Chapter 2

Background

2.1 Reinforcement Learning

In Reinforcement Learning (RL) we refer the model being trained as an agent. The agent interacts with an environment that is often stochastic and receives a reward signal based on the actions it takes. For example in the game breakout, the agent receives a positive reward of 1 for each block that is broken. The goal of the agent is to maximise the rewards it receives over the course of a full run through the environment that it interacts with, which is called an episode. Agents are trained for many episodes in order to learn a policy that maximises reward, A policy maps the current observable state of the agent and the environment to an action or a distribution over possible actions. Depending on the environment, an action can take a discrete or continuous value. For example, the agent in the breakout environment has a discrete action space which consists of a few options: move left, move right, or don't move at all. In contrast, the action space of an agent in control of a car on a road may be continuous: how much to turn the wheel, how much pressure to apply on the acceleration or brakes, etc. In some scenarios there may be more than one agent in the environment at a given time, which is called Multi Agent Reinforcement Learning (MARL).

2.2 Deep Reinforcement Learning

Deep Reinforcement Learning is the use of deep neural networks with reinforcement learning algorithms.

2.2.1 Deep Q-Networks

One of the first successful applications of Deep RL was the Deep Q-Network (DQN) agent, which displayed human level performance in many Atari games by combining the Q-Learning RL algorithm with a deep neural network [Mni15]. In Q-Learning, an agent learns to approximate the Q-Function, which maps an action to the an estimate of the total future reward the agent would obtain by taking that action at the current time step - assuming that it acted optimally from that point onward. The action-value function is given by the following equation:

$$Q^*(s,a) = \max_{\pi} \mathbb{E}[r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + \ldots | s_t = s, a_t = a, \pi] \tag{2.1}$$

The action-value function has an optimality property that allows it to be written in terms of itself in the form of a Bellman Equation:

$$Q^*(s,a) = \mathbb{E}_{s'}[r + \gamma \max_{q'} Q^*(s',a') | s,a] \tag{2.2}$$

This is because of Bellman's principle of optimality which states that for an optimal policy, given an initial state and decision, the remaining decisions must also form an optimal policy. This representation of the Q-Function gives rise to an iterative update rule that converges on the optimal action-value function Q^*.

$$Q_{i+1}(s,a) = \mathbb{E}_{s'}[r + \gamma \max_{a'} Q_i(s',a')] \tag{2.3}$$

In the case of Deep Q-Learning, a neural network is used to approximate the action-value function. We write this as a Q-Network $Q(s,a;\theta)$, where θ is the networks weights. At each step θ is updated with the aim of converging on a good approximation of Q^* by reducing the mean squared error between the LHS and RHS of equation 2.3. Where the target $y = r + \gamma \max_{a'} Q_i(s',a')$ is substituted with the approximation $y = r + \gamma \max_{a'} Q_i(s',a';\theta_i^-)$. In the target approximation, an older version of the parameters θ is used, which when implementing DQN means that there are two copies of the network stored, one "live" version and one that lags behind by some number of steps. Reducing the mean squared error leads to a sequence of loss functions at each time step which can be differentiated with respect to the network weights:

$$L(\theta_i) = \mathbb{E}_{s,a,r}[(\mathbb{E}_{s'}[y|s,a] - Q(s,a;\theta_i)^2)] \tag{2.4}$$

$$\nabla_{\theta_i} L(\theta_i) = \mathbb{E}_{a,r,s'}\left[\left(r + \gamma \max_{a'} Q(s',a';\theta_i^-) - Q(s,a;\theta_i)\right) \nabla_{\theta_i} Q(s,a;\theta_i)\right] \tag{2.5}$$

These equations can be used with stochastic gradient descent (SGD) optimisation to learn the Q-function and therefore the optimal policy by picking the action with the largest Q-value. The algorithm used for the DQN implementation in this project is the double-DQN variant:

Algorithm 1: Double DQN Algorithm.

