A Thesis Title

Author Name

A dissertation submitted in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
of
University College London.

Department of Something
University College London

June 22, 2023

I, Author Name, confirm that the work presented in this thesis is my own. Where information has been derived from other sources, I confirm that this has been indicated in the work.

Abstract

My research is about stuff.

It begins with a study of some stuff, and then some other stuff and things.

There is a 300-word limit on your abstract.

Acknowledgements

Acknowledge all the things!

Contents

List of Figures

List of Tables

Chapter 1

Introduction

The power of symmetry in understanding the physical world has been surprisingly effective. Perhaps one of the most famous examples of this is the eightfold way of Gell-Mann[4], which brought much deeper insight into the structure of elementary particles. Further, Noeter's theorem states that symmetries are the source of conservation laws, such as energy conservation, momentum, and angular momentum.

Additionally, the amount humans leverage symmetry's power in simple everyday reasoning problems should not be underestimated. When picking up a box, we rely heavily on our implicit understanding of symmetries. We know that the same approach works for two boxes rotated 30^o from each other. This is an understanding that is not necessarily present in a Deep Reinforcement Learning agent.

These ideas about the world and associated rules are known as inductive biases. In order to make the agent understand these symmetries, there are two main approaches to induce these ideas into the agent. These approaches are data augmentation and encoding the invariance into the agent. Data Augmentation techniques use known symmetries about the environment and create artificial training data by transforming the input such that the symmetry is respected, for example, flipping the input image[9, 12]. Such techniques also have much history in supervised learning, where it is common to increase the amount of data by performing transformations on the input that preserve the output. The second approach is to structure the neural networks in the agents such that they may only learn behaviours that respect the symmetry of the environment[25, 27, 15]. Additionally, because of the underlying problem of generalising to symmetries, having networks respect invariance was one of the key breakthroughs in deep learning with the ConvNet[10], which respect translational invariance, which is particularly important for computer vision. Further, the G-convolution[2] provides this equivariance behaviour for more general groups.

. . . Something about the specific problem. . .

Chapter 2

Background

2.1 Markov Decision Process

Bellman[1] in 1957 describes a framework for solving stochastic control problems. A classic example of this is playing blackjack. There is often no certain outcome to playing a given hand. However, actions exist for which the expected probability of a player winning is higher. The Markov Decision Process formalism allows one to tackle such problems systematically, with mathematically tractable convergence bounds to optimal solutions in some cases.

The MDP describes these processes as an agent acting in an environment that is, in general, partially observable.

This is described by a tuple $(\mathcal{S}, \mathcal{A}, \mathcal{R}, P, \gamma)$. $\mathcal{S}$ is the state space, the state of the world that the agent knows about, and $\mathcal{A}$ is the action space, the possible actions it may take. For a given state and action, the reward provided is real-valued and may be stochastic, $\mathcal{R} : \mathcal{S} \times \mathcal{A} \rightarrow \mathbb{R}$. P are the transition probabilities between states $P : \mathcal{S} \times \mathcal{A} \times \mathcal{S} \rightarrow [0, 1]$, this defines the dynamics of the process. γ determines the reward discount and how myopic the MDP is.

The agent aims to maximise its expected discounted cumulative reward, the return, $\mathbb{E}[G_t = \sum_t^T \gamma^t R_t]$. The problem may be that of an episodic setting with a clear termination, a chess player winning, for example. Alternatively, it may be in a continuous setting with no clear termination, like managing stocks. To make decisions, an agent follows a policy. The policy, $\pi(a|s), \pi : \mathcal{S} \times \mathcal{A} \rightarrow [0, 1]$ maps states' actions to probabilities of taking action a, in the deterministic case it is just a map from state to action space.

A family of algorithms known as dynamic programming exists that, given finite states and actions, can find exact solutions to these problems, under reasonable constraints. However, these algorithms are often intractable for large state and action spaces. solutions to MDPs can be expressed in terms of the value function, $V(s) = \mathbb{E}[G_t|s_t = s]$, which is the expected return from a given state.

The optimal value function, $V^*(s)$, is the maximum value function over all possible policies. The optimal policy, $\pi^*(s)$, is the policy that maximises the value function for all states. The optimal value function and policy satisfy the Bellman optimality equations[1]:

$$V^*(s) = \max_a \mathbb{E}[R_{t+1} + \gamma V^*(s_{t+1})|s_t = s, a_t = a]. \tag{2.1}$$

The value function is very closely linked to the action-value function $Q(s,a) = \mathbb{E}[G_t|s_t = s, a_t, = a]$ which is the expected return from taking action a in state s and then following the current policy, there also exists a bellman optimality equation for the action-value function:

$$Q^*(s,a) = \mathbb{E}[R_{t+1} + \gamma \max_{a'} Q^*(s_{t+1}, a')|s_t = s, a_t = a]. \tag{2.2}$$

Using dynamic programming and finding exact solutions to an MDP and the optimal policy $\pi*$ to optimise your return is often impossible. Reinforcement learning is a common alternative to dynamic programming, which provides a principled approach to learning approximate solutions. It describes a family of techniques that learn from previous experience to improve at solving MDPs.

