1st literature review

The thesis project will begin with a literature review. This review will begin by focusing on literature, mainly MSc theses, produced by the many students that Erik-jan has supervised, in addition to the introduction to reinforcement learning textbook by Barto & Sutton. The list of literature reviewed in this batch are as follows:

1. W.J.E. Völker: Reinforcement Learning for Flight Control of the Flying V
2. Thomas v.d. Laar: Deep Reinforcement Learning for Aircraft Landing
3. Lucas Vieira: Safe & Intelligent Control.
4. Peter Seres: Distributional Reinforcement Learning for Flight Control
5. Zhou Xin Ge: End-to-End Hierarchical Reinforcement Learning for Adaptive Flight Control
6. Ramesh Konatala: Design of Reinforcement Learning based Incremental Flight Control Laws for the Cessna Citation II(PH-LAB) Aircraft
7. Stefan Heyer: Reinforcement Learning for Flight Control
8. Jonathon Hoogvliet: Hierarchical Reinforcement Learning for Model-Free Flight Control
9. Barto & Sutton: Reinforcement learning, an introduction

Item 1 is for getting some exposure to rl controllers used in the context of flight control, specifically that of the flying-V which is main focus of the thesis. This item is reviewed first because of the fact that it presents the cross section of rl for flight control and the flying v, which should provide a good overview of both topics. Then items 2 till 8 are all past master theses whos' topic were in reinforcement learning for flight control and supervised by Erik-Jan van Kampen, a summary of each is available in [[Previous_RL_MSc_theses]]. Item 9 is for becoming proficient in the theoretical basis of reinforcement learning.

Item 1: Reinforcement Learning for Flight Control of the Flying V

Questions & Answers-

From chapter 2:

1. What is the history and the derivation of the Bellman equation?
   1. The Bellman equation does not necessarily refer to a single equation. The equation most colloquially called the Bellman equation is the Bellman equation of the Value function, which is simply **the** definition of the value function for a given state. It returns the same value as the value function, so in the sense of what it computes the Bellman equation of a value function is equal to the value function. But the equation itself allows you to compute the value function, unlike the equation for the value function which is simply an expectation operator taken on the return of a state.
   2. The bellman equation for value function can be derived by starting from the expected value definition of the value function, expanding the return term by using it's recursive definition being the sum of current reward plus future returns, then expanding the expectation operator using the definition of the operator for a discrete random variable. This derivation is shown in figure 1 from. ![[../pictures/bellman_derive.png]]

      Figure 1, derivation of the bellman equation.

2. What is the distinction between the weight vector **w** and parameter vector **theta**??
    1. **w** is the weight vector that parametrizes a function approximator for the state or action value functions, typically the approximator is something like an ANN hence usage of the term "weight vector". **theta** is the parameter vector for a policy, which does not necessarily or usually mean an ANN, could also just be some function. But from reading other theses, it seems that generally both the policy and value functions can take an ANN form, thus the difference in naming is only conventional.
3. Willem states an interesting distinction between actor and critics, "The actor network then uses gradients to update the policy parameter vector θ and the critic network the value-function parameter vector w". So the actor only updates its policy, and the critic only updates its value function?
    1. Turns out yes, an actors job is to provide the action that an agent takes, and the critics job is to provide a value function estimate that can be used to "gauge how adequate" the actions are.

From chapter 4:

1. Nasa has a thing called the "Common Research Model", what is this?
    1. The NASA CRM is a generic aircraft design model [1], for which an extensive open-source aerodynamic database has been gathered through several wind tunnel tests of the aircraft model. The NASA CRM can almost be thought of as another aircraft type, much like the Boeing 747 or the Airbus A380. It is a wide-body commercial transport aircraft with a supercritical transonic wing, designed for a cruise Mach number of M_infty = 0.85.
2. Pitch break, what is that?
    1. It's the onset/presence of longitudinal static instability in an aircraft, commonly happening at high angles of attack, where the C_M_alpha of the aircraft becomes positive (hence static instability).
3. Is there a model of the flying-V which models its' pitch-break behaviour?
    1. Reading Willems' thesis, a model is not made for that level of alpha since there is only a limited fidelity of control surface effectiveness in the part of the flight envelope where pitch break occurs (above 15 degrees alpha), making a model for that region not useful.

From chapter 6:

1. Figures 6.1 & 6.2 have a subfigure graphing the actuation of in/outboard elevons. These are confusing figures though because it shows that the deflections of in/out board elevons are opposite!?

Notes-

From chapter 2:

- *Generalized policy iteration* (GPI) is a dynamic programming approach to finding an optimal policy and the true value function "simultaneously", which is achieved by alternating between performing *policy evaluation* and *policy improvement*. Policy evaluation is done to improve the approximation of the value function, which is done by simply computing the bellman equation for value function, Eq 2.10 from Willem. Policy improvement is done to improve the policy itself, which is done by setting the policy to be equal to the action (which remember is a probability distribution!) which has the highest expected return, Eq 2.13 from Willem. GPI is a useful theoretical foundation for more advanced techniques such as *heuristic dynamic programming* (HDP), but GPI in its pure analytical form is too computationally expensive to compute/reach convergence especially for as complex a MDP as flight control, and require complete models of transition probabilities (i.e. the complete dynamics of the MDP).

- *Monte Carlo* (MC) methods are a class of methods that learn a value function by repeatedly and randomly the returns from visiting a given state, in contrast to GPI which learn the value function from analytically evaluation the expected returns as well as the bellman equation for value function. These methods do not *bootstrap*, as they do not base one value function estimate on another.
- *Temporal difference* (TD) methods combine the ideas of GPI (or dynamic programming) and MC methods, and updates an estimate of the value function (or estimate of any variable) through a finite difference scheme of x_1 = x_0 + dt(dx). (p.s, found interesting paper from the dawn of time that maybe i should read [2], he also introduces *n-step TD* with notation which i dont understand, what is G_{t:t+n}).
- Distinction between on and off policy learning: an RL agent in on policy learning optimizes the policy which it is using, while in off policy the agent optimizes a policy which it is not using. So for instance an offpolicy agent can use a pre-trained policy to explore a policy space and then use this exploration to improve a separate policy. Here a definition can be introduced, where the pre-trained policy is a "behaviour" policy, and the separate policy is the "estimation policy. Willem's thesis emphasizes on describing what off-policy is and its' implications. Where until recently off-policy algorithms have been largely too varied for large continuous state and action spaces.
- "small" literature review of deep learning available here
- Quote "(policy-based) methods have the important advantage over value-based methods that they can be applied to continuous action spaces". May need to verify
- Quote "An important practical advantage of using a parameterised policy is that it offers a way of adding prior knowledge about its desired form". (i think) This could mean for example favouring the roll clockwise when the airplane rolled anti clockwise.
- Willem asserts that the latest off policy, model free RL algos for continuous spaces include DDPG, TD3, and SAC.