input : $\mathcal{D}$ – empty replay buffer; θ – initial network parameters, θ^- – copy of θ
input : N_r – replay buffer maximum size; N_b – training batch size; N^- – target network replacement freq.
for *episode* $e \in \{1, 2, \ldots, M\}$ **do**
  Initialize frame sequence $\mathbf{x} \leftarrow ()$;
  for $t \in \{0, 1, \ldots\}$ **do**
    Set state $s \leftarrow \mathbf{x}$, sample action $a \sim \pi_{\mathcal{B}}$;
    Sample next frame x^t from environment $\mathcal{E}$ given (s,a) and receive reward r, and append x^t to $\mathbf{x}$;
    if $|\mathbf{x}| > N_f$ **then** delete oldest frame $x_{t_{min}}$ from $\mathbf{x}$ **end** ;
    Set $s' \leftarrow \mathbf{x}$, and add transition tuple (s,a,r,s') to $\mathcal{D}$, replacing the oldest tuple if $|\mathcal{D}| \geq N_r$;
    Sample a minibatch of N_b tuples $(s,a,r,s') \sim \text{Unif}(\mathcal{D})$;
    Construct target values, one for each of the N_b tuples: ;
    Define $a^{\max}(s';\theta) = \text{argmax}_{a'} Q(s',a';\theta)$;
    $y_j = \begin{cases} r & \text{if } s' \text{ is terminal} \\ r + \gamma Q(s', a^{\max}(s';\theta);\theta^-), & \text{otherwise.} \end{cases}$;
    Do a gradient descent step with loss $\|y_j - Q(s,a;\theta)\|^2$;
    Replace target parameters $\theta^- \leftarrow \theta$ every N^- steps;

Experience Replay

Experience replay is a technique used in DQN and many other reinforcment learning algorithms. At a step in the training of an RL agent, the experience that the agent goes through

can be captured by the current state, the action the agent takes, the reward it receives, and the state it progresses to after. These experiences can be stored over the course of the traning and "replayed" to the agent and learnt from multiple times. When updating the Q-Network weights in DQN, a batch of experiences is randomly sampled from the replay memory and used in the gradient descent process. One benefit of this is that it restores the independence assumption made by stochastic gradient descent. SGD assumes that the samples in a batch are independent and identically distributed. In reinforcement learning, subsequent states and states close to each other in time are often strongly correlated, which breaks this assumption. By sampling randomly across experiences from the beginning of training the independence assumption is restored. Also, data efficiency is improved by using experience replay because each experience can be used more than once.

2.2.2 Policy Gradients

As described before, DQN aims to estimate the value of taking an action, and this allows us to derive an optimal policy from the Q-Values by taking the action that corresponds to the highest value. Another type of reinforcement learning algorithm finds the policy directly by taking as input the environment state and producing as output a probability distribution over the action space. The policy is learnt by repeatedly updating the policy parameters in proportion to the gradient of the expected return with respect to the policy parameters. To train an agent using a policy gradient method the basic idea is to act according to the policy output distribution for a number of episodes or "rollouts", and afterwards adjust the weights in the network such that actions that led to rewards become more likely, and actions that led to negative reward become less likely. Updates to the policy need to be by the right amount, too little and learning is very slow, and too much and the policy diverges as the action probabilities are pushed too far towards the extremes (0 and 1).

Vanilla Policy Optimization

Policy Gradient methods look to maximise return by updating policy parameters using gradient ascent on policy performance: $\theta_{t+1} = \theta_t + \alpha \nabla_\theta J(\pi_{\theta_t})$. The policy gradient can be found by approximating the advantage function, which describes how much better or worse an action is on average compared to other action in a state. Using the advantage estimate $\hat{A}_t$, we can estimate the policy gradient as

$$\nabla_\theta J(\pi_\theta) = \hat{\mathbb{E}}_t \left[\log \pi_\theta(a_t|s_t) \hat{A}_t \right] \tag{2.6}$$

Which is computed by taking an average over multiple rollouts of the current policy. Using the update rule the policy can then be optimised on performance to maximise expected reward. VPG makes many small steps in parameter space to improve policy performance, but suffers from poor sample efficiency for this reason. Even small differences in parameters can have a large impact on policy performance, so taking a large step in the wrong direction can collapse the policy performance.