2.2 Reinforcement Learning

Reinforcement Learning (RL) encompasses the techniques used to solve MDPs where experience is required to solve the MDP. When framed as MDPs, many real-world problems have such large state or action spaces that some approximations must be made to find approximate solutions. Despite the approximate nature of their solutions, RL agents have achieved superhuman performance in chess, go, starcraft and other games and have recently found diamonds in Minecraft [22, 23, 6]. Many of these successes with current RL approaches highlight the challenges current algorithms face due to their hunger for data. Games, especially games that can be played on computers, are relatively cheap to sample data from and can be played many times[1]. This sample hunger, combined with an inability to transfer knowledge between tasks, highlights that there is a significant advantage to encoding prior knowledge into RL algorithms. The knowledge of symmetries enables agents to attempt simplified problems and draws abstract connections between different tasks. This thesis focuses on the encoding of symmetries into RL algorithms. This section will describe the most common approaches to solving RL problems.

2.3 Deep Reinforcement Learning Approaches

The underlying optimisation problem in reinforcement learning is finding a policy which produces the maximum expected reward. However, not all RL algorithms take this approach directly. Additionally, many algorithms choose to forgo learning the dynamics of a process and instead model the value function or policy directly. These are Model-Free algorithms. In contrast, model-based algorithms learn the problem's dynamics and simulate the agent's behaviour to learn. This section will describe the most common approaches to solving RL problems. Typically, learning algorithms face a few persistent problems in Deep RL that make solving problems difficult. These are;

- **Sample Efficiency** The algorithms require a large amount of data to learn. This is because training large neural networks requires lots of data to learn. As mentioned before, breakthroughs in RL are often made in games, where data can be simulated and collected cheaply[3]

- **Training Robustness** The algorithms can often struggle to converge depend-

[1] It is worth noting that RL has not just solved problems that pertain to games, but has been very successful in solving robotic control problems

ing on the initialisation of the environment and the agent, and special random seeds are required to see good performance[7].

- **Reward Sparsity** Reward functions are often sparse, and coming across states that provide rewards, such as finding diamonds in Minecraft, is a difficult problem when initial exploration is random.[6].

2.3.1 Model Based Algorithms

Model-based reinforcement learning is the process of the agent learning the dynamics of a system and then performing ”planning” to decide how best to act next. In concrete terms, this corresponds to an agent learning the transition dynamics of the MDP, thus learning both the subsequent state of a state action pair, $P(s'|s,a)$ and the expected reward from a state action pair $r(s,a)$. From the learned model agent ”plans” by simulating the environment and acting in the simulated environment. Then the agent learns a policy with the simulated data. Common algorithms like Dyna-Q, use Q-learning over a comnination of real and simulated data to learn a policy.

Model-based algorithms have the potential to provide much better sample efficiency than model-free algorithms, as they can learn from simulated data. However, this in itself can be difficult to learn. However, the Dreamer models have shown substantial success in tackling difficult problems[6, 5]. This report does in some way, build a world model. However, it is not used for planning. As such, the details of model-based algorithms will not be discussed further.

2.3.2 Model Free Algorithms

Model-free algorithms forgo trying to build an approximation of the MDP’s transition dynamics defined by P and instead try to optimise the agent behaviour by either approximating how good different states are or learning the best action to take in given states. When combining standard dynamic programming algorithms such as q-learning with function approximation, or actor-critic methods, the challenge is finding a way to learn the form of the parametric function approximator, which is often a neural network. As such, there is a common approach between multiple algorithms, where you form a supervised learning problem, finding a function that minimises a loss. In the RL setting, the data is gained from the MDP. The fundamental difference between Q-Learning and Actor-Critic approaches is that Q-learning uses an epsilon greedy policy to collect data and experience. Whereas, Actor Critics learn a stochastic policy from the data. However, the training loop is much the same; In most cases, some operations need to be performed on the replay buffer, and they store more data than just sequential $(s, a, r)_t$ tuples. Often subsequent actions are required to calculate the loss. This can be done during the update loop. Finally, rather than interacting with one environment, training is done in parallel in multiple environments. The process is much the same; states, rewards and actions become vectors for this. Environment vectorisation has improved training stability and robustness [13].

