# Policy RL 1

Let's go back a bit to our value function approximation problem.
How to efficiently estimate value functions for continuous states? Remember our value approximations assume discrete spaces and iterate until convergence. There is no way to converge if the state space is continuous.
**Highly discretized state spaces**: increase the "definition" of discretization. Even after this, value estimation algorithm should visit all states$\rightarrow$ curse of dimensionality. A better alternative is to coarse coding.
RBFs
**Coarse Coding**
A method to represent continuous spaces by using a set of overlapping receptive fields (or features). Each feature corresponds to a region in the continuous space and multiple features can overlap, leading to a redundant and distributed representation. This redundancy allows for better generalization and robustness in representing continuous

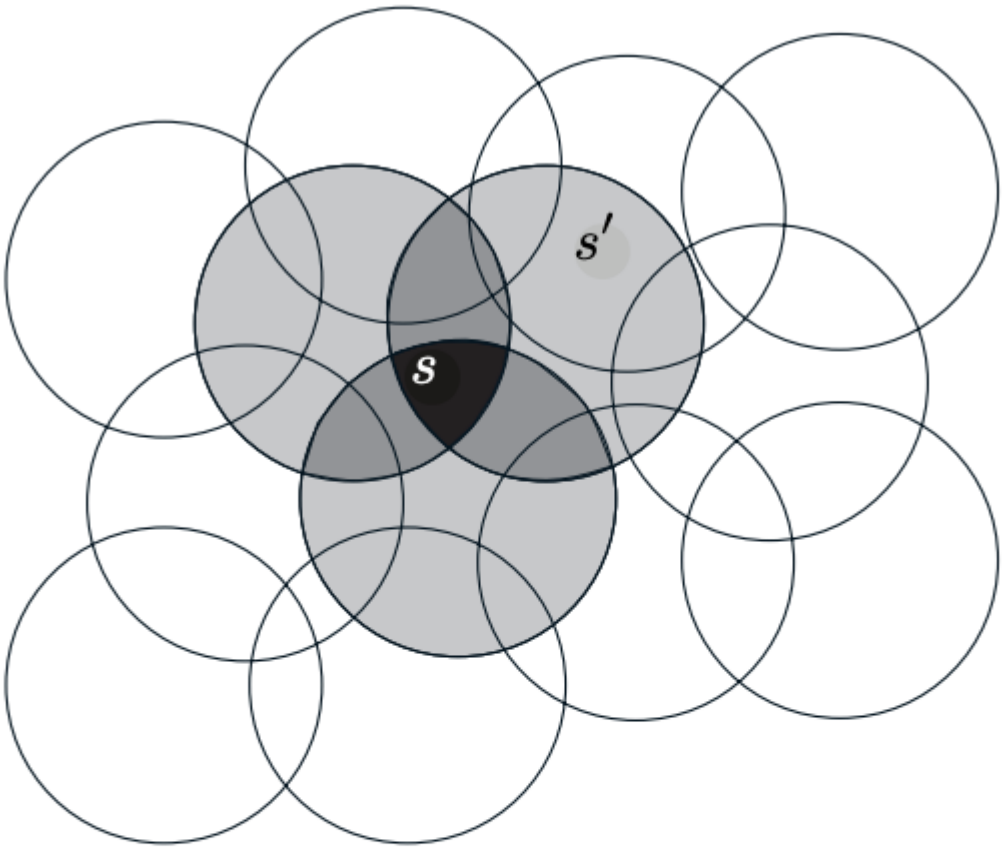
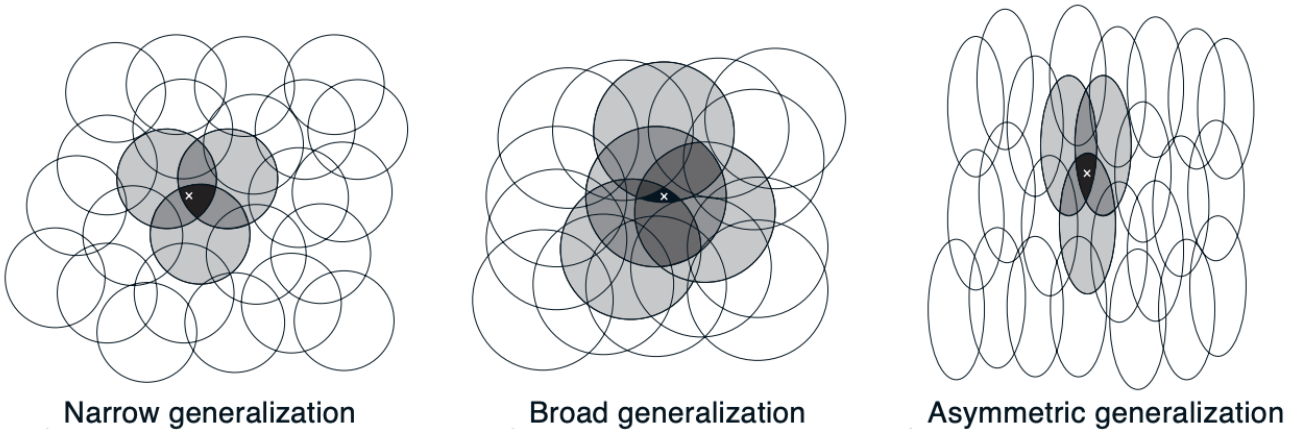



