Ronnie Liu
Updated version: 25-06-2022

Reading Notes

Q-LEARNING AND DQN

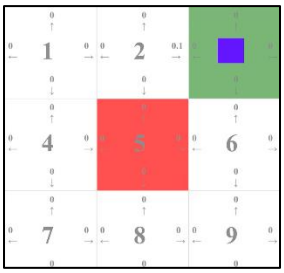
- Animation for Q-learning formula: https://www.youtube.com/watch?v=RuraP4ef4nU

A model-free temporal difference learning on quality functions.

$$Q(s_t, a_t)_{new} = Q(s_t, a_t)_{old} + \alpha \left[\underbrace{r + \gamma \max_{a_{t+1}} Q(s_{t+1}, a_{t+1})}_{\text{TD(0) Reward Estimate y\_t}} - Q(s_t, a_t)_{old} \right]$$

Here, α is the learning rate, r is the current reward and γ represents the discount rate.

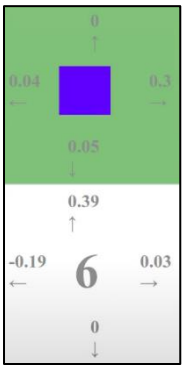
Visualizing the Bellman Equation with a simple example



Instructions:

- Reward +1 if green box, -1 if red box
- Purple box: agent
- α = 0.1 and γ = 0.9

Suppose that the agent explored numerous paths and the box 6 has the following quality functions:



Now, the agent went from box 6 to 5 (up)

- s_t: Box 6
- a_t: Up -> box 6 to 5

- $Q(s_t, a_t)_{old} = 0.39$
- $r = 1$
- $\max_{a_{t+1}} Q(s_{t+1}, a_{t+1}) = 0.3$, which means you will be rewarded (the most) if the next action is to stay at the 5th box.

Therefore, $Q(s_t, a_t)_{new} = 0.39 + 0.1[1 + 0.9 * 0.3 - 0.39] = 0.478$

Initially, the q-value for state and action t was 0.39. Since, the agent moved a green box, that means the agent is rewarded with +1 point, which increases the quality function of this pair of state and action t.

Q-Learning Algorithm

- Reference: https://www.baeldung.com/cs/epsilon-greedy-q-learning

When it comes to choose an action A from state S in each step of one episode, we can either exploit by choosing the action with "the highest estimated reward most of the time" or explore by choosing a random action (epsilon-greedy with a small probability epsilon).

The result of this algorithm is a Q-Table, a matrix filled with different combination of pairs (state, action) that defines the estimated optimal policy π*. An optimal policy is a series of actions that maximize our future rewards.

Algorithm 1: Epsilon-Greedy Q-Learning Algorithm

Data: α: learning rate, γ: discount factor, ϵ: a small number
Result: A Q-table containing Q(S,A) pairs defining estimated optimal policy π^*

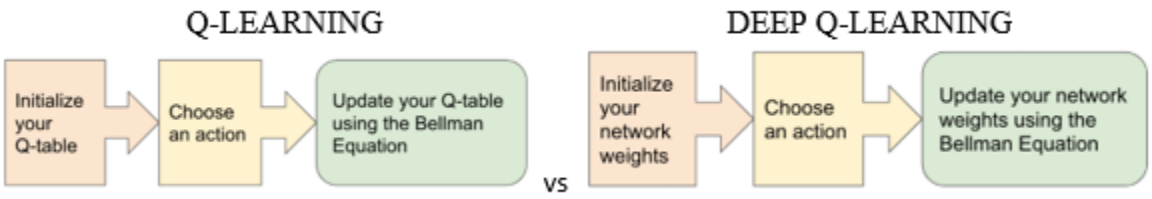
```
/* Initialization                                          */
```
Initialize Q(s,a) arbitrarily, except Q(terminal,.);
Q(terminal,.) ← 0;
```
/* For each step in each episode, we calculate the
   Q-value and update the Q-table                          */
```
for *each episode* **do**
```
   /* Initialize state S, usually by resetting the
      environment                                          */
```
   Initialize state S;
   for *each step in episode* **do**
      do
```
         /* Choose action A from S using epsilon-greedy
            policy derived from Q                         */
```
         A ← SELECT-ACTION(Q, S, ϵ);
         Take action A, then observe reward R and next state S';
         Q(S, A) ← Q(S, A) + α [R + $\gamma \max_a$ Q(S', a) - Q(S, A)];
         S ← S';
      while *S is not terminal*;
   end
end

Algorithm 2: Epsilon-Greedy Action Selection

Data: Q: Q-table generated so far, : a small number, S: current state
Result: Selected action
Function *SELECT-ACTION(Q, S, ϵ)* **is**
   n ← uniform random number between 0 and 1;
   if $n < \epsilon$ **then**
      A ← random action from the action space;
   else
      A ← maxQ(S,.);
   end
   return selected action A;
end

Deep Q-Learning

- Reference: https://towardsdatascience.com/deep-q-learning-tutorial-mindqn-2a4c855abffc#:~:text=Critically%2C%20Deep%20Q%2DLearning%20replaces,process%20uses%202%20neural%20networks.



