policy
- The downfall of TRPO is that is requires solving one of two problems
    - The constraint that is implemented onto the policy
    - Computing a penalty term

Multi-agent RL

- The main question of MARL is: what information can agents use to make their action selection?
    - Or what information can agents use to condition their policy on?
    - Decentralized policies say that agents can only learn from their own local history
    - Centralized policies say that agents can learn from other agents' local history as well

Centralized Training and Execution

- Shared information between agents can include local observation histories, learned world and agent models, value functions or even other agent policies
- This approach means that we reduce the multi-agent problem to a single agent problem where one policy is learning actions to tell each agent which action to take
- The benefit of using a centralized policy is during partial observation environments or complex coordination between agents where now the policy can "fill in" some of the blanks in the environment
- The downfall of centralized learning is:
    - Scalarizing joint rewards across all agents which might be near impossible for zero-sum games
    - The joint policy must learn over the joint action space which grows exponentially with each agent that is added
    - Communication between agents is not possible in all cases

(CTDE) Centralized training with decentralized execution

- Use centralized training to train agent policies, but execute in a decentralized manner
    - Utilizes the local information of all agents to update the agent policies
    - Each agent's policy only requires the agent's local observation to select actions
    - i.e. the policy of each agent is decentralize and they learn from their local history only, but the critic is centralized and is conditioned on the joint observation history and therefore provides better value estimation to their individual policies

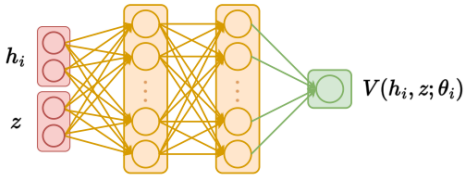
Multi-Agent Policy Gradient Theorem

- Extending the single agent policy gradient theorem to a multi-agent setting is a matter of considering that the expected returns of each agent is dependent on the policies of all the other agents
    - Meaning that the actual rewards received over time is a function of the other agent's and their actions taken in the environment (since the actions come from the policy then directly the returns are dependent on the policies of the other agents)
    - The policy gradients can be represented as follows:
        - $\nabla_{\phi_i} J(\phi_i) = E[Q_i^{\pi}(\hat{h}, \langle a_i, a_{-i} \rangle) \nabla_{\phi_i} \log \pi_i(a_i | h_i = \sigma_i(\hat{h}); \phi_i)]$

- The gradient in this case is not a new equation, but the authors are being explicit that when this is implemented into code this is what happening during the optimization step
    - In our case, we will have the loss function and still optimize the loss function by minimizing the loss, but from a theoretical standpoint this is the notation

Centralized Critics

- Conditioning the policy on the local history of the agent only ensures decentralized execution since there is no information coming from outside the agent $\pi(h_i^t;\phi)$
- Conditioning the critic on all the information from each agent is generally an okay idea since after the training is done, the critic is discarded altogether and only the policy is needed to make decentralized actions
    - The value function conditioned on the observation history of all agents given its current neural network parameters θ_i : $V(h_1^t, \dots, h_n^t; \theta_i)$
- We can add a vectorized parameters (z) in which we can pass any sort of inaccessible information such as the full state of the environment and any external data

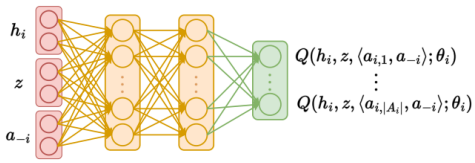


- I think in this case we could even use image data from each of the agents in which we could incorporate something like SuperGlue here as z
    - The loss of this new centralized critic becomes
        - $\mathcal{L}(\theta_i) = (y_i - V(h_i^t, z^t; \theta_i))^2$ where $y_i = r_i^t + \gamma V(h_i^t, z^t; \theta_i)$
    - However, doesn't this mean that by using the advantage function that inherently the policy is being influenced by outside observations?
        - Yes, this is correct, but this is only during the training phase and once training is over the agent only has its local history to influence its actions when the critic is no longer used
        - So, inherently the policy and the critic network are being influenced by the other agent's histories during the training phase
- Centralized critics conditioned on the history of the agent and external information can stabilize training particularly in partially observable multi-agent environments

Centralized Action-Value Critics

- The value function of each agent is conditioned on the observations and actions of each individual agent
- Policy: conditioned only on the local observations of each agent
    - Though the advantage function still influences the policy for simplicity we will say the policy is only conditioned on its own history
    - $\mathcal{L}(\phi_i)$: $- Q(h_i^t, z^t, a^t; \theta_i)\log\pi_i(a_i^t \mid h_i^t; \phi_i)$ for each agent i
- Critic: conditioned on its own local history, the external information, and the next actions taken by all agents

- $\mathcal{L}(\theta)$: $(y^t - Q(h_i^t, \mathcal{Z}^t, a^t; \theta_i))^2$ where $y_i = r_i^t + \gamma Q(h_i^{t+1}, \mathcal{Z}^{t+1} a^{t+1}; \theta_i)$
  - h_i^t its own local history
  - $\mathcal{Z}^t$ external information
  - a^t all other agents' actions beside agent i
  - θ_i neural network parameters for the critic to estimate the value function