Algorithm 1 Model Free RL Training Loop

```
Initialise $\theta$ randomly.                                   ▷ Network Parameters
Initialise $t = 0$                                               ▷ Buffer Timestep
Initialise $T = 0$                                                ▷ Total Timestep
Initialise replay buffer $\mathcal{D} \leftarrow \emptyset$
Sample state $s$
while do $T < T_{max}$
    Take action $a$ according to policy $\pi(a|s)$
    Observe reward $r$ and next state $s'$
    Store transition $(s, a, r)$ in replay buffer $\mathcal{D}$
    ▷ Replay buffer respects temporal order of experience.                  ◁
    $s \leftarrow s'$
    $t \leftarrow t + 1$
    $T \leftarrow T + 1$
    ▷ If a state is terminal, then reset the environment. Continue to sample ex-
      perience. Masking is used to ensure that terminal states have zero value. ◁
    if $t \mod T_{experience} == 0$ then                         ▷ Update Loop
        Perform minibatch gradient descent updates $\theta$ to minimise $L(\mathcal{D}, \theta)$
        ▷ Where L is the loss function, these losses are very different in the AC
          and Q-Learning, but the training loop is the same.                ◁
        $t \leftarrow 0$
        $\mathcal{D} \leftarrow \emptyset$
```

2.3.2.1 Q-Learning

Q-Learning forgoes optimising the true objective, the policy, to learn the optimal value function. This allows one to find an optimal policy, as there is always a greedy policy with a stationary point at the optimal value function. Dynamic Programming exploits this idea in algorithms such as Value Iteration[1] and Policy Iteration[8]. As it is approximate, the convergence guarantees to optimal policies are absent in Deep-Q-Learning. An exact solution may not be found due to computational restraints or partial observability. However, excellent solutions can be found in practice using this method [14]. Traditional Q learning updates the value of the current state-action pair, $Q(s, a)$, using the Bellman optimality equation for the action-value function,

$$Q_{t+1}(s,a) = Q_t(s,a) + \alpha \left[r(s,a) + \gamma \max_{a'} Q_t(S_{t+1}, a') \right]. \tag{2.3}$$

Where α is the step size, all other symbols take their normal values. The important point is that this converges to the optimal value function given infinite state action space visitation. In the setting of Deep Q Learning, we lose the guarantees of convergence in exchange for flexibility.

The Bellman optimality equation can be used to construct a loss function to train a parametric function approximation, a deep learning model, through gradient methods. However, gradient-based methods are data-hungry, and if this method were employed in an on-policy fashion with a deterministic greedy optimal policy, there would be no exploration. As such, the agent would not explore, so an alternative exploration policy is used. The results of its trajectory or trajectories, depending on whether it is an episodic setting, are stored in a Replay Buffer,

$$\mathcal{D} = \{S_0, A_0, R_1, S_1, A_1, \dots\} \tag{2.4}$$

This can be sampled when training the network. The loss function to optimise is derived from the Bellman equation,

$$L_w(\mathcal{D}) = \mathbb{E}_{(s,a,r,s')\sim\mathcal{D}}\left[\left(Q_w(s,a) - (r + \gamma(1 - \mathbb{I}(s' = S_T))\max_{a'} Q_{w'}(s',a')\right)^2\right]. \tag{2.5}$$

Here, it is essential to note that Q'_w can be the original network if $w = w'$. However, there are some problems associated with this. The gradient with respect to the loss must now become a semi-gradient, where the Q'_w term must be considered a constant with respect to w. The indicator, $\mathbb{I}(s' = S_T)$, is one if s' is a terminal state.

Alternatively, the semi-gradient can be avoided with Double-Q-Learning. Double-Q-Learning uses target networks[26]. This makes two copies of the primary, Q_w, network and performs gradient updates on one, Q_w, with the secondary network, $Q_{w'}$, parameters being moved towards w after some further number of updates so that its parameters lag that of w. This can be done with weighted averages or just a direct update.

Some problems arise in continuous state spaces, where the $\max$ operation may involve an optimisation loop. This is not ideal. Algorithms such as DDPG[11] use another network to parameterise an optimal policy instead of the max operation. DDPG is an example of an Actor-Critic algorithm that straddles the divide between Q-Learning and Policy Based Methods.

2.3.2.2 Policy Based Methods

While Policy Based Methods all strive to optimise the expected cumulative return for an agent, there are quite a few subtleties between the dominant gradient-based approaches. In the episodic setting, where it is possible to obtain complete trajectories and optimise for episodic returns, policy gradient functions can directly optimise the expected cumulative reward,

$$J_G(\theta) = \mathbb{E}_{s_0\sim d_0}\left[v_{\pi_\theta}(S_0)\right]. \tag{2.6}$$

However, in infinite time horizon tasks, there is no termination and optimising for the expected reward of the next action may be a more prudent objective,

$$J_R(\theta) = \mathbb{E}_{\pi_\theta}\left[R_{t+1}\right]. \tag{2.7}$$

Many common algorithms for episodic tasks can be extended for infinite time horizon problems. In both cases, it should be noted that the objective function from which we are taking the gradient is an expectation. This provides a problem, and we use the score function trick to take the gradient before the expectation.