**Figure 9.6:** Coarse coding. Generalization from state $s$ to state $s'$ depends on the number of their features whose receptive fields (in this case, circles) overlap. These states have one feature in common, so there will be slight generalization between them.

variables.

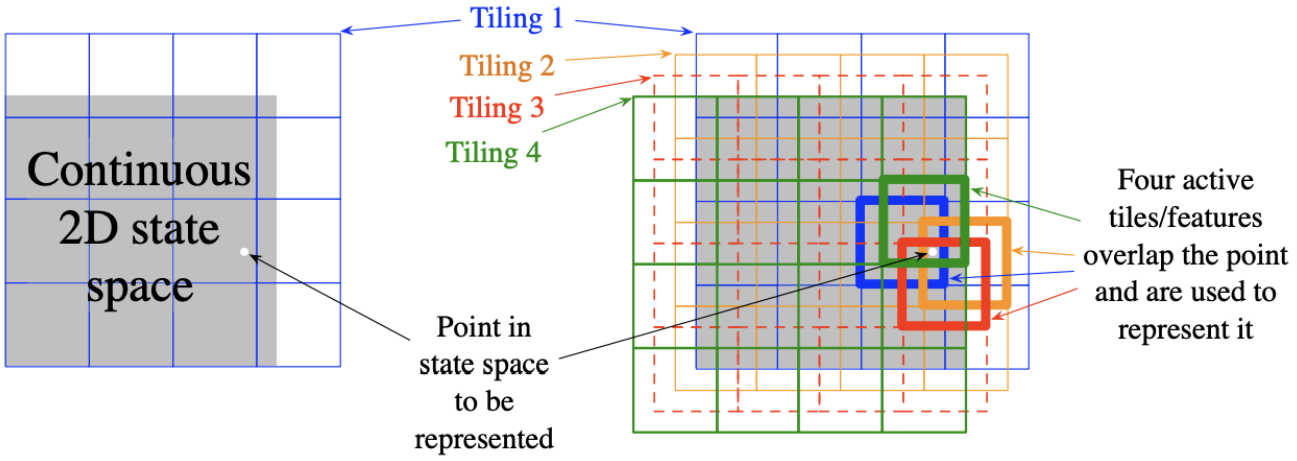
Generalization in linear function approximation methods is determined by the sizes and shapes of features' receptive fields.



**Tile Coding**

Tile coding is a specific implementation of coarse coding for multidimensional spaces. The tiles in different tilings are offset from each other.

The receptive fields here are squares rather than the circles in Coarse Coding.




It has several advantages:

- trivial to compute due to binary feature vectors
- generalization is better due to the states ending up in between any of the same tiles.
  In general, asymmetric offset tilings are better for generalization
  To encourage generalization among a particular dimension, create more finer coding in that dimension and more coarse coding in other dimensions (asymmetric)

Radial Basis Functions (RBFs) are natural extension of coarse coding into continuous features. Instead of having $0$ and $1$ feature vectors, now we have values that are in an interval $[0, 1]$

# Stochastic Policies

We prefer stochastic policies over deterministic ones due to the following reasons:

1. Good for exploration-exploitation (rock-paper-scissors and aliased gridworld). Deterministic policies might get stuck in such equally probable value states.
2. Handling Partial Observability: Stochastic policies introduce randomness in action selection, making the agent's behavior more robust under certainty. **Example:** in a maze with limited sensory input, a stochastic policy can help the agent explore different paths, increasing the likelihood of finding the goal.