From chapter 3:

- In this chapter Willem describes a lot of the high level decisions he made during the thesis, such as which algo to choose, how to engineer the rewards (cost function), and compute resource.
- Willem used cloud computing since he only had an 8gb ram laptop with no gpu.
- TD3 was chosen over SAC in the end, **a)** TD3 is deterministic (SAC is stochastic) so less dangerous for flying, **b)** "highly oscillatory actions" of SAC. But he notes that the differences between these two algos are otherwise small.

From chapter 4:

- 3 sources for a flying-V model are available. From which Willem chose model 1.
  1. model 1, simulation based on stability & control derivatives from a VLM based CFD simulation of the full scale flying V, (i assume is linear)
  2. model 2, simulation of model 1 improved with wind tunnel measurements
  3. model 3, simulation based on derivatives obtained from flying the RC-controlled 4.6% scale model flying-V.
- The thesis and some other materials refer to a "NASA common research model", but dont ever say even at the briefest mention what it is, so frustrating. Remember to document in the weekly report!!!!!
- J. Benad [3] from TU Berlin carried out the first preliminary design of the flying-V concept in collaboration with Airbus.
- Willem uses the FCS sized by Cappuyns [4].
- Willem cites from Dally [5] that "direct control inputs resulted in unstable behaviour during tests", wherein the RL agent's action signals only span [-1, 1] (square bracket is inclusive, round bracket is

exclusive) and the mapping from this normalized range to actuator action is one-hundreth of the actuator change increment. This actually raises a slightly interesting note, because clipping the gaussian to [-1,1] naively will lead to the values -1 and 1 to be sampled a bit higher than they should be. So instead, one should use the truncated normal distribution.
- The flying V's damping coefficient for short period, phugoid, and dutch roll would not meet regulatory requirements of level 1 handling quality *in some flight conditions and CG locations* without an SAS [6].
- Flying V has a pitch break behaviour where it becomes statically unstable (*longitudinally*) beyond certain alphas (~15) [6].

Item 1 References

[1] M Rivers. (2019). NASA Common Research Model: A History and Future Plans. *AIAA Aviation 2019 Forum*. doi: 10.2514/6.2019-2188

[2] A Samuel. (1959). Some studies in machine learning using the game of checkers. *IBM Journal on Research and Development*. url: https://api.semanticscholar.org/CorpusID:2126705

[3] J Benad. (2015). The Flying V - A new Aircraft Configuration for Commercial Passenger Transport. Deutscher Luft-undRaumfahrtkongress 015, Rostock. doi: 10.25967/370094

[4] T Cappuyns. (2019). Handling Qualities of a Flying V Configuration. TU Delft, http://resolver.tudelft.nl/uuid:69b56494-0731-487a-8e57-cec397452002

[5] K Dally. (2021). Deep Reinforcement Learning for Flight Control. TU Delft, http://resolver.tudelft.nl/uuid:fcef2325-4c90-4276-8bfc-1e230724c68a

[6] S van Overeem, X Wang, E-J van Kampen. (2022). Modelling and Handling Quality Assessment of the Flying-V Aircraft. *AIAA Scitech 2022 Forum*. doi: 10.2514/6.2022-1429

[7] M Palermo, R Vos. Experimental aerodynamic analysis of a 4.6%-scale flying-v subsonic transport. *AIAA Scitech 2020 Forum*. doi: 10.2514/6.2020-2228.

Item 3: Deep Reinforcement Learning for Aircraft Landing

Questions & Answers-

Notes-

From chapter 4:

- 5 DRL's are compared by training on the LunarLander OpenAI gym environment.
- PPO is claimed to be better than TRPO as it is simpler to implement, and have better sample "complexity".
- Two tests are designed to compare the 5 chosen DRL algorithms, **test 1:** training all DRLs in the nominal gym and then testing them on a windy environment, **test 2:** training all agents on a windy environment and then testing them on various windy environments.

From chapter 7:

- Using the repo He evaluated 5 DRL algos together, and concluded that SAC had the best perfromance in virtually all parameters he kept track of. This is in contrast to what Vlad and Willem concluded, that TD3 was better.
- A full 3d ALS model with PID and landing scenario was implemented.
- The implemented SAC showed unstable learning behaviour, with rewards dipping in "lower" stages of learning (what is lower?), and the trained agents showed oscillatory actuation usage, something that was remarked on by Vlad and Willem.

From Appendix A:

- Pseudocode of the 5 DRL's are written.

Item 4: Safe & Intelligent Control

Questions & Answers-

From chapter 3:

1. I cannot understand figure 3.9. He plots mean episode length over samples. What is it plotting?
    1. I think i figured it out. It shows that in general as more samples are taken, the episode length increases and plateaus at the maximum episode length, which is when the highest return can be obtained. I.e., there is a correlation between episode length and return, hence him saying that "...SAC and DSAC learn to converge to a maximum return..."
2. Lucas used multiple seeds to evaluate the performance of all 4 controllers, but is there any stochasticity when it comes to evaluating the offline-trained controllers?
    1. TBD...