Trust Region Policy Optimization (TPRO)

TPRO builds upon VPG by constraining policy updates to making small changes on the action distributions using the KL-Divergence between the new and old policies. The KL-Divergence measures the similarity between probability distributions and has the follow-

ing formula [SLA+15]:

$$D_{KL}(p||q) = \sum_{x \in \mathcal{X}} P(x) log \left(\frac{P(x)}{Q(x)} \right) \tag{2.7}$$

TPRO updates the policy parameters according to the surrogate advantage, which measures how the new policy performs relative to the previous one:

$$\begin{aligned} \theta_{t+1} &= \underset{\theta}{\operatorname{argmax}} \mathcal{L}(\theta_{\text{old}}, \theta) \\ &\text{s.t.} \bar{D}_{KL}(\theta||\theta_{\text{old}}) \end{aligned} \tag{2.8}$$

And the surrogate advantage is defined as:

$$\mathcal{L}(\theta_{\text{old}}, \theta) = \hat{\mathbb{E}}_t \left[\frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{\text{old}}}(a_t|s_t)} \hat{A}_t \right] \tag{2.9}$$

An opimal policy can then be found by maximising the surrogate advantage function. The TPRO optimisation can be formulated with the divergence as either a constraint or a penalty, with the former being more widely used. The idea behind TPRO is that the KL divergence limits the size of the updates to the policy parameters, while maintaining monotonic improvement over policy iteration [SLA+15].

Proximal Policy Optimization (PPO)

Proximal Policy Optimization (PPO) improves on TPRO by simplifying the surrogate objective function $\mathcal{L}(\theta, \theta_{\text{old}})$. While in TRPO the size of the policy update is constrained using the KL-Divergence between the old and new policy distributions, PPO simply clips the ratio between the policies, thereby preventing large updates that could would normally lead to instability. The new surrogate objective is [SWD+17]:

$$\begin{aligned} \mathcal{L}(\theta_{\text{old}}, \theta) = \hat{\mathbb{E}}_t \left[\min(r_t(\theta)\hat{A}_t, \text{clip}(r_t(\theta), 1-\epsilon, 1+\epsilon)\hat{A}_t) \right] \\ \text{where } r_t(\theta) = \frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{\text{old}}}(a_t|s_t)} \end{aligned} \tag{2.10}$$

Clipping the ratio between $1+\epsilon$ and $1-\epsilon$ ensures that the new policy does not diverge from the old policy too much in a single step by performing a trust region update. In practise PPO is simpler than TPRO because of the removal of the KL-Divergence and performs especially well on continuous control tasks [SWD+17].

Algorithm 2: PPO, Actor-Critic Style [SWD+17]

```
for iteration = {1, 2, ...} do
    for actor = {1, 2, ...} do
        Run policy π_θold for T timesteps
        Compute advantage estimates Â_1, ..., Â_T
    Optimise surrogate L wrt θ, with K epochs and minibatch size MNT
    θ_old ← θ
```

Actor Critic

In the policy gradient methods described above, it is often necessary to estimate the value function. In these cases the value function can be learnt as well as the policy,

and used in policy update steps to reduce gradient variance. This is commonly known as a Actor Critic architecture, where the Actor learns the policy and the Critic learns a value function [SB14]. By learning the state-value function $V(S_t)$ and the relationship between the action-value function and state-value function defined by the Bellman optimality equation, we can calculate an estimate of the advantage value of an action $A(s_t, a_t) = r_t + \gamma V_v(s_{t+1}) - V_v(s_t)$. Equivalently, we can learn the action-value function $Q(s_t, a_t)$ and use the relationship $V(s) = \sum_a \pi(a|s)Q(s, a)$ to calculate $A(a_t, s_t)$ [SB14]. The PPO implementation used in this project uses an Actor Critic architecture where the state-value function is learnt by the critic network to estimate advantage values by a form of Generalised Advantage Estimation.

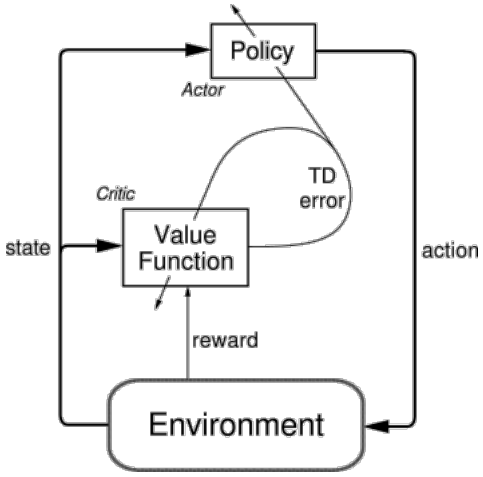

Figure 2.1: Actor Critic Architecture [SB14]

Soft Actor Critic (SAC)