# Why learn Policy?

There are couple of advantages of learning the policy directly:

1. We are able to deal with continuous actions
2. We are changing the policy **smoothly**, meaning after an update, we only slightly change the probability distribution over actions. In case of $\epsilon$-greedy action selection on $Q$-values, the policy heavily changes when the best action becomes another one.

The objective of a policy is always the same, namely optimize the expected return of its start state $s$:

$$J(\theta) = v_{\pi_\theta}(s_0) = E[\sum_{t=0}^{T-1} r_{t+1}]$$

Note that we assume here $\gamma = 1$ but it can also be discounted. This helps to derive the gradient in simpler steps.
We want to do gradient ascent on $J(\theta)$ :

$$\theta_{n+1} = \theta_n + \alpha_n \Delta G_{\theta_n}$$

**Finite Difference Gradient Estimator**:
This is the simplest update is this one. We estimate the gradients by a small parameter change in $\epsilon$-range:

$$\Delta J(\theta) \approx \frac{(J + \epsilon) - (J - \epsilon)}{2\epsilon}$$

However, this means that for $n$ parameters, we would need at least $2n$ rollouts for an estimate. For stochastic policies, this estimate is extremely noisy and it is hard to distinguish the difference between $R^+$ and $R^-$.
**Basic Random Search**

1. Pick a policy $\pi_\theta$, perturb the parameters $\theta$ by applying $+v\delta$ and $-v\delta$ where $v < 1$ is constant noise and $\delta$ is random number from normal distribution
2. Run the policies and apply actions based on $\pi(\theta + v\delta)$ and $\pi(\theta - v\delta)$ and collect the rewards $r(\theta + v\delta)$ and $r(\theta - v\delta)$
3. For all $\delta$, compute the average $\Delta = \frac{1}{N}\sum[r(\theta + v\delta) - r(\theta - v\delta)]\delta$ and update the parameters $\theta$ using $\Delta$ and a learning rate $\alpha$:

$$\theta_{j+1} = \theta_j + \frac{\alpha}{N}\sum_{k=1}^{N}[r(\pi_{j,k,+} - r(\pi_{j,k,-})\delta_k$$

**Augmented Random Search**
Improve BRS by dividing the reward by their standard deviation and normalizing the states. Finally, by using, the top $k$ best rollouts to compute the average.

Overall, these methods are simple to understand and requires less hyperparameter tuning. However, they tend to favor "lucky" rollouts and it is hard to distinguish if good performance is **due to parameter variation or environment noise**. Lastly, they are less sample efficient.

# Policy Gradients

The goal is to do gradient ascent to maximize performance measure $J(\theta)$ :

$$\theta_{t+1} = \theta_t + \alpha \Delta J(\theta_t)$$

where $\Delta J(\theta_t)$ is an N-dimensional gradient.

$J(\theta)$ is the expected reward:
Denote $\tau$ as a state-action sequence $s_0, a_0, \dots s_{T-1}, a_{T-1}, s_T$. The return of a trajectory is given by $R(\tau) = \sum_{t=0}^{H} R(s_t, a_t)$ and our goal is to find $\theta$ :

$$max_\theta J(\theta) = max_\theta \sum_\tau P(\tau;\theta) R(\tau)$$

Dynamics model vanishes because it doesn't contain $\theta$ in it:
In the end, we obtain the following gradient:

$$\Delta_\theta E_{\tau\sim\pi_\theta} G(\tau) = E_{\tau\sim\pi_\theta}(\Delta_\theta log P(\tau|\theta) G(\tau)$$

$$= E_{\tau\sim\pi_\theta}(\sum_{t=0}^{T} \Delta_\theta \log \pi_\theta(a_t|s_t) G(\tau))$$

Increase probability of paths with positive $G$. Decrease probability of paths with negative $G$.
To estimate the gradient, we can sample trajectories and approximate the expectation.
Vanilla PG has high variance due to two factors:

- First, it can be seen as Monte Carlo method of policy-based RL. Hence, MC samples bring high variance compared to TD
- Suppose we are playing CartPole. Our reward is 1 for each time step, until we terminate. This leads to always positive gradients, which can be seen "supporting" the last actions. Only if we sample the other action, we might experience an even higher return which pushes the policy towards the newly explored actions.

The reward-to-go typically has **a shorter time horizon** compared to the full episode return, which **reduces the number of random variables** (rewards) contributing to the return. Fewer random variables mean less cumulative variance.