One uses the score function trick to derive the gradient-based update term in the episodic case. First, the expected reward is expressed in its natural form as the average return under the distribution of trajectories defined by the policy and the MDP's dynamics and proceed to do some algebra to get it into a more friendly form,

$$\begin{aligned}
\nabla_\theta J_G(\theta) &= \nabla_\theta \mathbb{E}_{\tau\sim\pi_\theta}\left[R(\tau)\right] && (2.8)\\
&= \nabla_\theta \sum_\tau R(\tau)P(\tau|\theta) && (2.9)\\
&= \sum_\tau R(\tau)\nabla_\theta P(\tau|\theta) && (2.10)\\
&= \sum_\tau R(\tau)P(\tau|\theta)\frac{\nabla_\theta P(\tau|\theta)}{P(\tau|\theta)} && (2.11)\\
&= \mathbb{E}_{\tau\sim\pi_\theta}\left[R(\tau)\nabla_\theta \log(P(\tau|\theta))\right]. && (2.12)
\end{aligned}$$

It would be ideal if the trajectories' probability could be calculated without any knowledge of the MDP's dynamics $P(S_{t+1}|S_t, A_t)$. This can be done by exploiting the definition of the trajectory's probability.

$$P(\tau|\theta) = P(S_0)\prod_{t=0}^{T} P(S_{t+1}|S_t, A_t)\pi_\theta(A_t|S_t) \tag{2.13}$$

Thus using log rules, the expectations can be rearranged in the form,

$$\nabla_\theta J_G(\theta) = \mathbb{E}_{\tau\sim\pi_\theta}\left[R(\tau)\nabla_\theta \right. \tag{2.14}$$

$$\left.\left\{\log(P(S_0)) + \sum_{t=0}^{T}\log(P(S_{t+1}|S_t, A_t)) + \log(\pi_\theta(A_t|S_t))\right\}\right], \tag{2.15}$$

$$= \mathbb{E}_{\tau\sim\pi_\theta}\left[R(\tau)\nabla_\theta \sum_{t=0}^{T}\log(\pi_\theta(A_t|S_t))\right]. \tag{2.16}$$

With this form of the policy gradient, we could sample multiple trajectories and perform gradient descent on the policy.

This would be an unbiased estimate of the policy gradient. Empirically, the variance of the gradient updates for deep networks is inefficient in the number of training samples or impossible, depending on the task. One further remedy to this is that we can note that a policy at time $t = t'$ should not depend on the rewards in the past; this insight can be seen by splitting the expectation,

$$\nabla_\theta J_G(\theta) = \mathbb{E}_{\tau\sim\pi_\theta}\left[R(\tau)\nabla_\theta \sum_{t=0}^{T}\log(\pi_\theta(A_t|S_t))\right] \tag{2.17}$$

$$= \mathbb{E}_{\tau\sim\pi_\theta}\left[\left(\sum_{k=0}^{T} r(S_k, A_k)\right)\left(\nabla_\theta \sum_{t=0}^{T}\log(\pi_\theta(A_t|S_t))\right)\right] \tag{2.18}$$

$$= \sum_{t=0}^{T}\mathbb{E}_{\tau\sim\pi_\theta}\left[\sum_{k=0}^{t-1} r(S_k, A_k)\nabla_\theta \log(\pi_\theta(A_t|S_t))\right. \tag{2.19}$$

$$\left. + \sum_{k'=t}^{T} r(S_{k'}, A_{k'})\nabla_\theta \log(\pi_\theta(A_t|S_t))\right]. \tag{2.20}$$

As the rewards at time t are only correlated to the actions only at time t, and none of the previous times, the left-hand side of the expectation can be factorised into two expectations,

$$\mathbb{E}_{\tau\sim\pi_\theta}\left[\sum_{k=0}^{t-1} r(S_k, A_k)\nabla_\theta \log(\pi_\theta(A_t|S_t))\right] \tag{2.21}$$

$$
\begin{aligned}
&= \mathbb{E}_{\tau\sim\pi_\theta}\left[\sum_{k=0}^{t-1} r(S_k, A_k)\right] \mathbb{E}_{\tau\sim\pi_\theta}\left[\nabla_\theta \log(A_t|S_t)\right] \\
&= \mathbb{E}_{\tau\sim\pi_\theta}\left[\sum_{k=0}^{t-1} r(S_k, A_k)\right] \mathbb{E}_{\tau\sim\pi_\theta}\left[\frac{\nabla_\theta \pi_\theta(A_t|S_t)}{\pi_\theta(A_t|S_t)}\right] \\
&= \mathbb{E}_{\tau\sim\pi_\theta}\left[\sum_{k=0}^{t-1} r(S_k, A_k)\right] \sum_{A_t} \pi_\theta(A_t|S_t)\frac{\nabla_\theta \pi_\theta(A_t|S_t)}{\pi_\theta(A_t|S_t)} \\
&= 0.
\end{aligned}
$$