Notes-

From article:

1. Both (D)SAC and IDHP use an actor-critic structure, however the critic network in IDHP is very different than the ones used by (D)SAC, wherein the network estimates a *derivative* of the value function, instead of the value function itself. Consequently the hybrid structure of this (D)SAC-IDHP controller is only hybrid as far as the actor networks are concerned, and *not the critic network*. In contrast, the critic in SAC is a Q-function critic

---

Item 5: Distributional Reinforcement Learning for Flight Control

Questions & Answers-

From article:

1. Casper makes the distinction between *agent state vector **s*** and *environment state vector **x***. But is he referring to the *observation vector **o*** when talking about ***s***? With ***o*** being the subset of ***x***. This is important because he says "the state vector used by the partial state derivatives and incremental model has to be explicitly defined as the *environment state vector **x*** in order for the incremental model to retain a meaningful estimation of the system dynamics."

1. TBD.

Notes-

From chapter 1:

- He defined high sample efficiency as needing "a low amount of data points during training in order to reach a certain level of tracking performance"
- Learning performance was defined as "high sample efficiency, stable learning process which is robust to changes in hyperparamater or observability"

From article:

- "RL-state and the action as inputs per definition of the Q-function with a single scalar output", implying that Q-function is defined as the action value function, need to confirm this implication with studying Q-functions!!
- SAC's actor netowork outputs Gaussians for each actions as a function of state/observations.

From chapter 3:

- Distributional RL can be distinguished from non distributional methods in 3 ways:
  1. They parameterize and use the return distributions, instead of only using expectation of a distribution
  2. The distance metric used (a quantile Huber loss in case of DSAC which used IQN) to optimize parameters
  3. Ability to avoid risk in both pre and post training
- DSAC, made by a team from [Tsinghhua] is an extension of SAC where it changes the critic network to a *distributional* critic network.

From chapter 4:

- Sampling efficiency is quantified by simply seeing how many episodes are needed to achieve a certain level of reward. Evidence for improved sampling efficiency of DSAC can be found in the research paper. From Figure 4.9 Peter claims that a 20% increase in sampling efficiency was achieved, but he was probably referring to Figure 4.7.
- A number of methods to reduce the oscillatory actuation from DSAC controllers are prsented by Peter:
  1. Increasing penalty of high actuation in the reward function
  2. Adding a CAPS (control action policy smoothness) loss term to the reward function, which penalizes control policies which are not smooth temporally and spatially (changes quickly over time or quickly between states)

From chapter 7:

- *Estimator windup* is when the covariance of the estimator in an RLS increases exponentially and is caused by a combination of factors, the combination of forgetting factor being set to < 1 and **a)** lack of system excitation or **b)** non-uniformly distributed information over all parameters **c)** time-scale separation in the variation of parameters. Consequence of windup is abrupt changes in estimate once the system becomes excited (according to Heyer et al.). And can be combatted by setting forgetting factor to 1.

Item 6: End-to-End Hierarchical Reinforcement Learning for Adaptive Flight Control

Questions & Answers-

From chapter 5:

1. What do the numbers in tables 5.6-5.9 mean?
    1. I think the fractions are supposed to indicate the rate of success.

Notes-

From chapter 1:

- The term "options" is introduced by Sutton in this publication. This term refers to the set of primitive and temporally extended actions. It is also supposedly a method of Hierarchical RL.

From chapter 3:

- A very nice descrciption of Actor-Critic design is given, paraphrasing: "The actor decides the control actions and is hence equal to a control law, while the critic learns a (state)-value function estimate to determine if executed action has added value and provides a TD error to update teh actor"

Item 7: Design of Reinforcement Learning based Incremental Flight Control Laws for the Cessna Citation II(PH-LAB) Aircraft

Questions & Answers-

1. Why is the acronym iADP instead of IADP (all caps)

From chapter 3:

1. Does he omit the discussion of IDHP? And why? 1.

From chapter 4:

1. It is implied that PE is only neccessary for the output feedback case of (i)ADP as Ramesh states that "... full state information is not available exploration of states is necessary", which has the implication that full state information can supplement using PE for exploration, but it appeared that according to Zhou PE is neccessary for ADP in general?
    1. PE is used as a system identification input, where all system states are excited so that the state space can be explored, so Zhou is more correct. And it feels like that Ramesh is misusing the term "PE", or at least not properly explaining it enough, because it seems that persistent excitation can come in more forms than only white noise.

Notes-

From article

- ADP methods require exploration of the states in order to learn the environment, which is done through Persistent Excitation.PE can be done by adding white noise to the action values to ensure that

the system is excited at all times. Fortunately, PE is only performed during the training stages of the algorithms, e.g. an iADP controller is trained with PE included to arrive at a set of incremental models ***F*** & ***G***, where after these models can be used for actual flight with no PE included in the iADP controller.
- It would be **very interesting** to look at comparison of iADP and IHDP from Zhou's dissertation, where she implements (and derives!) both algorithms.

From chapter 3:

- A nice -but hard to read- pseudo code of a basic acator-critic method is given in algorithm 1.
- *LADP* stands for linear approximate dynamic programming, which constantly learns a linear model of the system online. Here, the optimal control action to minimize an LQR cost is used to implement *GPI*. Where the state error cost matrix P is iterated representing policy evaluation, and the control action u is iterated representing policy improvement.
- *iADP* stands for incremental approximate dynamic programming, which is an incremental counterpart of LADP, and learns an incremental model, and uses a different form of the optimal control action which contains incremental models of system dynamics ***F*** & ***G*** instead of the linear ss models ***A*** & ***B***. Detailed derivations are publish by Zhou et al

From chapter 4:

- I can tell from fig 4.9 that Ramesh is the one who did his thesis using MATLAB.

Item 8: Reinforcement Learning for Flight Control

Questions & Answers-

Notes-

From chapter 1:

- He notes that ADP method's, despite being adaptive, still require offline training, touting it almost as an advantage of IHP. When in fact, an IDHP controller *also* should be trained offline first before its' first real flight, otherwise the controller dynamics will not be initialized, even if the IDHP agent was intialized with an offline trained actor like in the case of the hybrid controllers from Casper and Lucas.

From chapter 2:

- actor-critic designs (ACDs), introduced first by Sutton, fall under the umbrella of approximate dynamic programming (ADP). ACDs can be further grouped into heuristic dynamic programming (HDP), dual heuristic programming (DHP), and globalized dual heuristic programming (GDHP).