**REINFORCE: Monte-Carlo Policy-Gradient Control (episodic) for $\pi_*$**

Input: a differentiable policy parameterization $\pi(a|s, \boldsymbol{\theta})$
Algorithm parameter: step size $\alpha > 0$
Initialize policy parameter $\boldsymbol{\theta} \in \mathbb{R}^{d'}$ (e.g., to $\mathbf{0}$)

Loop forever (for each episode):
  Generate an episode $S_0, A_0, R_1, \dots, S_{T-1}, A_{T-1}, R_T$, following $\pi(\cdot|\cdot, \boldsymbol{\theta})$
  Loop for each step of the episode $t = 0, 1, \dots, T-1$:
    $G \leftarrow \sum_{k=t+1}^{T} \gamma^{k-t-1} R_k$ $(G_t)$
    $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \alpha \gamma^t G \nabla \ln \pi(A_t|S_t, \boldsymbol{\theta})$

This second issue can be tackled by the usage of **baseline** which is a constant that doesn't influence the gradients being unbiased:

$$\begin{aligned}\mathbb{E}_\tau\left[(G(\tau)-b)\sum_{t=0}^{T}\nabla_\theta \ln p_\theta(a_t|s_t)\right] &= \mathbb{E}_\tau\left[G(\tau)\sum_{t=0}^{T}\nabla_\theta \ln p_\theta(a_t|s_t)\right] - \mathbb{E}_\tau\left[b\sum_{t=0}^{T}\nabla_\theta \ln p_\theta(a_t|s_t)\right] \\ &= \nabla J(\theta) - b\underbrace{\int p_\theta(\tau)\nabla_\theta \ln p(\tau)d\tau}_{=0} \\ &= \nabla J(\theta)\end{aligned}$$

This result can be proved with Expected Log Probability Lemma, which uses the fact that the integral of probability functions is 1 and the log-trick that is used scoring functions.

**Baseline Choices**

- Constant baseline : Return of the entire trajectory
- Optimal constant baseline
- Time dependent baseline
- State-dependent baseline : the value function

$$b(s_t) = E[r_t + r_{t+1} + \ldots + r_{H-1}] = V^\pi(s_t)$$

Baselines increase/decrease probabilities of the paths that are better/worse than the average.

**Advantage Function**

Advantage function measures how much better/worse taking a particular action $a$ in state $s$ compared to average action in that state:

$$A^\pi(s_t, a_t) = Q^\pi(s_t, a_t) - V^\pi(s_t)$$

In this sense, having information about **good actions in challenging states** is better than information about invariant actions in good states.

REINFORCE with a baseline estimates both advantage function and policy update by keeping two separate sets of weights and learning rates:

**REINFORCE with Baseline (episodic), for estimating $\pi_{\boldsymbol{\theta}} \approx \pi_*$**

Input: a differentiable policy parameterization $\pi(a|s,\boldsymbol{\theta})$
Input: a differentiable state-value function parameterization $\hat{v}(s,\mathbf{w})$
Algorithm parameters: step sizes $\alpha^{\boldsymbol{\theta}} > 0$, $\alpha^{\mathbf{w}} > 0$
Initialize policy parameter $\boldsymbol{\theta} \in \mathbb{R}^{d'}$ and state-value weights $\mathbf{w} \in \mathbb{R}^d$ (e.g., to $\mathbf{0}$)

Loop forever (for each episode):
  Generate an episode $S_0, A_0, R_1, \ldots, S_{T-1}, A_{T-1}, R_T$, following $\pi(\cdot|\cdot,\boldsymbol{\theta})$
  Loop for each step of the episode $t = 0, 1, \ldots, T-1$:
    $G \leftarrow \sum_{k=t+1}^{T} \gamma^{k-t-1} R_k$ $(G_t)$
    $\delta \leftarrow G - \hat{v}(S_t,\mathbf{w})$ ← *advantage*
    $\mathbf{w} \leftarrow \mathbf{w} + \alpha^{\mathbf{w}} \delta \nabla \hat{v}(S_t,\mathbf{w})$
    $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \alpha^{\boldsymbol{\theta}} \gamma^t \delta \nabla \ln \pi(A_t|S_t,\boldsymbol{\theta})$ ← policy gradient update

**Actor-Critics**
Remember in our value-based methods, we previously learn the terms $\sum_{t=1}^{T} R_t$ by the $v/q$ functions, as it is the expected value of those. Hence, we can also plug them in here:

$$E_{\tau\sim\pi_\theta}(\sum_{t=0}^{T} \Delta_\theta \log \pi_\theta(a_t|s_t)G(\tau)) =$$

$$E_{\tau\sim\pi_\theta}(\sum_{t=0}^{T} \Delta_\theta \log \pi_\theta(a_t|s_t)Q^\pi(s_t,a_t))$$