Thus equation2.20 becomes,

$$
= \sum_{t=0}^{T} \mathbb{E}_{\tau\sim\pi_\theta}\left[\sum_{k'=t}^{T} r(S_{k'}, A_{k'})\nabla_\theta \log(\pi_\theta(A_t|S_t))\right]. \tag{2.22}
$$

The sum over future rewards is the expected return. Combined with the log derivative's expectation of zero. This formulation enables lower variance updates with the same expectation. A detailed review of this is available in [19]. The general policy gradient is of the form,

$$
\nabla J_G(\theta) = \mathbb{E}_{\tau\sim\pi_\theta}\left[\sum_{t=0}^{T} \mathcal{R}_t \nabla_\theta \log(\pi_\theta(A_t|S_t))\right]. \tag{2.23}
$$

Where $\mathcal{R}_t$ is a return estimator a time t, if the return estimator is the discounted sum of rewards, this is the REINFORCE algorithm [28]. The most used modern policy gradient algorithms take this form: a return estimator either sampled or bootstrapped with a possible constant baseline offset. This leads to Actor-Critic methods, where the actor is the policy network, and the critic is a value function estimation network.

2.3.2.3 Actor Critic Methods

Actor critics in deep learning typically consist of a pair of function approximation networks, one the actor to approximate the optimal policy, the other to approximate the optimal value function. Because the gradient of a constant is zero, having a baseline that is independent of the policy does not affect the expectation of the policy gradient estimator, but if picked wisely, it may reduce the variance,

$$\nabla J_G(\theta) = \mathbb{E}_{\tau \sim \pi_\theta} \left[\sum_{t=0}^{T} (\mathcal{R}_t - b) \nabla_\theta \log(\pi_\theta(A_t|S_t)) \right]. \tag{2.24}$$

As the variance of the updates depends on the magnitude of the $(\mathcal{R}_t - b)$ term, a function is learned to estimate the value of the state, $V_\phi(S_t)$, and the baseline is set to the value of the state, $b = V_\phi(S_t)$. The critic trained to minimise the mean squared error between the estimated value and the actual return. This produces an unbiased update of lower variance, which has better training behaviour[24].

2.3.3 Generalised Advantage Estimation

Generalised advantage estimation[20], rather than using the discounted return as a learning target. It learns the λ-return, a weighted sum of the n-step discounted returns. The λ-return is defined as,

$$R_t^\lambda = (1 - \lambda) \sum_{n=1}^{\infty} \lambda^{n-1} R_{t:t+n}^V. \tag{2.25}$$

Where $R_{t:t+n}^V$ is the n-step discounted return and lambda is a hyperparameter that controls the trade-off between bias and variance on the return estimator as it gradually uses its own estimates for states values, when $\lambda = 0$, it is an unbiased return estimate. The n-step discounted return is defined as,

$$R_{t:t+n}^V = \sum_{k=0}^{n-1} \gamma^k r_{t+k} + \gamma^n V_\phi(S_{t+n}). \tag{2.26}$$

The lambda return provides a biased but lower variance return estimator, which can benefit training and is the backbone of many modern policy gradient algorithms. Typically the lambda return is calculated recursively backwards in episodic settings,

$$R_t^\lambda = r_t + \gamma(1 - \lambda) V_\phi(S_{t+1}) + \lambda R_{t+1}^\lambda. \tag{2.27}$$

And then the advantage is the difference between the value function and the lambda return,

$$A_t^\lambda = R_t^\lambda - V_\phi(S_t). \tag{2.28}$$

Having the generalised advantage as the optimisation target provides control over the variance of the gradient estimator. The generalised advantage estimator is defined as,

$$\nabla J_G(\theta) = \mathbb{E}_{\tau\sim\pi_\theta}\left[\sum_{t=0}^{T} A_t^\lambda \nabla_\theta \log(\pi_\theta(A_t|S_t))\right]. \tag{2.29}$$

In this situation, the critic network learns to minimise the advantage. Advantage estimation is often used with regularisation methods that stop the current policy from moving too far between training updates. These regularisation methods penalise the KL divergence between the current and previous policies. This is the case for TRPO[18] and the computationally efficient and simple PPO[21].