From chapter 9:

- Estimator wind up occurs when forgetting factor of RLS is < 1, Stefan avoids this by setting the forgetting factor = 1. But this has the disadvantage that the RLS does not forget old system models, and overtime the model parameters of RLS change less. This was directly addressed by setting the covariance matrx to its initial values when the estimators innovation sees large changes.

Item 9: Hierarchical Reinforcement Learning for Model-Free Flight Control

Questions & Answers-

From article:

1. Jonathon says that "episode lengths differ per hierarchical layer", and that "q-de controller is evaluated for 10.5 seconds, gamma-q for 94.1 seconds..." but how is that possible? Is the entire agent not trained or ran at once?
    1. He does not explain in his thesis, need to look for hints from other sources...

Notes-

From article:

- A sinusoidal training signal made agents during training more stochastic.
- It would seem that one way in which HRL can be used to address curse of dimensionality is to simply reduce the number of cross-relations which exists in a Q table. E.g., if an environment has 5 number of states and 1 action, the dimension of the full Q-table would be 7. But seperating the environment into subproblems can allow for the Q-table to be reformulated, where instead of a single 7 dimensional Q-table, you could instead have e.g. 5 two-dimensional Q-tables.
- An attempt of using a discrete Q-table to encapsulate the entire task of aircraft altitude control was made, where it was found that the sample efficiency of this approach is absolutely diabolical, requriing 120 million samples to reach a comparatively poor tracking performance.
- The time traces of all 3 controllers altitude tracking seems relatively poor, especially when compared to the likes of killian's controllers.

Item 10: Reinforcement learning, an introduction

From chapter 1-introduction:

Elements of reinforcement learning:

1. Policy
    - Defines the agent's way of behaving at a given state, can be a lookup table, or a continuous function, or even a *search process*. It is allowed to be stochastic, where actions are given probabilities of being executed, or deterministic, where a certain action is taken for certain states.
2. Reward signal
    - Defines the goal of the reinforcement learning problem, from an optimization standpoint a Reward signal is akin to the cost function. In every RL problem, an agent receives a reward signal in the form of a real number at each and everystep; it is the main and only ingredient in the value function.
3. Value function
    - Indicates what is the *long run* reward for being in a certain state or being in a certain state *and* taking a certain action, in the former case the value function is called *state value function*, the latter case leads to the *action value function*. It is the total or expected (in the probability sense)

reward that an agent can expect to obtain by being in a certain state/state-action pair. There exists distinct value functions for being in

4. Environment model (optional)
    - The mathematical models of the environment in which an agent is to act.

Reinforcement learning's river of history has 3 main streams, the first one is the *optimal control* stream, the second is the *trial-and-error learning* stream, and the third stream is the *temporal-difference* method. In the early stages, the field of optimal control was concerned with designing control laws which could minimize any certain measure of a dynamical system over time. A very famous progenator of this field is Richard Bellman, who along with others extended the theory of two mathematicians: Hamilton and Jacobi, who developed the theory which allowed development for equations to provided neccessary and sufficient conditions for the optimality of a control law. It was with Chris Watkins' work in 1989 that the streams of optimal control and trial-and-error learning combined to give rise to approaching the biological phenomenon of learning by modelling environments as MDP's and using optimal control theory to determine actions in such an environment, leading to the field of *reinforcement learning*, which addressed the curse of dimensionalty head on and attempted to control for relatively more complex system models than optimal control theory original addressed.

The essential character of trial-and-error learning as selecting actions on the basis of evaluative feedback that does not rely on knowledge of what the correct action should be.

From chapter 2- multi-armed bandits:

This chapter gives an introduction on the RL problem -specifically maximizing return- using the illustrative RL problem of the multi-armed bandit, a simple and non-associative class of problems.

> [!Definition]+ **Non-associative** Is an adjective used to describe how one variable does not depend on another, in contrast to associative where dependency may exist. Nonassociative tasks have no need to associate different actions with different situations, simply put there is only one situation and the agent just has to find the best action for that situation. For associative tasks, there is an associated *best action* for each situation (*state*).

Exploitation is to take the action which your current estimates say give the highest return or has the value, while exploration is to select one of the actions that is not estimated to give the highest return or value. By using an epsilon greedy method for selecting action, it can be shown that asymptotically the optimal action is chosen with a probability greater than 1-epislon, which for small epsilons will mean near certainty. This is ofcourse only the case in the limit, which practically is improbable to achieve.

> [!Definition]+ **Action-value function** A value function that estimates the return (value) of a taking a certain action given a certain state.

Estimating the return of an action can be trivially done by simply taking the realisation average, i.e. sum all rewards obtained from selecting an action, and then divide by the number of times that action was selected, which will yield an estimate of the actions' return. This method, however, is memory inefficient and can be memory intensive as the number of sample and actions grow. A smarter method of estimating return can be done by using an incremental formula, where the next return estimate is equal to the previous return estimate plus a factor of 1/n times the sampled reward minus previous return estimate:

$Q_{n+1} = Q_n + \frac{1}{n}(R_n - Q_n)$

This format of previous estimate plus a factor of difference in sample and estimate, is a frequently occurring expression in reinforcement learning, for example in the case of bootstrapped learning.

Two methods for selecting actions are thus far introduced. First method is to deterministically select action with highest return estimate Q(a); so called action-value methods. Second method is to prefer actions with higher value estimates through a softmax function of $H(a)$; so called gradient based action-value methods.

- equation 2.8-2.9 gives an interesting equation for having a moving average which is unbiased to initial conditions.

From chapter 3-finite Markov decision processes:

This chapter gives an introduction on (finite) MDP, the mathematical framework used to describe all RL problems. The finite MDP is evaluative just like the multi-armed bandit problems where actions have rewards, but are also associative as there exists an optimal action for each situation (state). The important distinction which finite MDPs bring over multi-armed bandits is the temporally related nature of rewards, where the problem of exploitation vs exploration goes beyond estimating the optimal action for the initial state, but includes estimate the optimal action**s** which will bring the agent to future states and the highest long term return. Formally speaking, the bandit problem only required estimating an action value function that is purely a function of action $q*(a)$, and in the finite MDP case it is necessary to estimate an action value function that is also a function of **states** $q*(s,a)$, or equivalently an estimation of the optimal state value function $v*(s)$.