- Instead of calculating values of all joint actions across all agents, this method will use the actions of other agents as input and output a scalar value based on these as inputs
  - This is done to drastically reduce dimensionality whereas evaluating all the joint actions would lead to an exponential increase in dimensions

Counterfactual action-value estimation

- In the above we used a state-action value estimate to see what the direct impact of an action would be that helped ***fix the multi-agent credit assignment problem***
  - We did not use a state value estimation, but this can still be used as a part of the advantage function estimation
- The *difference rewards* between the received reward and the reward agent i would have received if it had chosen a different action $\tilde{a}_i$ can aid in the multi-agent credit assignment problem
  - $d_i = R_i(s,\langle a_i,a_{-i}\rangle) - R_i(s,\langle \tilde{a}_i,a_{-i}\rangle)$
  - However, since this requires a "default action" there isn't a best way to compute this from the reward function itself
  - We can, however, use the expectation of the agent's current policy as the default action and use the *aristocrat utility:* $d_i = R_i(s,\langle a_i,a_{-i}\rangle) - E[R_i(s,\langle a_i',a_{-i}\rangle)]$

Individual-Global-Max (IGM) Property

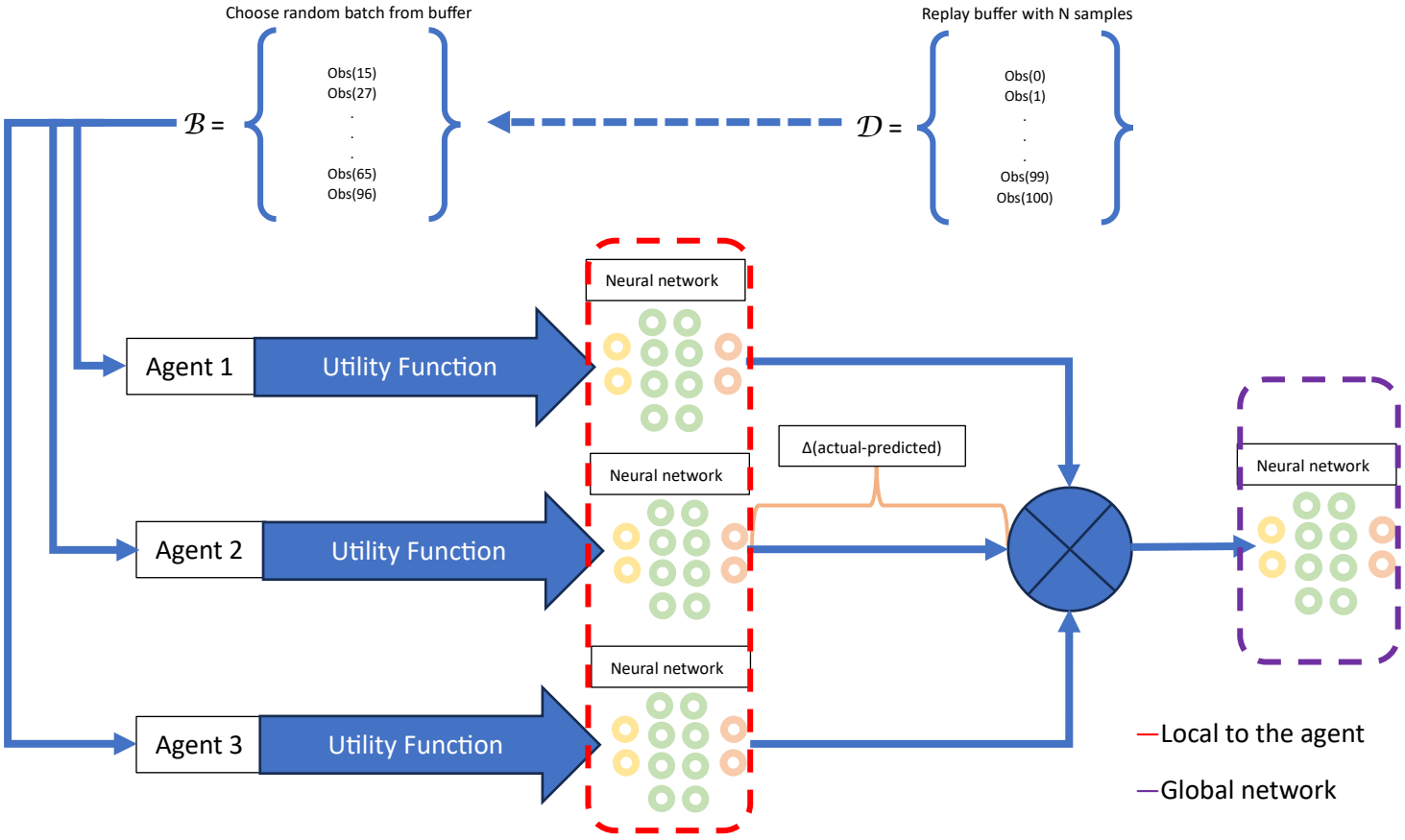
- The process of breaking down the overall global reward system into an agent's individual contribution (its utility function) to the global reward system
  - Two major implications
    - Each agent can follow its own greedy policy with respect to its own utility function to allow for decentralized execution where each agent selects the *greedy joint action*
    - The target value can be obtained by computing the greedy individual actions of all agents with respect to their individual utility function