Instead of mapping a state-action pair to a q-value, DQN maps input states to (action, q-value) pairs. In DQN, we use two neural networks. Every N steps, the weights from the main network are copied to the target network, which helps stability in the learning process.

DQN can also use Experience replay, which helps to train on small batches and speed up the DQN implementation.

- Reference: https://www.researchgate.net/figure/Pseudo-code-of-deep-Q-learning-with-experience-replay_fig3_338776794

Algorithm 1 Deep Q-learning with Experience Replay

Initialize replay memory $\mathcal{D}$ to capacity N
Initialize action-value function Q with random weights
for episode $= 1, M$ **do**
  Initialise state s_t
  for $t = 1, T$ **do**
    With probability ϵ select a random action a_t
    otherwise select $a_t = \max_a Q^*(s_t, a; \theta)$
    Execute action a_t and observe reward r_t and state s_{t+1}
    Store transition (s_t, a_t, r_t, s_{t+1}) in $\mathcal{D}$
    Set $s_{t+1} = s_t$
    Sample random minibatch of transitions (s_t, a_t, r_t, s_{t+1}) from $\mathcal{D}$
    Set $y_j = \begin{cases} r_j & \text{for terminal } s_{t+1} \\ r_j + \gamma \max_{a'} Q(s_{t+1}, a'; \theta) & \text{for non-terminal } s_{t+1} \end{cases}$
    Perform a gradient descent step on $(y_j - Q(s_t, a_j; \theta))^2$
  end for
end for

PAPERS RELATED TO DQN

Paper 1. A Review on Deep Reinforcement Learning for Fluid Mechanics
- Reference: https://arxiv.org/abs/1908.04127

*This paper mentions the theory behind the Deep Q-Learning starting at page 8.

Value-based Methods

2.1.2 Value-based methods

In value-based methods, the agent learns to optimally estimate a *value function*, which in turn dictates the policy of the agent by selecting the action of the highest value. One usually defines the *state value function*:

$$V^{\pi}(s) = \mathop{\mathbb{E}}_{\tau\sim\pi}\big[R(\tau)|s\big],$$

and the *state-action value function*:

$$Q^{\pi}(s,a) = \mathop{\mathbb{E}}_{\tau\sim\pi}\big[R(\tau)|s,a\big],$$

which respectively denote the expected discounted cumulative reward starting in state s (resp. starting in state s and taking action a) and then follow trajectory τ according to policy π. It is fairly straightforward to see that these two concepts are linked as follows:

$$V^{\pi}(s) = \mathop{\mathbb{E}}_{a\sim\pi}\big[Q^{\pi}(s,a)\big],$$

meaning that in practice, $V^{\pi}(s)$ is the weighted average of $Q^{\pi}(s,a)$ over all possible actions by the probability of each action. Finally, the *state-action advantage function* can be defined as $A^{\pi}(s,a) = Q^{\pi}(s,a) - V^{\pi}(s)$. One of the main value-based methods in use is called Q-learning, as it relies on the learning of the Q-function to find an optimal policy. In classical Q-learning, the Q-function is stored in a Q-table, which is a simple array representing the estimated value of the optimal Q-function $Q^{*}(s,a)$ for each pair $(s,a) \in \mathcal{S}\times\mathcal{A}$. The Q-table is initialized randomly, and its values are progressively updated as the agent explores the environment, until the Bellman optimality condition (Bellman & Dreyfus, 1962) is reached:

$$Q^{*}(s,a) = R(s,a) + \gamma \max_{a'} Q^{*}(s',a'). \quad (1)$$

The Bellman equation indicates that the Q-table estimate of the Q-value has converged and that systematically taking the action with the highest Q-value leads to the optimal policy. In practice, the expression of the Bellman equation (Bellman & Dreyfus, 1962) is used to update the Q-table estimates.