MDPs take on the general form as shown in figure 1, the entire system in an MDP is described by a set of 5 variables, the agent, the environment, and 3 time varying signals of action, state, and reward. An agent is defined typically by its policy and value functions, and the environment is defined by the dynamics functions which prescribes some state and reward given an action. MDPs suppose that any decision making process can be reduced to simply this set of five variables, and has proved to be a useful abstraction of reality in reinforcement learning. It is similar to the state space models of systems found in control theory, which has 3 signals: state, control input, and measured output, as well as 4 matrices which describe the state transition function, control input function, as well as the observation functions.

![[mdp.png]]

> **Figure 2**, fundamental dynamics of an MDP, which is defined fully by all the signals and blocks shown here (agent, environment, reward, action, reward, and state).

> **Environment**
>
> It is completely characterized by the probabilities given by a discrete (or continuous) probability density function $p(s',r|s,a)$, i.e. the probability of next state=s' next reward=r given that the current state=s and the agent's action=a, p's functional mapping is thus p: S x R x S x A -> [0,1].
>
> > [!Definition]+ **Dynamics of MDP** $p(s',r|s,a) \doteq Pr{S_t = s', R_t = r | S_{t-1} = s, a_{t-1} = a}$
>
> By summing/integrating this probability density function over all possible rewards, it is possible to obtain a marginal probability density function which is called the state transition probability function in MDPs $p(s'|s,a) = \sum_{r\in R}p(s',r|s,a)$, which describes the probability of the next state=s' given the environments' previous state=s and the agent took action=a.

> The expected reward $r(s,a)$ of a given s,a pair can be obtained by taking the expectation of $p(s',r|s,a)$, $r(s,a) = \sum_{r \in R} r \sum_{s' \in S}p(s',r|s,a)$.

> **Reward**
>
> When specifying the reward in an environment, it is important to remember to only reward an agent once the end goal is achieved, not once some sub goals are met. As it could be possible that achieving sub-goals and not the final goal provide the highest rewards, leading to an agent not learning to achieve the final goal.

> **Return**
>
> Return G_t is defined as the sum of future rewards, $G_t = R_{t+1} + R_{t+2} + ... + R_{T}$, where T is the terminal time step. The general definition of G_t incorporates the idea of *discounting*, where rewards further in the future are *discounted* more and prevents rewards in continuing tasks from diverging:
>
>> [!Definition]+ **Return** G_t: $G_{t} \doteq R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + ... + \gamma^{T-t-1}R_{T} = \sum_\limits{k=0}^{T}\gamma^{k} R_{k + t + 1}$ where $0 \leq \gamma \leq 1$ is called the discount rate.
>
> It is useful to compactly rewrite the return as a recursive function: $G_t = R_{t+1} + \gamma G_{t+1}$ Th To apply this definition of return to both episodic and continuing tasks, episodic tasks are defined to terminate with an absorbing state where the probability of transitioning out of the absorbing state is zero and its' reward is 0. This allows both type of tasks to use the $\lim\limits_{T \to \infty}$ definition of return.

> **Value function**
>
> It is the expected return for the environment being in a certain state, and is written as a function $v_{\pi}(s)$ which denotes the value v of being in a state s and following the policy π
>
>> [!Definition]+ **Value function** $v_{\pi}(s)$: $v_{\pi}(s) \doteq \mathbb{E}[G_t | S_t = s] = \mathbb{E}[R_{t+1} + \gamma G_{t+1} | S_t = s]$

> **Action-value function**
>
> It is the expected return if an agent takes a certain action when the environment is in a certain state, and is written as a function $v_{\pi}(s,a)$ which denotes the value v of taking the action a while being in a state s and following the policy π
>
>> [!Definition]+ **Action-value function** $q_{\pi}(s)$: $q_{\pi}(s) \doteq \mathbb{E}[G_t | S_t = s, a_t = a] = \mathbb{E}[R_{t+1} + \gamma G_{t+1} | S_t = s, a_t = a]$

By expanding the expectation operator of either value functions and re-expressing the functions into a recursive formula, you arrive at what are known as the *Bellman Equations*;

>> [!Definition]+ **Bellman equation for $v_{\pi}(s)$** $v_{\pi}(s) = \sum\limits_{a}\pi(a|s)\sum\limits_{s',r}p(s',r|s,a)[r + \gamma v_{\pi}(s')]$
>
>> [!Definition]+ **Bellman equation for $q_{\pi}(s)$** $q_{\pi}(s,a) = \sum\limits_{s',r}p(s',r|s,a)[r + \gamma \sum\limits_{a'} \pi(a'|s)q_{\pi}(s',a')]$

The bellman equations relates the value of a certain state or state action pair to its successor state and vice versa. It can be used to interpret the meaning of the value functions, for instance the Bellman equation for $v_{\pi}(s)$ shows that $v_{\pi}(s)$ uses the probability of choosing each action $a$ and transitioning to a certain state $s'$ and weighs it against the reward of this state transition $s \to s'$ and the discounted value of $s'$, which is exactly what the expected return of a state is meant to signify.

> **Bellman optimality equation**
>
> It is a criterion which must be satisfied in order for a policy $\pi$ to be deemed the optimal policy $\pi_*$.
>
> > [!Definition]+ **Bellman optimality equation for ** $v_{\pi}(s)$: $v_{}(s) = \max\limits_a \mathbb{E}[R_{t+1} + \gamma v_(S_{t+1}) | S_t=s, A_t=a]$

If the optimal value functions were available, then maximizing return becomes a trivial task as it merely requires following a greedy policy, of selecting the action that has the highest action-value for any given state. However in many real life problems, the number of states and actions available, i.e. the dimensionality of the problem, is **a)** too large for any computer to store an entire value function, and/or **b)** too large for any computer or algorithm to tractably compute the entire optimal value function. For problems where it is possible to compute an approximation of and store the entire optimal value function containing the value of each state action pairs, they are called *tabular* cases and methods which utilizes these value functions are called *tabular methods*.