Linear Value Decomposition (not practical)

- The simplest method to decomposition rewards is using a linear decomposition where the global reward is the sum of each agent
  - $r^t = \bar{r}_1^t + \ldots + \bar{r}_n^t$ where each $\bar{r}_i^t$ is the utility of each agent at timestep t
    - Each utility is obtained by the decomposition and is therefore does not represent a true reward from the environment

Value Decomposition Networks (VDN)

- Maintains a replay buffer $\mathcal{D}$ containing the experience of all agents and joint optimizes the loss function
- $\mathcal{L}(\theta)$: $\frac{1}{B}$ $\text{sum}_{\in B}$ $(r^t + \gamma \max_{\in A} Q(h^{t+1}, a, \bar{\theta}) - Q(h^t, a^t; \theta))^2$
    - Where $\gamma \max_{\in A} Q(h^{t+1}, a, \bar{\theta})$ is the actual return of an action for each agent
    - And $Q(h^t, a^t; \theta)$ is the predicted value of each agent's action
    - The point: the value function is learning the decomposed value function (which is a global value function) of all the agents by using the sum of their respective actions in an overall target / value estimation



Monotonic Value Decomposition (too restrictive)

- The name QMIX represents an agent's network and the overall mixing network that combines each agent's network into Q_{tot}

- QMIX is a centralized action-value function that is approximated by a monotonic aggregation computed by the mixing network which is parameterized by the output of the trained hypernetwork
    - $Q(h, z, a, \theta) = f_{mix}(Q(h_1, a_1; \theta),\ldots,Q(h_n, a_n; \theta_n);\theta_{mix})$
        - Multiple the action-value functions together to get an overall action-value function
        - This ensures the monotonicity property

QTRAN

- Uses three neural networks to aid in factorizing tasks free from additivity and monotonicity as with VDN and QMIX
    - 1. Network to learn the joint action-value network: $Q(h, z, a; \theta^q)$
    - 2. Utility functions for each agent: $Q(h_i, a_i; \theta_i)$
    - 3. Network to learn the state-value network: $V(h, z; \theta^v)$
    - $\Sigma_{i\in I}\, Q(h_i, a_i; \theta_i) - Q(h, z, a; \theta^q) + V(h, z; \theta^v)$
        - $V(h, z; \theta^v) = \max_{a\in A} Q(h, z, a; \theta^q) - \Sigma_{i\in I}\, Q(h_i, a_i^*; \theta_i)$