Actor: $\pi$ policy
$Critic$ : $q_w$ value function
Note that we can still subtract a baseline from the policy gradient to obtain advantage function.
Instead of approximating the two functions $Q^\pi(s,a)$ and $V^\pi(s)$ or the advantage function directly, common approaches use the critic to approximate the state-value function $V^\pi(s)$ and estimate the advantage in one of the following ways:

- $A^\pi(s,a) = R(s,a) - V^\pi(s)$: MC advantage estimate, where $R(s,a)$ is the return received in state $s$.
- $A^\pi(s,a) = r + \gamma V^\pi(s') - V^\pi(s)$: TD advantage estimate, or $n-$step TD formulation:
- $A^\pi(s,a) = \sum_{t=0}^{T} (\gamma\lambda)^l \delta_{t+1}$: GAE where $delta_t = r_{t+1} + \gamma V^\pi(s_{t+1} - V^\pi(s_t))$ is the TD error. GAE is the average of all $n-$ step TD advantages weighted by a discount parameter $\lambda$.

**A2C**: advantage actor critic. Estimate Q function by 1-step TD
**Exploration-Exploitation**

- $\epsilon-$ greedy is not applicable here. Wildly different actions cause breakage.
- Use entropy regularizer that encourages exploration while keeping the policy not fully random. Similar to KL regularization in TRPO
  Summary of PGs:
- no replay buffer
- on-policy
- less sample efficient

# Policy RL 2

When updating our policy, we want to make sure that we don't change too much. The reason for that is that our samples come from $\pi_\theta$ and the more we change $\pi_\theta$, the more our gradient estimate becomes inaccurate.

Policy gradients keep old and new policy close in **parameter space**, but not in policy space. However, we are not directly interested in the change of the parameters, but of the **policy distribution**. A better way of doing do is by using KL divergence as difference.

**TRPO**

TRPO establishes an upper bound for the divergence error. It guarantees a policy improvement if a local approximation is optimized with a trusted region.
TRPO minimizes a quadratic equation to approximate the inverse of FIM. To do this for every policy is expensive since it requires a large batch of rollouts to approximate it correctly. Overall, it is less sample efficient than other PG methods when trained with first-order methods such as Adam
TRPO objective:

$$max_\theta E_t[\frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{old}}(a_t|s_t)}A_t]$$

subject to $E_t[D_{KL}[\pi_{\theta_{old}}(*|s_t), \pi_\theta(*|s_t)]] < \delta$

**PPO**

A simpler but empirically similar alternative to TRPO. It has two variants:

- PPO-penalty: TRPO with KL-penalty instead of constraint.
- PPO-clip: no constraints. Just simple clipping to remove incentives to move too far.

Let probability ratio $r_t(\theta)$ :

$$r_t(\theta) = \frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{old}}(a_t|s_t)}$$

Then TRPO objective is given by:

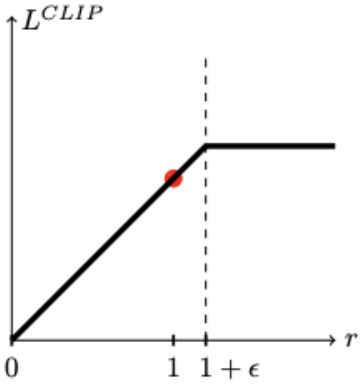
$$L^{TRPO}(\theta) = E_t[r_t(\theta)A_t]$$

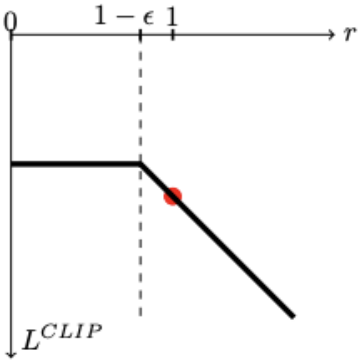
On the other hand, PPO objective is:

$$L^{CLIP}(\theta) = E_t[min(r_t(\theta)A_t, clip(r_t(\theta), 1-\epsilon, 1+\epsilon)A_t)]$$

where $\epsilon$ is hyperparameter $(0.1 - 0.2)$
If $r_t(\theta) > 1$, the action $a_t$ at state $s_t$ is more likely in the current policy than the old policy. We take the minimum of the clipped and non-clipped objective. The clipping operator is a lower bound (pessimistic bound) of the unclipped objective:

$A > 0$




$A < 0$