Improvements for Vanilla Deep Q-Learning

- Prioritized Replay Buffer
  - Replay Buffer Batch is sampled uniformly, which means we may not select the most important experiences (that are rare during the exploration phase)
  - Schaul et al. (2015) propose the use the training loss value in order to sample the replay buffer (p.9).
- Fixed Q-Targets
  - Since Q-value and the target are moving, this implies big oscillations in the training ("gradient descent algorithm is chasing a moving target"): creates instability
  - We use two neural networks while training the DQN algorithm: one for the quality values and one for the target, so that the target remains fixed for numerous updates from the Q-values (easier learning)
- Double DQN
  - For one Q-network: creates over-estimation of the Q-values, which creates sub-optimal policies
  - Solution: two DQNs, one for selecting the best action, while the other one is used to estimate the target value.

Paper 2. Playing Atari with Deep Reinforcement Learning
- Reference: https://arxiv.org/abs/1312.5602

Abstract

- Model: CNN, trained with a variant of Q-learning (with experience replay and epsilon-greedy for exploration)
    - Input of Q-learning: raw pixels
    - Output of Q-learning: value function estimating future rewards
- Results: 7 Atari 2600 games from Arcade Environment
    - 6 of the games were outperformed
    - 3 of them were surpassed a human expert of Atari

Background

- Loss: $L_i(\theta_i) = \mathbb{E}_{s,a\sim\rho(\cdot)}\left[(y_i - Q(s,a;\theta_i))^2\right]$
- $y_i = \mathbb{E}_{s'\sim\mathcal{E}}\left[r + \gamma \max_{a'} Q(s',a';\theta_{i-1})|s,a\right]$ is the target for iteration i

Note that this algorithm is *model-free*: it solves the reinforcement learning task directly using samples from the emulator $\mathcal{E}$, without explicitly constructing an estimate of $\mathcal{E}$. It is also *off-policy*: it learns about the greedy strategy $a = \max_a Q(s,a;\theta)$, while following a behaviour distribution that ensures adequate exploration of the state space. In practice, the behaviour distribution is often selected by an ϵ-greedy strategy that follows the greedy strategy with probability $1-\epsilon$ and selects a random action with probability ϵ.

Preprocessing

- Each image has 210 x 160 pixels, with 128 color palette -> curse of dimensionality
- Preprocessing of the images: grayscale, down-sampling and crop into a 84 x 84 image that captures roughly the playing area (the final step is to have square inputs)

Model Architecture

- Input of CNN: 84 x 84 x 4 images
- Final layer: fully connected linear layer with a single output for each valid action
- Number of valid actions: 4 to 18 depending on the game
- CNN with DQN approach

Experiments

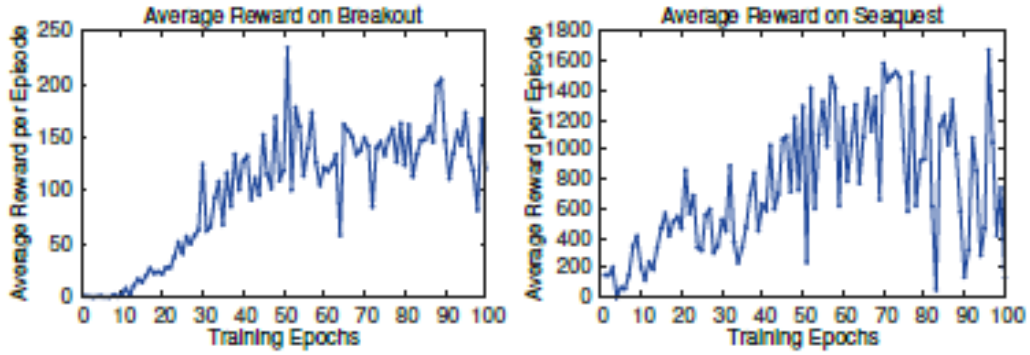
- Perform experiments on 7 ATARI games with the same network architecture **(concept of generalization in RL? Different environments while using the same architecture for training) Zero-Shot Learning**
- RMSProp algorithm with minibatches of size 32
    - Root Mean Squared Propagation: extension of gradient descent that uses a decaying average of partial gradients in the adaptation of the step size for each parameter
    - Definition taken from: https://machinelearningmastery.com/gradient-descent-with-rmsprop-from-scratch/#:~:text=Root%20Mean%20Squared%20Propagation%2C%20or,step%20size%20for%20each%20parameter.

- Behavior Policy: epsilon-greedy, linearly from 1 to 0.1 for the first 1 000 000 frames and stays at 0.1.
- Agent selects actions on every k frame (hyperparameter) instead of every frame. The paper used k = 4 except for Space Invaders (k = 3).

Training and Stability

First chosen metrics: total reward collected in an episode (game) averaged over the number of games

- Very noisy, not much stable progress
- "Small changes to the weights of a policy can lead to large changes in the distribution of states the policy visits."