For problems with high dimensionality, a *function* instead of a table could be used to compute and approximate the value functions, in which case only the parameters defining the function need to be stored.

From chapter 4- dynamic programming

Classical dynamic programming (DP) assumes knowledge of a perfect model of the environment and is computationally expensive, which limits its application to RL. The great utility of DP is in contributing theory. DP based algorithms can turn the Bellman equations defined in the previous chapter into update rules that are used to compute the optimal value functions, and from which an MDP can be easily solved.

Given an MDP with a finite set of states $\mathbb{S}$, there will be exactly $|\mathbb{S}|$ number of value functions. Assuming that the dynamics of the MDP is perfectly known, i.e. we know the probability density function $p(s',r|s,a)$ exactly, and given a certain policy $\pi(a|s)$, it is then possible to create a linear system of $|\mathbb{S}|$ equations with exactly $|\mathbb{S}|$ unknowns. Meaning it would be possible to find the exact value functions $v_{\pi}(s) ; \forall s \in \mathbb{S}$. The consideration of dimensionality returns as the number of terms in each equation in this system is equal to the number of actions times number of states times number of rewards, and as aforementioned the tractability of solving this system of equations diminishes quickly with additional dimensions.

An iterative method of deriving the optimal value function for a given policy can be applied, and involves a back and forth series of *policy evaluation* and *policy improvement*, which as an intended byproduct produces the optimal policy.

> [!Definition]+ **Policy evaluation**
>
> The Bellman equation is computed for a given policy and initial value function, each evaluation provides a new estimate of the value function.

> $v_{k+1}(s) = \sum\limits_a \pi(a|s) \sum\limits_{s',r}p(s',r|s,a)[r + \gamma v_k(s')]$
>
> To loop this update equation until convergence is to perform an evaluation of a policy, as it finds the value function for the given policy.

> [!Definition]+ **Policy improvement**
>
> The old policy π is replaced by a new policy π' which takes an action a' s.t. the action-value function $q_{\pi}(s,a')$ is greater than the value function $v_{\pi}(s)$. For instance, a greedy policy w.r.t. $v_{\pi}(s)$.

The classical DP method is the perform these two steps in sequence over many iterations until the value function and policy are stable, i.e. converged, which is only the case when both these estimates are optimal. This is because if a policy is suboptimal, then there exists another policy which has a higher value function, which in general is true by opting for a greedy policy. Upon switching to a new policy, the old estimate of the value function will no longer approximate the value function for this new policy, and thus when the Bellman equation is computed there will be a difference in the new and old value functions. However, if a policy is optimal, then in the policy improvement step no changes would be made to the current policy, and thus the estimated value function will still hold causing no changes to be observed between the previous and subsequent value function estimate from the Bellman equation. Thus stability can only exist in policy iteration once the policy is optimal, and by extension once the value function is optimal.

Thus in other words, the estimated value function and policy are optimal if and only if both estimates reached a state of stability in a policy iteration algorithm. The *policy improvement theorem* states that by updating a policy to one which takes an action with a higher action-value than the value of the original policy, the value function of the new policy is guaranteed to be at least greater than that of the old policy.

Solving an MDP, i.e. solving an RL problem by implementing an algorithm which iterates between policy evaluation and improvement to derive the optimal policy and value function is the general description of all RL algorithms, which can all be described as *Generalized Policy Iteration* algorithms. Such algorithms do not have the true analytical dynamics of the MDP being addressed, which is one distinction between PI and GPI. Each of which takes a different approach on each element of this iterative process. E.g. the use of ANNs to represent the value function, or performing varying number of evaluation steps versus improvement steps, or the treatment of the probability density functions as continuous or discrete.

DP method of directly applying policy evaluation and improvement iterations is thought to be non-generalizable due to the *curse of dimensionality*, where the number of elements in the set of states increases exponentially as a function of the number of state variables. For example, the state space of a problem of 3 state variables spans 3 dimensions, s_1, s_2, s_3, let's declare each variable to vary between 10 discrete values; introducing an additional state variable s_4 will increase the number of state combinations from $10^3 \to 10^4$, i.e. by simply introducing one additional state variable resulted in the size of the set of states to increase 10 fold.

This in practice has proven to not be as big a hinderance as expected, due to Moore's law, and in general convergence speeds of DP methods can be greatly enhanced by initializing the algorithm with good estimates of the optimal policy and value function.

> [!Definition]+ **Generalize Policy Iteration** A general algorithm which implements the idea of an evaluative process and an improvement process iteratively interacting with each other to solve an RL problem.

From chapter 5- monte carlo methods

This is the first *learning* method, as opposed to the previous dynamic programming method. Quoting from Sutton & Barto:

> "Monte Carlo (MC) methods are ways of solving the reinforcement learning problem based on averaging sample returns"

Such methods estimates *returns* of states or state-action pairs by taking large number of return from an agent acting in an environment or in a simulated environment, so called *actual* and *simulated* experience, then averaging them to find an estimate for the return of each sate-action pair. I.e. they use the sampled returns to estimate the state-value function or the action-value function, which by definition requires experiencing an entire episode before an update is performed, as only then is a return known. These methods adopt the policy evaluation and improvement steps from GPI with the exception being that value functions are **sampled** from experiences instead of **computed** using analytical models of the MDP.

An important property of MC methods is the fact that the estimated value functions for each state or state-action pair are independent from other states or state-action pairs, as each estimates do not use another estimate to compute its' value, in other words MC methods do not bootstrap.

> [!Definition]+ **Bootstrap** To update estimates of the value of a state using estimates of the value of other states, which creates dependencies between estimates.

Just like in GPI, the estimated value function and policy will only be stable in the sense that new updates do not change the estimates when they are optimal.

On/Off-policy methods

Off-policy methods uses a behaviour policy b to sample experiences from an MDP, which is used to estimate the value functions for a target policy π where explicitly $b \neq \pi$.

Whereas on-policy methods uses a policy π to sample experiences in order to estimate value function of or improve π.

On-policies method work with the same framework of GPI with the exception that the policy improvement iterations improve the policy by selecting an ϵ greedy policy rather than a purely greedy policy, to ensure that exploration of the state space still occurs.