Joint action learning with agent modeling

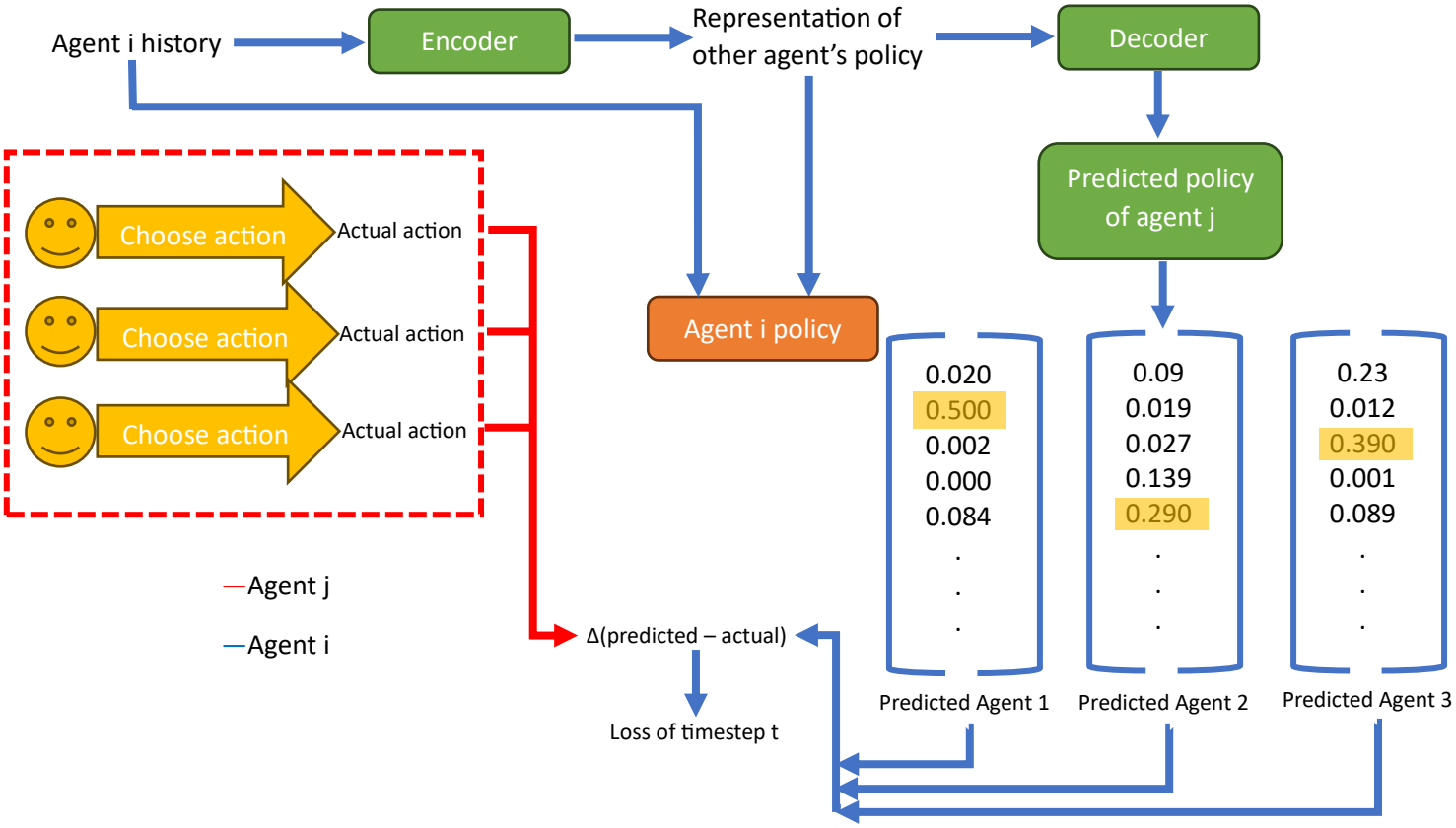
- Each agent's policy, ideally, is conditioned on the policies and value functions of all other agents
    - This ideal is not practical as policies can be much too complex, policies might be available during centralized training but not in decentralize execution, and the policies of other agents change during training
- To overcome these barriers of learning representations of other agent's policies we can introduce encoder-decoder neural network architectures

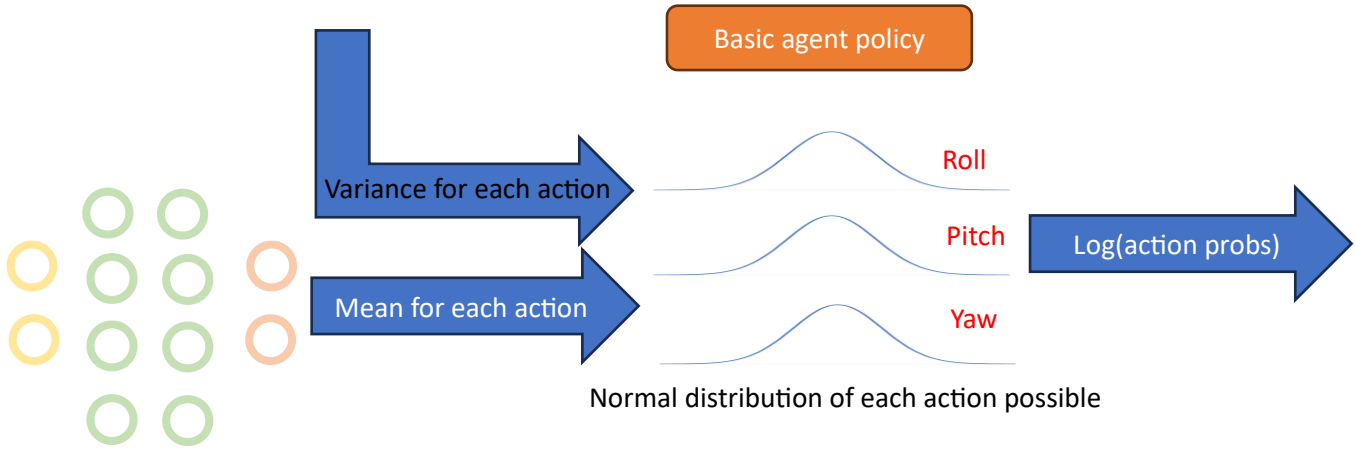
Encoder-decoder method

- Encoder-decoder neural networks are often used as sequence-to-sequence architecture
    - In our case, the encoder will take the observation history and output a representation of the policies for each other agent
        - $m_i^t = f^e(h_i^t; \psi_i^e)$ where f^e is the encoder network
    - The decoder network will take this representation of a policy and predicts the action probabilities of all other agents at the current timestep
        - $\hat{\pi}_{\neg i}^{\,i,t} = f^d(m_i^t; \psi_i^d)$ where f^d is the decoder network
- Every neural network needs a loss function to optimize and therefore the cross-entropy loss is defined by:
    - $\mathcal{L}(\psi_i^e, \psi_i^d) = \Sigma\, -\log\hat{\pi}_j^{\,i,t}(a_j^t)$ with $\hat{\pi}_j^{\,i,t} = f^d(f^e(h_i^t; \psi_i^e); \psi_i^d)$
- When applied to the level-based foraging environment the centralized A2C + AM considerably outperformed centralized A2C which illudes to the idea that an agent learning the other agent's policies leads to lower variance and quicker convergence to higher returns
- The next question is: what do each of the agents think about a particular action that agent i did and what do they think about the knowledge of agent i?
    - This is known as recursive reasoning which is implemented using Bayesian variational inference to model the beliefs of agents about the policies of other agents as well as the uncertainty of these beliefs
- Agnostic to the MARL algorithm used to learn the policies of agents