Soft Actor Critic (SAC) is another policy gradient algorithm that employs the actor critic architecture. SAC is an off-policy algorithm, which means that previously collected samples can be reused. To encourage exploration by rewarding randomness in the policy, SAC includes a measure of the policies entropy into the reward. By encouraging exploration, learning is accelerated as the state space is explored more quickly, and the policy is less susceptible to falling into sub optimal local minima early in the training. For a random variable X with a probability mass function $P(X)$, the entropy $H(X)$ is defined as:

$$H(X) = \mathbb{E}\left[-\log(P(X))\right] \tag{2.11}$$

Using SAC, the policy is trained to maximise the both the reward and the policy entropy [HZAL18]:

$$J(\theta) = \sum_{t=1}^{T} \mathbb{E}(s_t, a_t) \sim \rho_{\pi_\theta}\left[r(s_t, a_t) + \alpha\mathcal{H}(\pi_\theta(\cdot|s_t))\right] \tag{2.12}$$

In this equation, α is the temperature parameter, which controls how important entropy is in the objective. The first version of the SAC algorithm learns three functions simultaneously, the policy, a soft Q-Value function Q_w and soft state-value function V_ψ [HZAL18]. The version implemented in this project is the modified version that instead learns the policy and two Q-Functions Q_{ϕ_1} and Q_{ϕ_2} and uses the clipped Double-Q trick.

Algorithm 3: Double-Q SAC Algorithm [DGL+20]

Initialize parameters θ_1, θ_2, ϕ_1, ϕ_2, and α;
Initialize target parameters $\theta'_1 \leftarrow \theta_1$, $\theta'_2 \leftarrow \theta_2$, $\phi'_1 \leftarrow \phi_1$, $\phi'_2 \leftarrow \phi_2$;
Initialize learning rate β_Q, β_π, β_α and τ;
Initialize iteration index $k = 0$;
repeat
  Select action $a \sim \pi_{\phi_1}(a|s)$;
  Observe reward r and new state s';
  Store transition tuple (s, a, r, s') in buffer $\mathcal{B}$;
  Sample N transitions (s, a, r, s') from $\mathcal{B}$;
  Update soft Q-function $\theta_i \leftarrow \theta_i - \beta_Q \nabla_{\theta_i} J_Q(\theta_i)$ for $i \in \{1, 2\}$;
  if k *mod* m **then**
    Update policy $\phi_i \leftarrow \phi_i + \beta_\pi \nabla_{\phi_i} J_\pi(\phi_i)$ for $i \in \{1, 2\}$;
    Adjust temperature $\alpha \leftarrow \alpha - \beta_\alpha \nabla_\alpha J(\alpha)$;
    Update target networks:
      $\theta'_i \leftarrow \tau\theta_i + (1 - \tau)\theta'_i$ for $i \in \{1, 2\}$;
      $\phi'_i \leftarrow \tau\phi_i + (1 - \tau)\phi'_i$ for $i \in \{1, 2\}$;
  $k = k + 1$
until *Convergence*;

2.3 SMARTS

SMARTS (Scalable Multi Agent Reinforcement Learning Training School) is a simulation platform designed for the training of reinforcement learning agents with a focus on the domain of autonomous driving. SMARTS provides a simulation environment, and on top of this abstractions that allow complex vehicle scenarios to simulated, such as background vehicles, traffic maps, and realistic action spaces for continuous or discrete control over vehicles. SMARTS is designed with multi agent reinforcement learning in mind with the ability to train multiple ego agents in the same environment simultaneously, which makes it an ideal test bed on which to investigate the performance of agent training as the number of trained agents is increased.

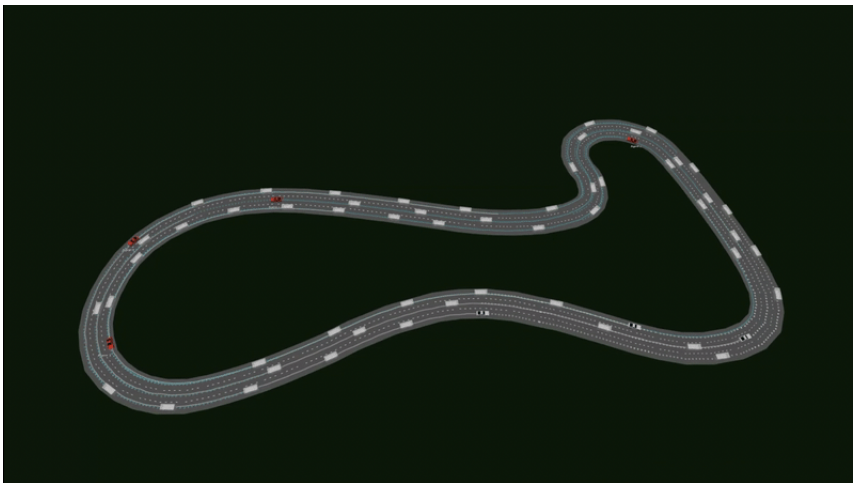

Figure 2.2: SMARTS simulator screenshot of the "loop" environment. The "ego agents" being trained are highlighted in red and the "social agents" that make up the background traffic are in grey.