Second chosen metrics: Policy's estimated action-value function Q (averaged)

- "Estimate of how much discounted reward the agent can obtain by following its policy from any given state."

how much discounted reward the agent can obtain by following its policy from any given state. We collect a fixed set of states by running a random policy before training starts and track the average of the maximum[2] predicted Q for these states. The two rightmost plots in figure 2 show that average

- Curve/function much more smoother and progress is clearly shown here.

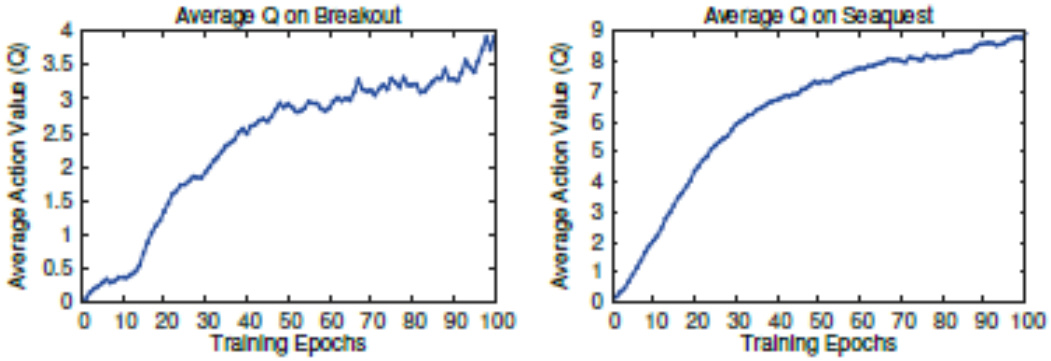


Figure 3: The leftmost plot shows the predicted value function for a 30 frame segment of the game Seaquest. The three screenshots correspond to the frames labeled by A, B, and C respectively.

5.2 Visualizing the Value Function

Figure 3 shows a visualization of the learned value function on the game Seaquest. The figure shows that the predicted value jumps after an enemy appears on the left of the screen (point A). The agent then fires a torpedo at the enemy and the predicted value peaks as the torpedo is about to hit the enemy (point B). Finally, the value falls to roughly its original value after the enemy disappears (point C). Figure 3 demonstrates that our method is able to learn how the value function evolves for a reasonably complex sequence of events.

	B. Rider	Breakout	Enduro	Pong	Q*bert	Seaquest	S. Invaders
Random	354	1.2	0	−20.4	157	110	179
Sarsa [3]	996	5.2	129	−19	614	665	271
Contingency [4]	1743	6	159	−17	960	723	268
DQN	**4092**	**168**	**470**	**20**	**1952**	**1705**	**581**
Human	7456	31	368	−3	18900	28010	3690
HNeat Best [8]	3616	52	106	19	1800	920	**1720**
HNeat Pixel [8]	1332	4	91	−16	1325	800	1145
DQN Best	**5184**	**225**	**661**	**21**	**4500**	**1740**	1075

Table 1: The upper table compares average total reward for various learning methods by running an ϵ-greedy policy with $\epsilon = 0.05$ for a fixed number of steps. The lower table reports results of the single best performing episode for HNeat and DQN. HNeat produces deterministic policies that always get the same score while DQN used an ϵ-greedy policy with $\epsilon = 0.05$.

PAPERS RELATED TO DQN

Paper 1. Morphology-Agnostic Learning (generalization in RL)
- Reference: https://www.linkedin.com/pulse/anymorph-learning-transferable-policies-inferring-agent-trabucco/

Generalization in RL: transferring policies using agent morphology, which eliminates extra work for engineers to provide infos about the design of the agent (morphology)

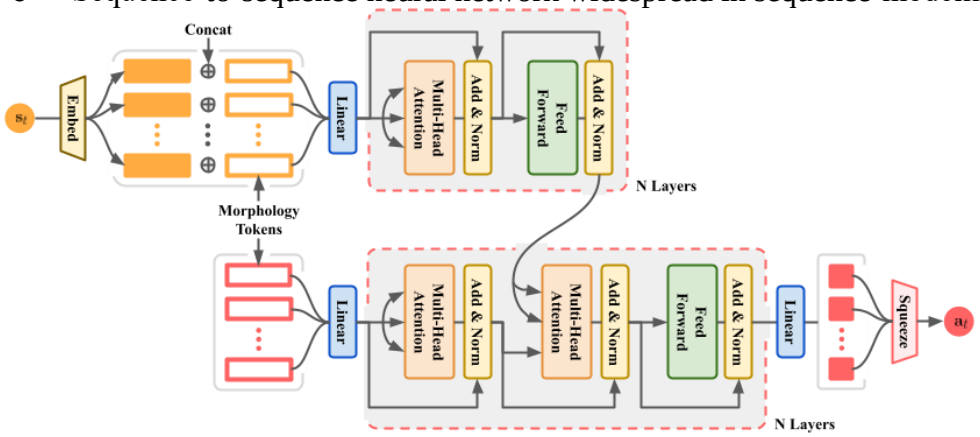
In this paper, controlling different morphologies off a simulated cheetah using the same policy (model architecture).