Encoder-decoder flow path

- Agent i has an encoder to represent agent j's policy
- The decoder returns the predicted policy's probability actions
- Agent j takes an action (actual action)
- This "actual action" is compared to the model prediction probability of this "actual action"
- The loss is the difference between the actual value (100% since it's a label) and the model's predicted probability value

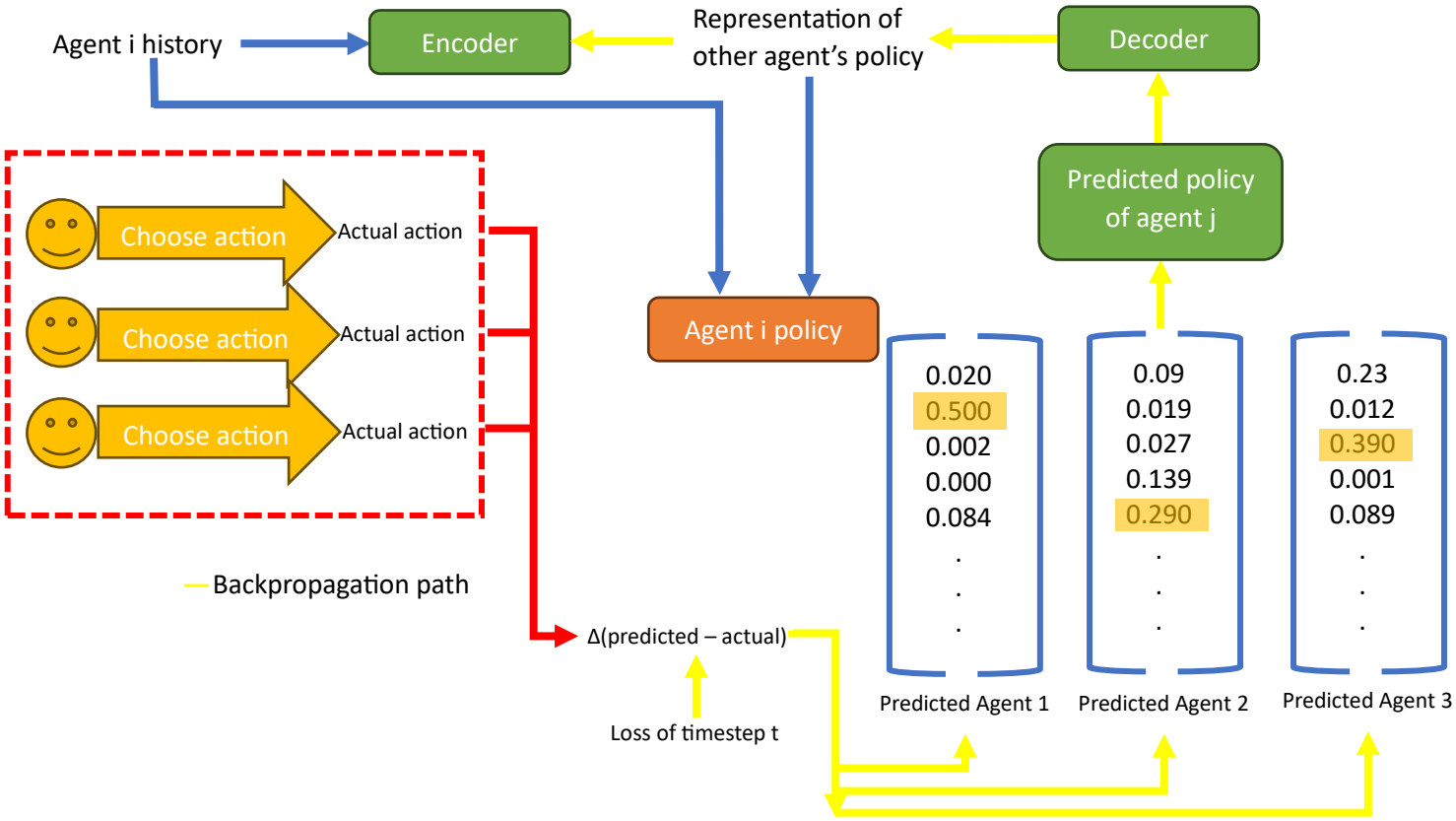




- The encoder representation can be used to condition the policy and the value function of each agent