SMARTS also provides automatic management of background or "social" vehicles that

Observation Type	Description
events	Simple information about the vehicle such if it has reached its goal or not, or if it has been involved in any collisions
ego vehicle state	More detailed scalar information describing the ego vehicle, such as its position, speed, acceleration, steering etc.
neighborhood vehicle states	List of the states of vehicles in the neighborhood of the ego vehicle, in the same format as the ego vehicle state
top down RGB	An RGB image of the area surrounding the ego vehicle seen from above, with the ego vehicle at the center
occupancy grid map	A grid the of area around the ego vehicle with binary occupancy data
waypoint paths	A list of waypoints in front of the ego vehicle showing potential routes ahead. The waypoint paths can be seen as the blue bots in front of the ego agents in figure

Table 2.1: Observations available in SMARTS [**?**]

are not being actively trained but can still exhibit complex behaviours. The Social Agent Zoo is a repository that stores pre-trained agents that can be added to a map alongside the ego agents being trained with the aim of facilitating more complex interactions for the ego agent so that it can learn more robust policies as it experiences more diverse interactions with other vehicles on the road.

2.3.1 Agent Types

SMARTS offers multiple different action and observation spaces that can be fined tuned to the task requirements.

Action Types

There are three action space types: Continuous, Lane, and ActuatorDynamic. Agents using the Continuous action space type have continuous control over the amount of brake, throttle, and the degree of steering. The Lane action type is a discrete action space in which the agent can choose one of four actions: "keep_lane", "slow_down", "change_lane_left", and "change_lane_right". For agents using this action type, the vehicle speed and steering in managed automatically by SMARTS according to the agent action. For example if a Lane type agent was returning the action "keep_lane" whilst going round a corner, SMARTS would automatically steer the agent round the corner to keep it in its lane. The ActuatorDynamic action type is the same as Continuous except the agent must specify a steering rate of change in degrees per second instead of simply the steering angle [**?**].

Observation Types

SMARTS implements the typical types of observations that an autonomous driving agent might require. Table 2.1 shows a breakdown of the available observations. Agents in SMARTS must implement the AgentInterface in which the observations that should be available to the agent are defined.

2.3.2 Environment

SMARTS implements an environment for autonomous driving training that is compatible with the OpenAI Gym API. There are several maps for the environment such as the

loop map as well as intersections, roundabouts, and intersections that SMARTS calls "scenarios". Figure. Traffic simulation and maps are provided by SUMO (Simulation of Urban MObility) [**?**] and users can create their own maps using SUMOs netedit software. Scenarios can be configured to have certain patterns of background traffic, and the vehicle types included in the background traffic can also be configured. To aid scalability, SMARTS uses the concept of "bubbles" in which complex interactions happen. A bubble is a region on a map, inside the bubble the background traffic is controlled by the agents from the Social Agent Zoo, and outside the bubble the background traffic is controlled by the background traffic provider which is SUMO by default. Figure 2.3 shows the bubble mechanism. In the A zone the social vehicles shown in beige are controlled by the background traffic provider. Social vehicles in the blue transition zone are in the process of handing over control to an agent from the social agent zoo. The yellow vehicles in the red zone are in now being controlled by the social agent.

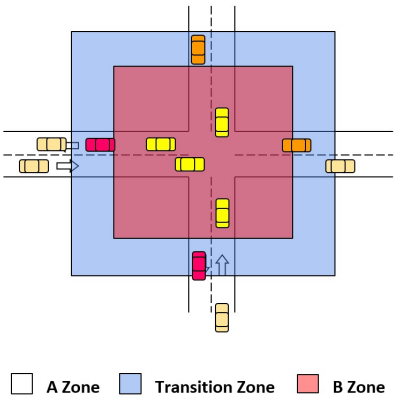

Figure 2.3: SMARTS bubble mechanism.