2.3.3.1 PPO

Proximal policy optimisation (PPO) follows the same lines as TRPO, as it is a regularised form of gradient-based policy optimisation with a critic that learns a value function as a bias reduction method. The PPO paper introduces two forms: a hard thresholding method PPO-Clip, and a soft regularisation method PPO-Penalty. The clip optimisation target removes the need for explicitly calculating a KL-divergence for a single interaction is,

$$L(s, a, \theta_k, \theta) = \min\left(\frac{\pi_\theta(a|s)}{\pi_{\theta_k}(a|s)} A^{\pi_{\theta_k}}(s, a), g\left(\epsilon, \frac{\pi_\theta(a|s)}{\pi_{\theta_k}(a|s)}\right) A^{\pi_{\theta_k}}(s, a)\right). \tag{2.30}$$

Where the update rule is defined by the gradient descent algorithm, and θ_k is the previous value of θ. The advantage of a policy π is given by a truncated λ-return,

$$A^\pi(s, a) = V_\phi^\pi(s) - R_t(s, a). \tag{2.31}$$

Where the $V_\phi^\pi(s)$ is the critic, typically a neural network. $R_t(s, a)$ is the sampled return for the state action pair. As the algorithm is off-policy, there is an importancesampling correction to the advantage. To stop wide variance in the policy, like TRPO, the magnitude of the update is capped with the g function,

$$g(\epsilon, A) = (1 + \text{sign}(A)\epsilon)A. \tag{2.32}$$

The ϵ hyperparameter must be set. This loss function ensures that there are no overly large policy updates.

2.4 Groups, Symmetries, Homomorphisms

2.4.1 Groups

Groups are an abstract mathematical idea on a set with a binary operation, ”·”. To form a group, the members of a set must satisfy the following:

1 Closure: applying the group’s operation maps all elements back onto another element.

2 Possession of an identity element: there must be an element of the set such that it and any element is mapped onto itself.

3 Possession of inverse elements: every element in the group has an inverse element.

4 Associativity: $(a \cdot b) \cdot c = a \cdot (b \cdot c)\ \forall a, b, c$

The key point is that specific symmetries form groups of all the transformations that leave the object/space invariant.

2.4.2 Invariances

Invariances are properties maintained under a transformation of the input space, e.g. mirror symmetry, where the distance to a point on the mirror line is the same from the left and the right if the object is symmetric. If you have a function $f : \mathcal{X} \rightarrow \mathcal{Y}$ that is invariant to a transformation $\pi_g^x : \mathcal{X} \rightarrow \mathcal{X}$, this is expressed as:

$$f(x) = f(\pi_g^x(x)), \forall g \in G, \forall x \in \mathcal{X}. \tag{2.33}$$

For the transformation to be a group, there must also include the identity element s.t $\pi_h^x(x) = x, h \in G, \forall x \in \mathcal{X}$. This is how symmetry is typically expressed. The symmetry belongs to an abstract group. For example, suppose an object has 3-fold rotation symmetry around an axis. In that case, an abstract group with operations between its elements mirrors that of rotations around the axis by $120^o, 240^o$ and 0^o. It is straightforward to check that these operations form a group. Rotations by 360^o left or right are the identity element. The group is closed; no matter how many rotations are performed, the object with the symmetry is still the same. The inverse of the rotation by 120^o is the rotation by 240^o and vice versa. Finally, the rotations are associative. This is the group of symmetries of the object. The object is invariant to the transformations in the group.

2.4.3 Equivariance

Equivariance is related in that if there is some transformation $\pi_g^y : \mathcal{Y} \to \mathcal{Y}$, where it is a different representation of the group, equivariance which is a more general statement than invariance is,

$$\pi_g^y(f(x)) = f(\pi_g^x(x)), \forall g \in G, \forall x \in \mathcal{X}. \tag{2.34}$$

Here, both π_gs are the same transformation. However, the spaces they act upon are different. This notion is especially important in the space of RL. From this definition, we can see that invariance is a particular case of equivariance where π_g^y is the identity, $\forall g \in G$.

2.4.4 Homomorphisms

A homomorphism describes a map between two structures that preserves an operation. In the context of reinforcement learning, the preservation is that of the dynamics of a Markov Decision Process.

2.5 MDP Homomorphism

An MDP homomorphism describes a surjective mapping between one MDP and another; Ravindran and Barto first described this notion in the early 2000s[16, 17].