Motivations of Morphology-Agnostic Learning

- Analogy to muscle memory in humans
- Saves time for RL algorithms and training (for example: if a robot's arm is damaged and needs an upgrade, the robot's morphology is modified, which means that this robot needs to be retrained from scratch).

Approach of Morphology-Agnostic Learning

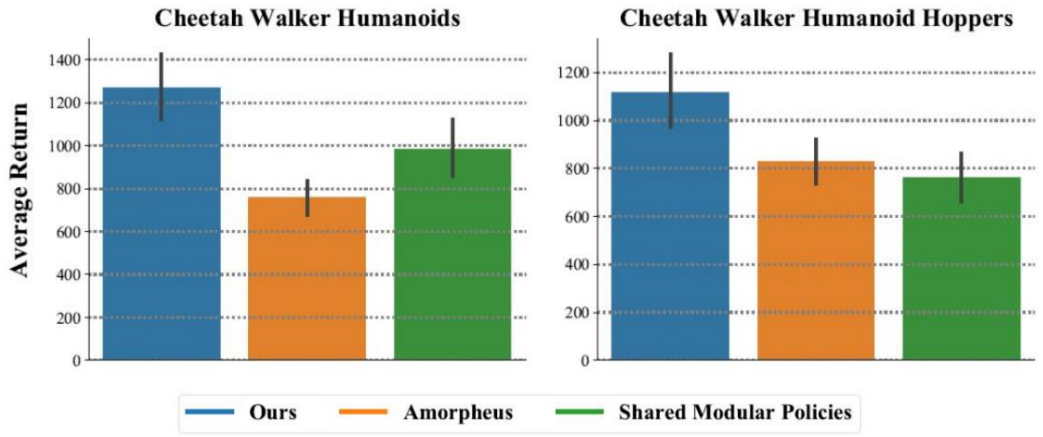
- Agent's design: sequence of learnable vector embeddings (morphology tokens)
  - Inspired from NLP (word tokens in language modeling)
- Model Architecture: Transformer
  - "Sequence-to-sequence neural network widespread in sequence-modelling"



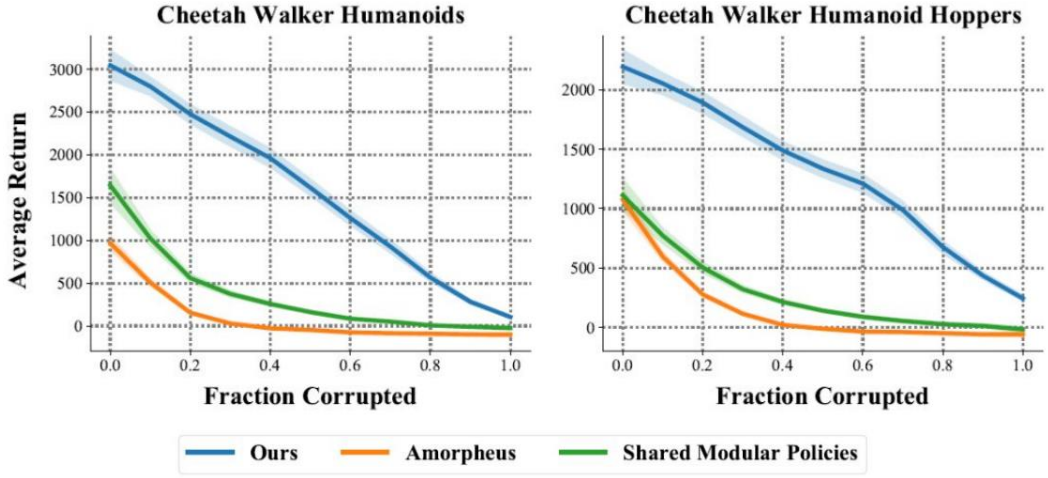
"Crucially, our policy eliminates the need to specify the morphology and design of the agent in advance. Instead, inter-token relationships are discovered with reinforcement learning."