Gradient flow path

- The backpropagation step will include updating the gradients in the decoder and encoder networks
    - This will change the policy of Agent i since the gradients are being updated in the encoder network in which the output is given to the agent's policy as external information along with the history of agent i



Environments with homogeneous agents

- Agents share the same or similar capabilities and characteristics which can include history, actions, observations, and rewards
    - Nearly identical agents
- Weakly homogenous agents vs strongly homogenous agents
    - Weakly homogenous agents can execute different actions compared to the optimal joint policy in a specific state between agents
    - Strongly homogenous agents execute the same actions under a joint policy that is their same local policy in a specific state

Parameter Sharing

- The policy network's parameters are the same for all agents which gives strongly homogenous agents
    - Observations, actions and rewards will update the homogenous network parameters
    - Being that the overall network parameters are learning from each agent's experiences the policy is learning on a larger, and more diverse, set of trajectories compared to an agent learning from only its own experience
- Sharing an agent's index might be beneficial in agents learning their own distinct behaviors, but papers show that this is not the case
    - They instead show that using Selective Parameter Sharing (SePS) which identifies agents that may benefit from sharing parameters by partitioning them based on their abilities and goals
        - Code here

Experience Sharing

- Sharing the experienced trajectories among agents and train a different set of parameters per agent
    - This relaxes the strongly homogenous agents and allows for agents to learn different policies
    - In IDQN the replay buffer $\mathcal{D}$ can now store shared trajectories of all agents where an agent can pull a trajectory from a whole slue of experiences from other agents as well (this can only work if the agents are weakly homogenous)
- To include this in the continuous space we can use importance sampling to correct for transitions collected by other algorithms and apply the idea of experience sharing to on-policy settings
    - This is done by allowing an agent to learn from its own sample trajectory, but also from the trajectories of the other agents given that this is looked at as off-policy (this is in reference to the trajectories are from different policies other than the current agent's policy that is being optimized)
- The benefit of experience sharing over parameter sharing is that each agent learns uniformly as opposed to agents learning on their own
    - This, in turn, allows agent's actions to be in coordination resulting in sample efficiency

Policy Self-Play in Zero-Sum Games

- Characterized by three primary attributes:
    - Sparse reward system: the game / event terminates after a finite number of moves at which point the victor receives some sort of sparse reward like +1, the losing player receives -1, or both players receive 0 for a draw
    - Large action space: agents can typically choose from many actions which leads to a large branching in the search space
        - Examples of this is the many different moves on a chess board or go board
    - Long horizon: reaching a terminal state requires a large amount of timesteps
        - The agents may have to explore long sequences of actions before they receive any reward
- This can be thought of as a tree in which each node is a state of the game
    - Each outgoing edge can be thought of as a *possible* action choice for whose turn it is
        - The tree can either be very wide where there are many actions leading to a large branching factor

        - The tree can also be very deep where this indicates to get to a terminal state requires many actions to get to a terminal state
    - Instead of exploring each tree in full, one can involve heuristic search algorithms such as alpha-beta minimax search

Population-Based Training

- Agent's policies are trained against itself *and* past versions of its policy
    - This trains the policy on a more diverse *population* of policies which can aid in a reduction of overfitting the policy to itself
        - Population in this context is used to insinuate that, over the training epochs, the number of policies can increase
        - This population of policies may change and adapt in each generation to perform better
- Steps for population-based training:
    - Initialize population: policy Π_i^1 for each agent i
        - Each population may start with a single uniform policy that selects actions uniform-randomly
    - Evaluate populations: in generation k, evaluate the current policies $\pi_i \in \Pi_i^k$ in each agent's population to measure the performance of each policy based on expected returns
    - Modify populations: change the current policy and/or add new policies to the populations based on the evaluations of policies $\pi_i \in \Pi_i^k$
- Each generation going forward will have a new set of policies that the agent will learn on
    - Inadvertently, each agent will be learning of the other agent's experiences within their environment by observing their actions
        - This leads to the non-stationarity problem and does not seem viable for our work

MARL in Practice

-

Decentralized Training and Execution

- Agents only have access to their own local history, models, and observations

Independent learning

- Each agent does not model the presence and actions of other agents instead the other agents are seen as a non-stationary part of the environment dynamics
- Each agent trains its policy in a completely local way using single RL methods
    - Benefit of scalability as there is no exponential growth of having many joint actions
    - Downsides:
        - Policies do not leverage information from other agents
        - Training can be affected by non-stationarity caused by concurrent training of agents
        - Agents cannot distinguish the difference between stochastic changes in the environments because of other agent's actions and the environment transition function

Independent Value-Based Learning (IDQN)