Definition: MDP Homomorphism Given some MDP $\mathcal{M}$, where there exists a surjective map, from $\mathcal{S} \times \mathcal{A} \rightarrow \overline{\mathcal{S}} \times \overline{\mathcal{A}}$. The MDP $\overline{\mathcal{M}}$[2] is an abstract MDP over the new space. The homomorphism h, then is the tuple of $(\sigma, \alpha_s | s \in \mathcal{S})$, where $\sigma : \mathcal{S} \rightarrow \overline{\mathcal{S}}$ and $\alpha_s : \mathcal{A} \rightarrow \overline{\mathcal{A}}$. This surjective map must satisfy two constraints for it to be a valid MDP homomorphism;

1. $R(s, a) = R(\sigma(s), \alpha_s(a))$
2. $P(s'|a, s) = P(\sigma(s')|\alpha_s(a), \sigma(s))$

Given this formulation, they show that there are equivalences between the real optimal value function and the optimal abstract value function,

$$V^*(s) = \overline{V}^*(\sigma(s)).$$
$$Q^*(s, a) = \overline{Q}^*(\sigma(s), \alpha_s(a)).$$

This is *optimal value equivalence*. Further, they introduced the idea of ”lifting”, where a policy learned in the abstract space can be mapped to the original space. The lifted policy $\pi^{\uparrow}$ is related to the abstract policy $\overline{\pi}$ by dividing by the preimage, the set of inputs to the homomorphic mapping that map to the same output.

$$\pi^{\uparrow}(a|s) = \frac{\overline{\pi}(\alpha_s(a)|\sigma(s))}{|a \in \alpha_s^{-1}(\bar{a})|}. \tag{2.35}$$

2.5.1 Group Structured MDP Homomorphisms

A natural homomorphic mapping is that of an MDP that possesses symmetry, which is concretised by having the homomorphisms be the representations of the group in state and action space,

1. $R(s, a) = (\pi_g^s(s), \pi_g^a(a))$,
2. $P(s'|a, s) = P(\pi_g^s(s')|\pi_g^a(a), \pi_g^s(s))$.

Where π_g^x is the representation of the element $g \in G$ in the space $\mathcal{X}$, in plain English, for each state or state action pair, there are a set of states with the same

[2] All abstract quantities are denoted with an $\overline{\text{overline}}$

reward and transition probability to some new state, where these states are related to each other by the operations of the elements of g on them.

The abstract MDP can then be solved with Dynamic Programming or approximate Dynamic Programming techniques, and the policy found in the abstract MDP can be ”lifted” to the original MDP.

2.6 The Infamous Cartpole

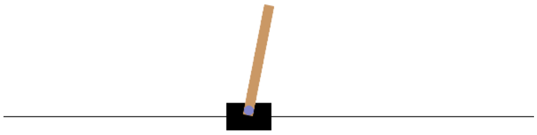

Figure 2.1: The Cartpole environment.

The Cartpole environment is the ”Hello World” of reinforcement learning; it describes a game where at each timestep, the aim is to balance a mass above a slider, 1 point is given for each timestep that the mass makes an angle of fewer than 12 degrees from vertical. The action space$a \in 0, 1$, for the problem, is to either push the pole left or right. The state space is the position of the cart, the velocity of the cart, the angle of the pole and the angular velocity of the pole; as such, $s \in \mathbb{R}^4$. Typically, the episodes are truncated at 500 timesteps, and the reward is 1, so the return is an integer $G_0 \in [0, 500]$. An episode terminates if the pole falls over or the cart moves too far from the centre. If a human were learning to solve this problem, one would notice that the expected result of pushing the cart left when it is leaning right at some positions is the same as pushing the cart right when it is leaning left at the displacement in the opposite direction. This is an example of symmetry in the problem, and it is this symmetry that we will attempt to exploit to improve the learning characteristics of the agent.

The symmetry present in Cartpole is the cyclic C_2 group. Other than the trivial group, the group with a single element, the C_2 group, is the simplest. This is a group with only two elements, the identity and inversion.

C_2	e	A
e	e	A
A	A	e

Table 2.1: The C_2 group table, where entry i, j is the result of group operation on the i^{th} and j^{th} element.

As such, to find the group structured MDP homomorphism, the agent needs to learn an equivariant mapping that respects $\pi^s_1(s) = s$ and $\pi^s_A(s) = -s$, as well as $\pi^a_1(a) = a$ and $\pi^a_A(a) = 1 - a$.

Bibliography

[1] Richard Bellman. A markovian decision process. *Indiana Univ. Math. J.*, 6:679–684, 1957.

[2] Taco Cohen and Max Welling. Group equivariant convolutional networks. In *International conference on machine learning*, pages 2990–2999. PMLR, 2016.

[3] Gabriel Dulac-Arnold, Daniel Mankowitz, and Todd Hester. Challenges of real-world reinforcement learning. *arXiv preprint arXiv:1904.12901*, 2019.

[4] M Gell-Mann. The eightfold way: A theory of strong interaction symmetry. 3 1961.

[5] Danijar Hafner, Timothy Lillicrap, Mohammad Norouzi, and Jimmy Ba. Mastering atari with discrete world models. *arXiv preprint arXiv:2010.02193*, 2020.

[6] Danijar Hafner, Jurgis Pasukonis, Jimmy Ba, and Timothy Lillicrap. Mastering diverse domains through world models. *arXiv preprint arXiv:2301.04104*, 2023.