Results

- Chosen metrics: average returns (future discounted rewards)
- Goal: run forward as quickly as possible



Evaluation of robustness to broken sensors



"Our method achieves gains of 28% and 32% on the "Cheetah Walker Humanoids" and "Cheetah Walker Humanoid Hoppers" tasks respectively. Average return is plotted on the y axis at 2.5 million environment steps with a 95% confidence interval over four independent trials and episodes of length 1000."

Paper 2. Survey of Generalization in Deep RL [Zero-Shot Policy]
- Reference: https://arxiv.org/abs/2111.09794

Introduction

- Classical RL: MDP
  - Training = Test (singletons) -> can lead to overfitting
- Generalization RL: CMDP (contextual MDP)
  - Train Distribution = Test Distribution (IID)
  - Or the opposite

- This paper covers SINGLE AGENT RL, not MARL (multi-agent).

Formalising Generalization in RL

Supervised Learning

$$\text{GenGap}(\phi) := \mathbb{E}_{(x,y)\sim D_{test}}[\mathcal{L}(\phi, x, y)] - \mathbb{E}_{(x,y)\sim D_{train}}[\mathcal{L}(\phi, x, y)].$$

- Difference between risk on training data and test data. The model generalizes the data well if this generalization gap is small.

Compositional Generalization

1. **systematicity**: generalisation via systematically recombining known parts and rules,
2. **productivity**: the ability to extend predictions beyond the length seen in training data,
3. **substitutivity**: generalisation via the ability to replace components with synonyms,
4. **localism**: if model composition operations are local vs. global,
5. **overgeneralisation**: if models pay attention to or are robust to exceptions.

Examples with block-stacking environment:
1. Systematicity: stack blocks in new configurations
2. Productivity: how many blocks the agent can generalize to, and complexity
3. Substitutivity: new colors on the blocks (doesn't change the physics of them)

Reinforcement Learning

https://ai.stackexchange.com/questions/20903/what-is-the-difference-between-training-and-testing-in-reinforcement-learning

Contextual Markov Decision Process

- We train and test the policy on different collections of tasks
- Each level or task is specified by some seed, ID or parameter vector.
- CMDP: a MDP where the state s = (c, s'), where c is the context, and s' is the underlying state.
- The context c takes the role of the seed, etc. It changes only between episodes.
- **Hence, CMDP is the entire collection of tasks or environments; each game is a separate CMDP**.

Distribution of p(s)
P(s) = p((c, s')) := p(c) p(s'|c)

If we have 200 different levels in one game, p(c) will be a uniform distribution over the fixed 200 seeds at training time, and over all seeds at testing time.

Training and Testing Contexts

For any CMDP M = <S, A, R, P, C>, we choose a subset of the context set C' and produce a new CMDP M'.

- Any possible subset of the set of all seeds can be used to define a different version of the game with a limited set of levels.

Expected Return of a policy in a CMDP M:

$$\mathbf{R}(\pi, \mathcal{M}) := \mathbb{E}_{c \sim p(c)}[\mathcal{R}(\pi, \mathcal{M}|_c)],$$

For the generalization problem: we choose a training context and testing context C_{train} and C_{test}.

1. Train a policy using the training context-set CMDP for some number of training steps
2. Evaluate the policy on the testing context-set CMDP
3. Objective: have a high expected return on the testing context.

Gap for Generalization in RL:

$$\text{GenGap}(\pi) := \mathbf{R}(\pi, \mathcal{M}|_{C_{\text{train}}}) - \mathbf{R}(\pi, \mathcal{M}|_{C_{\text{test}}}).$$

Gap for Generalization in Supervised Learning:

$$\text{GenGap}(\phi) := \mathbb{E}_{(x,y)\sim D_{test}}[\mathcal{L}(\phi, x, y)] - \mathbb{E}_{(x,y)\sim D_{train}}[\mathcal{L}(\phi, x, y)].$$

Zero-policy transfer

- Transferring the policy from training CMDP to testing CMDP
- No further training in the test context set
- In this generalization problem, no assumptions about shared structure within the CMDP between context-MDPs.