> [!Definition]+ **Coverage** The behaviour policy must have non zero probability of picking actions that the target policy has a non-zero probability of picking, i.e.
>
> $\pi(a|s) > 0 \implies b(a|s) > 0$

Importance sampling is one method used to enable using trajectories from b to estimate values of π or even improve it, where V_{π} is estimated by taking a simple average of G_b but with scaling on G_b before using it to update V_{i} using $\rho_{t:T-1}$ through either *ordinary importance sampling* or *weighted importance sampling*. Notice that by weighting the experienced rewards with this factor, rewards that resulted from actions which π is more likely to take than b are weighed more heavily and vice versa.

> [!Definition]+ **Importance-sampling ratio** The ratio of the probability of an experienced trajectory occurring when following the target policy π over the probability of it occurring following the

> behaviour policy b $\rho_{t:T-1} \doteq \prod\limits_{k=t}^{T-1} \frac{\pi(A_k|S_k)}{b(A_k|S_k)}$

> [!Definition]+ **Ordinary importance sampling** $V_{\pi}(s) \doteq \frac{\sum_{t\in \tau(s)}\rho_{t:T-1}G_t}{|\tau(s)|}$

Where $\tau(s)$ is the set of all timesteps where s is visited.

> [!Definition]+ **Weighted importance sampling** $V_{\pi}(s) \doteq \frac{\sum_{t\in \tau(s)}\rho_{t:T-1}G_t}{\sum_{t\in \tau(s)}\rho_{t:T-1}}$

The distinction between these two sampling methods is in their bias and variance of the estimated value function, the ordinary sampling provides an unbiased estimate with unbounded variance; weighted sampling provides an a biased estimate with bounded variance, a more desirable combination of properties than an unbounded variance since it results in more stable learning processes.

Alternative to importance sampling, which performs estimation of the value function using trajectories from all episodes at once, an incremental update method is also used, utilizing the *bootstrapping* concept, or more simply known as a moving average.

> [!Definition]+ **Incremental value function update** $V_{\pi}(s) = V_i(s)$ $V_{i+1} \doteq V_i(s) + \frac{W_i}{C_i}[G_i - V_i]$ where W_i is some variable or constant weight, and: $C_{i+1} \doteq C_i + W_{i+1}$

From chapter 6- temporal-difference learning

Temporal-difference (TD) methods combines the sampling trajectories for value function update from MC, and the bootstrapping nature of DP into one class of methods.

Unlike MC methods which use sampled *returns* to update value function estimates, TD methods use sampled *rewards* as well as *previous estimates of the value function* to update its' estimate. Meaning this class of methods can update estimates every time step, as opposed to every episode as is the case in MC methods.

> [!Definition]+ **Simple TD estimate update** $V_{\pi}(S_t) \leftarrow V_{\pi}(S_t) + \alpha[R_{t+1} + \gamma V_{\pi}(S_{t+1}) - V_{\pi}(S_t)]$

The simple TD estimate update is also known as the *one-step TD* or *TD(0)*, a type of *n-step TD* and *TD(λ)* algorithms respectively.

While theoretically both TD and MC methods make value function estimates which are unbiased and converge as number of samples tend toward infinity, empirically TD-methods are known to converge at a faster rate than MC methods. A rule of thumb that distinguishes TD and MC methods is that TD produces estimates which would fit future data better, while MC methods fit to current data better.

> [!Definition]+ **SARSA** SARSA is an on-policy TD control algorithm, which executes actions based on an estimated action-value function $Q(s,a)$, and updates the value estimate at every time step.
>
> $Q(S,A) \leftarrow Q(S,A) + \alpha[R + \gamma Q(S',A') - Q(S,A)]$

> [!Definition]+ **Q-learning** Q-learning is an off-policy TD control algorithm, which executes actions based on an estimated action-value function $Q(s,a)$, and updates the value estimate at every time step.

> $Q(S,A) \leftarrow Q(S,A) + \alpha[R + \gamma \max\limits_aQ(S',a) - Q(S,A)]$

There is a small difference between Q-learning and SARSA that makes Q-learning an off-policy RL algorithm and SARSA an on-policy one. In SARSA, the action-value (Q-value) estimate is updated based on the previous estimate of the Q-value as well as the Q-value estimate of the subsequent state in the trajectory, and the trajectory is the result of following a policy derived from the action-value function (Q-function), this results in SARSA being on-policy. In Q-learning, the Q-value estimate is updated not by using the Q-value of the next state in the trajectory, but rather by using the Q-value of a state that would have resulted by following a purely greedy policy, thus making Q-learning off-policy.

> [!Definition]+ **Maximization bias** The positive bias on estimation of values that results from using the maximization operator on said estimated values. For example, the Q-value estimate in SARSA suffers from maximization bias, where the estimated Q-value is biased to be higher than the true Q-value, as it uses an ϵ greedy choice over the Q-values to sample the next action.

Maximization bias is a fault of the algorithm, and can cause agents to take inefficient actions. One way to combat this is to use two different value estimates when extracting an action using the maximization operator, and when evaluating the value of extracted action. In the case of SARSA, instead of taking $a' = \underset{a}{\text{argmax}}Q(s,a), ; Q_{a'} = Q(s,a')$, where a' and $Q_{a'}$ are taken from the same Q-estimate $Q(s,a)$, the a' should be a result of one estimate $Q_1(s,a)$ and $Q_{a'}$ should be the result of a separate estimate $Q_2(s,a')$: $$ \begin{align} a' &= \underset{a}{\text{argmax}}Q_1(s,a) \ Q_{a'} &= Q_2(s,a') \end{align} $$

This small change results in the value estimate of $Q_{a'}$ becoming unbiased. A second estimate of $Q_{a'}$ can also be obtained by simply reversing the roles of Q_1 and Q_2. This thus leads to the idea of Double learning.

> [!Definition]+ **Double learning** A method used to avoid maximization bias, where two estimates of a value are kept, and the maximization operator and estimate evaluation are taken separately on the two estimates.