[7] Peter Henderson, Riashat Islam, Philip Bachman, Joelle Pineau, Doina Precup, and David Meger. Deep reinforcement learning that matters. In *Proceedings of the AAAI conference on artificial intelligence*, volume 32, 2018.

[8] Ronald A. Howard. Dynamic programming and markov processes. 1960.

[9] Misha Laskin, Kimin Lee, Adam Stooke, Lerrel Pinto, Pieter Abbeel, and Aravind Srinivas. Reinforcement learning with augmented data. *Advances in neural information processing systems*, 33:19884–19895, 2020.

[10] Y. LeCun, B. Boser, J. S. Denker, D. Henderson, R. E. Howard, W. Hubbard, and L. D. Jackel. Backpropagation applied to handwritten zip code recognition. *Neural Computation*, 1(4):541–551, 1989.

[11] Timothy P Lillicrap, Jonathan J Hunt, Alexander Pritzel, Nicolas Heess, Tom Erez, Yuval Tassa, David Silver, and Daan Wierstra. Continuous control with deep reinforcement learning. *arXiv preprint arXiv:1509.02971*, 2015.

[12] Yijiong Lin, Jiancong Huang, Matthieu Zimmer, Yisheng Guan, Juan Rojas, and Paul Weng. Invariant transform experience replay: Data augmentation for deep reinforcement learning. *IEEE Robotics and Automation Letters*, 5(4):6615–6622, 2020.

[13] Volodymyr Mnih, Adrià Puigdomènech Badia, Mehdi Mirza, Alex Graves, Timothy P. Lillicrap, Tim Harley, David Silver, and Koray Kavukcuoglu. Asynchronous methods for deep reinforcement learning. 2016.

[14] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. Playing atari with deep reinforcement learning. *arXiv preprint arXiv:1312.5602*, 2013.

[15] Arnab Kumar Mondal, Pratheeksha Nair, and Kaleem Siddiqi. Group equivariant deep reinforcement learning. *arXiv preprint arXiv:2007.03437*, 2020.

[16] Balaraman Ravindran. Smdp homomorphisms: An algebraic approach to abstraction in semi markov decision processes. 2003.

[17] Balaraman Ravindran and Andrew G. Barto. Symmetries and model minimization in markov decision processes. 2001.

[18] John Schulman, Sergey Levine, Pieter Abbeel, Michael Jordan, and Philipp Moritz. Trust region policy optimization. In Francis Bach and David Blei, editors, *Proceedings of the 32nd International Conference on Machine Learning*, volume 37 of *Proceedings of Machine Learning Research*, pages 1889–1897, Lille, France, 07–09 Jul 2015. PMLR.

[19] John Schulman, Philipp Moritz, Sergey Levine, Michael Jordan, and Pieter Abbeel. High-dimensional continuous control using generalized advantage estimation, 2015.

[20] John Schulman, Philipp Moritz, Sergey Levine, Michael Jordan, and Pieter Abbeel. High-dimensional continuous control using generalized advantage estimation. *arXiv preprint arXiv:1506.02438*, 2015.

[21] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms. *arXiv preprint arXiv:1707.06347*, 2017.

[22] David Silver, Aja Huang, Christopher J. Maddison, Arthur Guez, Laurent Sifre, George van den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, Sander Dieleman, Dominik Grewe, John Nham, Nal Kalchbrenner, Ilya Sutskever, Timothy Lillicrap, Madeleine Leach, Koray Kavukcuoglu, Thore Graepel, and Demis Hassabis. Mastering the game of go with deep neural networks and tree search. *Nature*, 529:484–503, 2016.

[23] David Silver, Thomas Hubert, Julian Schrittwieser, Ioannis Antonoglou, Matthew Lai, Arthur Guez, Marc Lanctot, Laurent Sifre, Dharshan Kumaran, Thore Graepel, et al. Mastering chess and shogi by self-play with a general reinforcement learning algorithm. *arXiv preprint arXiv:1712.01815*, 2017.

[24] Richard S Sutton and Andrew G Barto. *Reinforcement learning: An introduction*. 2018.

[25] Elise van der Pol, Daniel E. Worrall, Herke van Hoof, Frans A. Oliehoek, and Max Welling. Mdp homomorphic networks: Group symmetries in reinforcement learning. 2020.

[26] Hado Van Hasselt, Arthur Guez, and David Silver. Deep reinforcement learning with double q-learning. In *Proceedings of the AAAI conference on artificial intelligence*, volume 30, 2016.

[27] Dian Wang, Robin Walters, and Robert Platt. so2-equivariant reinforcement learning. *arXiv preprint arXiv:2203.04439*, 2022.

[28] Ronald J Williams. Simple statistical gradient-following algorithms for connectionist reinforcement learning. *Reinforcement learning*, pages 5–32, 1992.