Additional Assumptions for More Feasible Generalization

1. Training and Testing Context Set Distributions
   - Domain Generalization: training and test environments = different domains (same underlying generative distribution)
   - Robotics

2. Structure of the CMDP

Remarks and Discussion

Metrics

- Absolute performance on evaluation tasks
- Generalization gap

Benchmarks for GRL

Categorising Environments That Enable Generalisation

- PCG (Black-Box PCG): Procedural Content Generation
    - Relies on a single random seed to determine multiple choices during the CMDP generation
- Controllable (White-Box PCG)
    - Environments where the context set directly changes the parameters of interest in the CMDPs

Evaluation Protocols for Generalization

- Benchmark: combination of an environment and an evaluation protocol
- Evaluation protocol: specifies the training and testing context sets, any restrictions on sampling from the training set, and the number of samples allowed from the training environment
    - PCG: Context-Efficiency Restriction
        - We can only sample random seeds
    - Controllable
        - Contains factors variation that can be controlled by the user of the environment

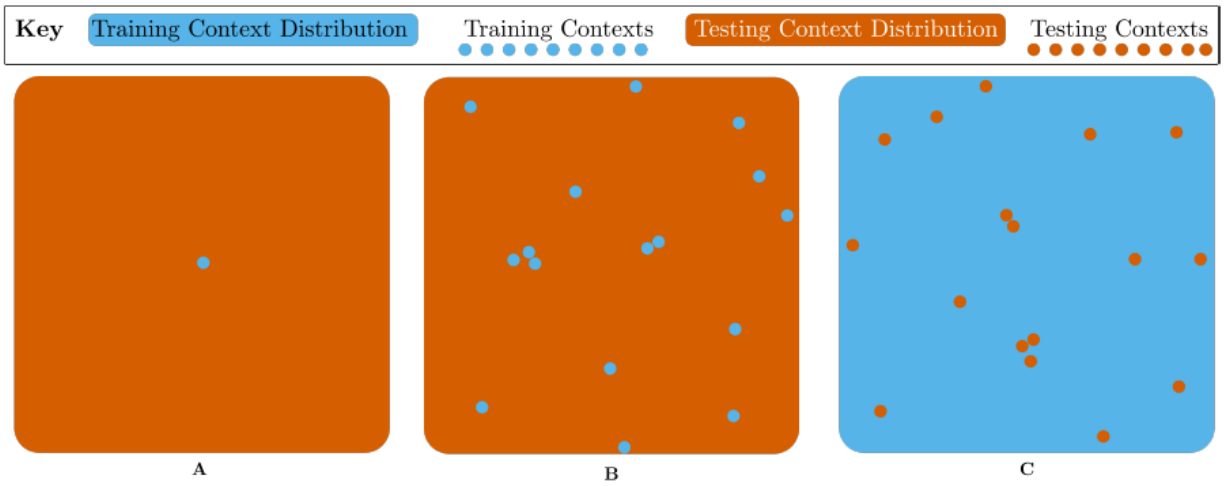


Figure 2: **Visualisation of Evaluation Protocols for PCG Environments. A** is a single training context, and the whole context set for testing. **B** uses a small collection of training contexts randomly sampled from the context set, and the entire space for testing. **C** effectively reverses this, using the entire context set for training apart from several randomly sampled held-out contexts that are used for testing. The lack of axes indicates that these sets have no structure.