- Each agent trains its own action-value function $Q(\cdot; \theta_i)$

- Maintains a replay buffer $\mathcal{D}_i$ (as a remember point – this is so that the moving target problem is mitigated by keeping a buffer of state, action, reward, and next state information)
- Learns from its own observation history, actions and rewards using DQN
- The loss function is still $\mathcal{L}(\theta)$: $(y^t - V(s^t; \theta))^2$, but y_t in this case is from the agent's own target network $y^t = r_i^t + \gamma \max Q(h_i^{t+1}, a_i; \bar{\theta}_i)$
    - The critic network is now $Q(h_i^t, a_i^t; \theta_i)$ in which it is only updated on its own history and actions
    - The difference in $\bar{\theta}_i$ and θ_i still hold that the former is for the target neural network and the latter is for the value function neural network
    - The value function parameters θ_i are minimized by aggregating the loss of each agent where $\mathcal{L}(\theta_1) + \mathcal{L}(\theta_2) + \ldots + \mathcal{L}(\theta_N)$
- The downside of using IDQN is the replay buffer could potentially provide a mismatch between observations, actions and rewards
    - Where two agents are in the same state and select the same action, but since their target networks are different, they get different returns for the same state/action

Independent Policy Gradient Methods

Independent REINFORCE

- The monte carlo version
- Policy gradient methods can be introduced into MARL interpedently per agent
    - Each agent maintains its own policy and learns independently from its own experiences
    - The policy gradient is computed based on the agent's own actions and rewards not considering other actions or policies taken by other agents
- Each agent's policy gradient is updated by the expectation of its policy multiplied by the agent's returns with the gradients being normalized by the inverse current probability of selecting the action under the current policy
    - One benefit of this is the agents individually learn from the most up to date information

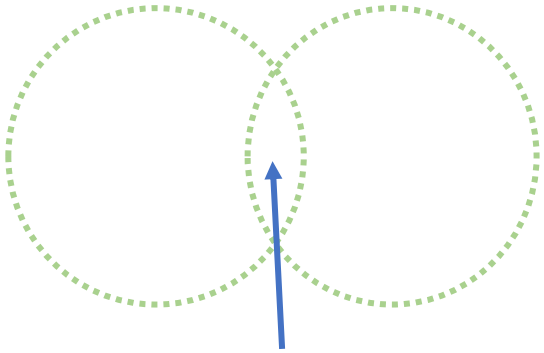
Independent A2C / PPO

- Implementing this is similar to REINFORCE since each agent is independent, they will use their own experiences to update their own policy
- In a parallelized environment, each agent receives experiences from multiple parallel environments
    - This then gives the problem that the collected experience across all agents and parallel environments form batches of higher dimensionality
    - The observations collected from k environments on a time step t from a two-dimensional matrix where the observation is a function of the timestep and the environment

$$\begin{matrix} o1(t,1) & \ldots & o1(t,K) \\ \ldots & \ldots & \ldots \\ o\eta(t,1) & \ldots & o\eta(t,K) \end{matrix}$$

- The loss in this case is summed over each environment for each agent and divided by the number of environments to get the average loss per environment
    - Policy loss: $\mathcal{L}(\phi) = 1/K * \sum_{i \in I} \sum_{k=1}^{K} L(\phi i \mid k)$
        - Where the $L(\phi i \mid k)$ = advantage * $\log_\pi$(action | agent history; ϕ_i)

        - Sum of each environment's loss
        - Sum of each agent's loss over the environments
        - Divided by the total environments to get total loss per agent per environment
    - Critic loss: $\mathcal{L}(\theta|k)$: $(y_i - V(h_i^{t,k}; \theta_i))^2$ where $y_i = r_i^{t,k} + \gamma V(h_i^{t+1,k}; \bar{\theta}_i)$
        - Same format as before, but for each environment
        - Clarification point: given the loss function notation of being a conditional probability $\mathcal{L}(\theta|k)$ I would like to know what is the intersection point between the policy parameters and the k^{th} environment since $\mathcal{L}(\theta|k) = P(\theta \cap k) / P(k)$



        - This point, intuitively, seems to be the intersection of the critic parameters such that they minimize the loss across all of the K environments – i.e. we are taking the loss of all the environments and dividing it by the number of environments to get a loss per environment and then minimizing this
            - So, this seems to be fitting the parameters to the general environment and not just picking the best performing parameters across the environments

Some terminology

Correlated equilibrium

- allow correlation between policies
    - Each agent knows its own action, but NOT the other's action
    - Each agent knows the probability distribution
        - The agent cannot deviate to increase their own expected reward

Nash equilibrium

- applies the idea of mutual best responses to general-sum games with two or more agents
    - Focuses on the fairness aspect of the fairness/welfare ratio (fairness / welfare defined below)
        - If we have an agent i then every other agent in the game is focusing on minimizing the maximized achievable reward by agent i
    - Good to use when agents have a mixture of competitive and cooperative environments
    - In systems where stability is desired Nash Equilibrium provides a point where the system can settle
    - No agent benefits from changing their strategy if others keep theirs unchanged

ε-Nash equilibrium

- relaxes the idea that an agent is not allowed to gain *anything* by unilaterally deviating from its policy
    - Instead, this constraint is relaxed to say the expected return cannot increase by some value determined by the user $\varepsilon > 0$ when deviating from its policy in the equilibrium