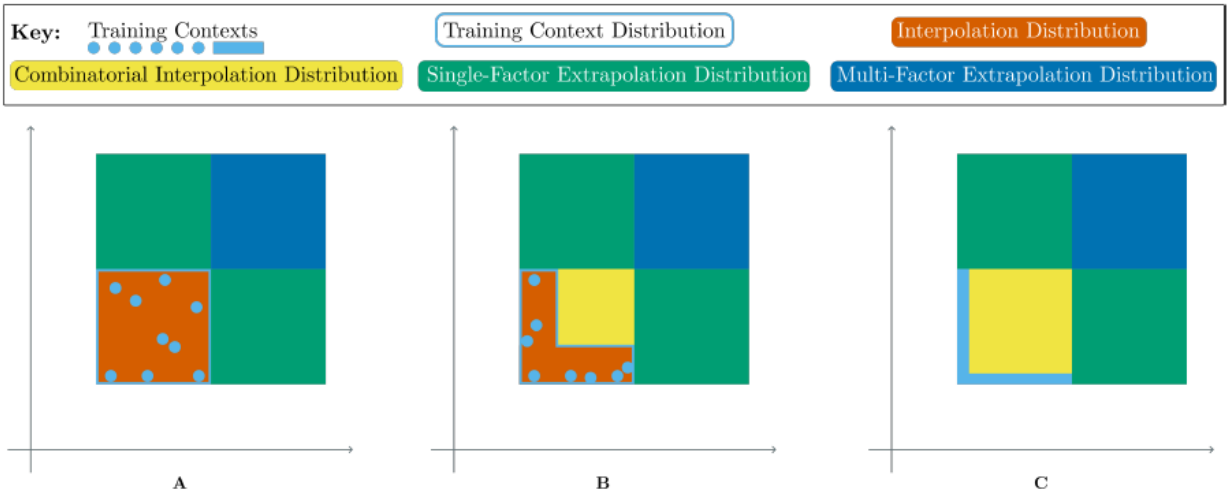


Figure 3: **Visualisation of Evaluation Protocols for Controllable Environments**. Each diagram visualises one possible training context set (blue), and multiple possible testing context sets (all other colours). In **A** we choose the range for each factor of variation independently for the training distribution, resulting in a convex shape for this distribution. In this setting possible testing distributions can either be interpolation (red), extrapolation along a single factor (either green square) or extrapolation along both factors (blue). In **B** and **C** the ranges for each factor of variation are linked together, resulting in a non-convex shape for the training distribution. This allows an additional type of generalisation to be tested, combinatorial interpolation (yellow), where the factors take values seen during training independently, but in unseen combinations. We continue to have the previous interpolation and extrapolation testing distributions. The difference from **B** to **C** is in the width of the training distribution in the axes along which we expect the agent to generalise. In **C** the policy will not be able to learn that the two factors can vary independently at all, making all forms of generalisation harder. Note that in actual environments and real-world settings it is likely this space will be higher than two dimensions and contain non-continuous and non-ordinal axes. The axes indicate that in this setting we have control over these factors of variation, in contrast to Fig. 2.

Methods for GRL

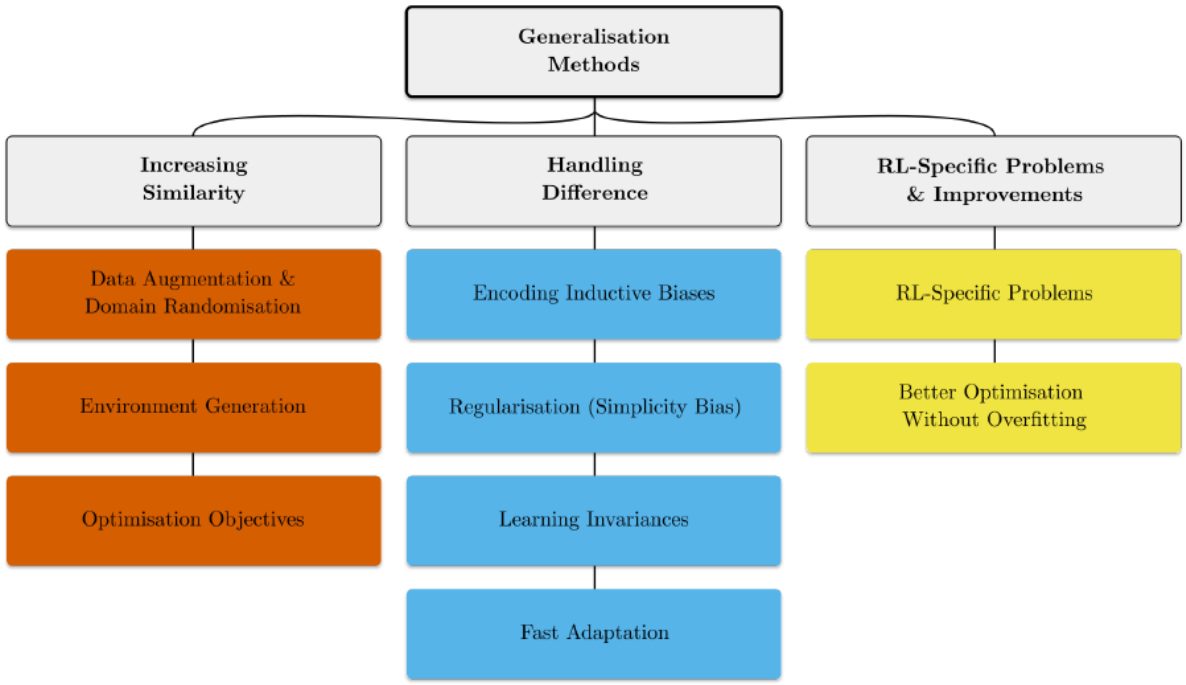


Figure 4: **Categorisation of methods for tackling generalisation in reinforcement learning**

Discussion and Future Work

Generalization beyond Zero-Shot Policy Transfer

- CRL: Continual Reinforcement Learning
  - Methods built for domain generalization in RL
- **Hierarchical and multi-task RL**
  - **Learn subcomponents, skills on source tasks**
  - **Increase learning speed and performance when transferred to novel tasks**
  - **Reuse previous knowledge or skills to learn the new task faster**

Other papers related to multi-task GRL

1. https://arxiv.org/abs/2003.13661
2. https://openreview.net/forum?id=SJx63jRqFm