Pareto-Optimality

- there is not better joint-policy that improves reward for at least one agent without decreasing the reward of another
    - Every game must have at least 1 Pareto-optimal solution
- Temporal difference learning: iterative way to update the value function
    - i.e. predict the cumulative reward of a state, step into the state and observe the actual reward, and then update the prediction based on how far it was off from the actual

Welfare-optimal

- a policy that is focused on the maximum expected return possible among all agents

Fairness-optimal

- a policy that is focused on each agent getting close to similar rewards
    - Ensure magnanimity between agents

Regret

- difference between reward received and the reward an agent could have received given a different policy
    - No-regret if limit is almost zero meaning that the agent acted in an optimal way
    - Unconditional regret is calculated without assuming any specific condition or state of the world
    - Conditional regret assumes that a state or some condition is fixed in the environment

Zero-sum game

- a game where an agent gets a positive reward for completing a task and the other agent receives an opposite negative reward to sum to zero
    - i.e. if an agent scores a goal they get +1 where the opponent agent gets -1

Stochastic

- basic definition is something that happens randomly and is not fixed

"Conditioned on the state"

- means that the agent's actions, learning, and decision-making processes are all influenced and determined by the current state of the environment in which it operates

Value of information

- evaluates how the outcomes of an action may influence the learning agent's beliefs about other agents, and how the changed beliefs will in turn influence the future actions of the learning agents

Partially observable environment

- Agent's are given incomplete information of the environment – this means maybe they are lacking the actions of other agents or they simply are not able to sense their whole environment (sensors on the back of a drone sensing what is behind it might not be there)

Difference rewards

- Approximates the difference between the received reward and the reward agent i would have received if it had chosen a different action $\tilde{a}_i$
    - $d_i = R_i(s,\langle a_i,a_{-i}\rangle) - R_i(s,\langle \tilde{a}_i,a_{-i}\rangle)$

Aristocrat utility

- Approximates the difference between the received reward and the reward agent i would have received if it had chosen the average action from its policy
    - $d_i = R_i(s,\langle a_i,a_{-i}\rangle) - E[R_i(s,\langle a_i', _i,a_{-i}\rangle)]$

Monotonic

- The function is either non-decreasing on its domain or non-increasing on its domain
    - Meaning that if it is non-decreasing it is either increasing or constant at every point on its domain
    - If it is non-increasing it is either decreasing or constant at every point on its domain

Alpha-beta minimax search

- A search algorithm that seeks to decrease the number of nodes that are evaluated by the minimax algorithm in its search tree
    - i.e. the algorithm will eliminate trees that are equivalent to a previous tree or of lesser value than that of previous trees since neither will be able to influence the final result

Neural network

Input layer

- is referred to the first layer that is multiplying the weights and the inputs

Output layer

- is referred to the last layer which gives the output of the network

Hidden layer

- is referred to the layers within the neural network which are internal representations

Depth

- is the number of layers

Width / hidden dimension

- is the number of neurons in a layer

Universal approximation theorem

- states that a neural network with a layer of 1 can represent any continuous function

Closed-form expression

- A function that can fully express a given problem
    - i.e., $a^2 + b^2 = c^2$ for the Pythagorean theorem

Non-closed form expression (open-form)

- An expression that cannot be explained by basic math operations and is instead a complex set of advanced operations such as differential equations or infinite integrals

Vanilla Gradient descent

- A method used for optimization that involves updating parameters in a way that is in the negative direction of the loss function; performs on all training data
    - In layman terms this simply tries to find a minimum of some function

Stochastic gradient descent

- Chooses random samples from the data and performs gradient descent on those specific points of data

Mini-batch gradient descent

- Take a batch of the training data and performs gradient descent on this batch
    - Between the extremes of vanilla gradient descent and stochastic gradient descent

Backward pass

- The process of computing gradients for every gradient backward through the network one time

Forward pass

- The process passing outputs forward in the network to get a final output given a set of inputs

Deep RL

Deadly triad

- Combining off-policy learning, bootstrapped targets, and value function estimations can cause the bootstrapped target to estimate to change very rapidly which can lead to unstable optimization

Off-policy learning

- The value function that is learned by sampling actions that can be random without taking them from the policy

On-policy learning

- The value function is learned from the actions that are taken by the policy given a state

Identically and independently distributed (I.I.D)

- Assumes training data is not correlated in any way and that it is pulled from an equal distribution

Surrogate loss function

- A loss function that updates the policy parameters

Returns

- Returns is the cumulative reward received over time (i.e. the actual cumulative reward received)

Value

- Value is the expected cumulative reward (i.e. the prediction by the critic)

Utility function

- A function that is conditioned on each agent's own observation history and action of the agent
    - $Q(h_i, a_i; \theta)$ is the utility function – all utility functions of each agent is jointly optimized to approximate the centralized action-value function

Recursive reasoning

- One who considers how their actions might affect others and what others might believe about their own knowledge

Gamma-contraction

- A property refers to how the application of the Bellman operator reduces the distance between arbitrary value functions, drawing them closer to the optimal value function with each iteration