From chapter 7-n step bootstrapping

TD and Monte Carlo methods are two basic frameworks for estimating value functions in an MDP, the only difference lies in what number is used to estimate the value function. In the TD case the value function is estimated using **rewards** that are sampled, while in MC methods **returns** are sampled to estimate the value functions. And in the former, bootstrapping operations are essentially the only operations used to update estimates, whilst MC methods can use either bootstrapping operations or a sample average to obtain the value function estimates. Both methods are free to work with either state-value functions, or action-value functions, or even *after state*-value functions which is a type of value function observed in some classes of MDP's.

In any case, TD methods and MC methods lie on opposite extremes of the same spectrum, and this spectrum is the number of time steps taken before the reward sampled is used to update the value function estimate. With TD methods, the number of time step taken before updating the estimate is *one*, hence TD methods are called 1-step bootstrapping methods.

> [!Definition]+ **n step TD estimate update** $V_{t+n}(S_t) \doteq V_{t+n-1}(S_t) + \alpha[G_{t:t+n}-V_{t+n-1}(S_t)]$ $G_{t:t+n} \doteq R_{t+1} + \gamma R_{t+2} + \dots + \gamma^{n-1}R_{t+n} + \gamma^nV_{t+n-1}(S_{t+n})$

Thus instead of simply taking the next steps' reward to update the current states' value estimate, n number of subsequent rewards are used to update the value estimate. Again, the same idea can be applied to more than merely the state-value function estimate. The more steps of rewards used to come up with an estimate of any return, the lower the error between the estimated value function and the true value function will become.

This idea of taking n steps can be used in all kinds of estimation and algorithms, be it estimating the state-value function, the action-value function, or if the algorithm is off-policy. For off-policy extension, while the importance sampling ratio mentioned before can be directly applied by taking the ratio for the duration from t to $t+n$ for the state-value estimation case, or $t+1$ to $t+n+1$ for the action-value estimation case, a more efficient n step extension of off-policy algorithms exists.

The n step tree backup algorithm is one such efficient extension. Suppose the behaviour policy b experienced a trajectory which starts with a state-action pair, took 2 actions, and therefore has visited 3 states. In the tree backup algorithm, the target policy π will be updated using the rewards of actions that this trajectory did not take

From chapter 8- planning and learning with tabular methods

Planning and learning are two approaches that an RL algorithm can adopt in its' architecture, both of which at their core create estimates of a value function and uses that estimate to perform actions, it is in how they come up with the estimate that differs. To employ a planning approach, the RL algorithm will require a model of the environment or the MDP, which confers these algorithms the name *model-based* algorithms. With a learning approach, the algorithm does not require any model, and as such are called *model-free* algorithms.

What is planning? Planning is when a model of the environment is used in value estimation before producing a policy, for example when the model is used to simulate a trajectory, and this trajectory is used to compute a value function which is then used to compute a policy.

Planning approaches can allow an algorithm's value estimates to converge in less episodes than a learning approach. But for an algorithm to plan means the potential for prior suboptimal policies to be reinforced, in effect causing the agent to be stuck with a suboptimal policy or value estimate. This effect can be observed in simple maze problems, where for the first set of training episodes the shortest path leading the agents from the starting location and goal is found by the algorithm, and the use of planning allows the algorithm to reinforce using this path. But if in the second half of training an even shorter path is created in the environment, the algorithm would be unlikely to explore and come across this more optimal path.

What is learning? Learning is when solely actual experience in an MDP is used for value estimation to produce a policy, for example when an agent performs actions in an MDP and an RL algorithm uses the experienced trajectory to compute values to compute a policy. The temporal difference methods (TD(0), SARSA, Q-learning) and Monte Carlo algorithms presented thus far all use the learning approaches in their architecture.

Part I summary

The space of all RL algorithms is spanned by a number of dimensions.

1. The depth of steps taken to update a value estimate, i.e. how many sequential state-state transitions.
2. The breadth of steps taken to update value estimate, i.e. how many state-state transitions are considered from a single state, expected value updates versus sampled value updates.
3. On-policy vs off-policy, the characteristics of the policy in each version can be varied.

4. Function approximation method, i.e. the size of state space that are tackled by the algorithm, covered in Part II not Part I. Most RL methods in practice estimate value functions at their core,

**From chapter 9- **

The biggest problem with extending the RL theory in the discrete and limited state space case to a continuous state space is the combinatorial explosion in the number of states, and therefore number of values that can exist for a given MDP, this adverse effect is analogous to the curse of dimensionality, which describes how complexity of a problem exponentially increases as more variables or dimensions are introduced. This consequence does not merely impact the number of values that may need to be kept track of in a value function, but also in the exploration of the value function, since it is very unlikely that all states or state-action pairs will be visited.

Therefore, the value functions in larger or continuous state spaces is best treated as an approximating function $\hat{v}(s,\mathbf{w})$ which is parameterized by some vector $\mathbf{w} \in \mathcal{R}^d$, and the true value function $v_{\pi}(s)$ will be approximated by this parameterized function $v_{\pi}(s) \approx \hat{v}(s,\mathbf{w})$.

With this approximation, the value estimate update rules previously presented need to be re-expressed. Taking the example of state-value functions, the previous *tabular* update of the value estimate is:

$$ \begin{equation} v_{k+1}(S_t) = v_{k}(S_t) + \alpha[G_{t+1} - v_{k}(S_t)] \end{equation} $$

The new value estimate update using function approximators for the value function is then not performed on the value itself, but on the weight vector:

$$ \begin{equation} \mathbf{w}*{k+1} = \mathbf{w}*{k} + \alpha[G_{t} - \hat{v}(S_t, \mathbf{w}_k)]\Delta\hat{v}(S_t, \mathbf{w}_k) \end{equation} $$

With $\Delta\hat{v}(S_t, \mathbf{w}_k)$ being a column vector of partial derivatives of the approximator function w.r.t its' parameters, note that the term inside the square brackets is a scalar.

%% This therefore implies that the term inside the square brackets is a matrix? %%

These are MC updates, but a TD method for updating can also be used where the sampled reward R_t plus discounted value $\gamma \hat{v}(S_{t+1},\mathbf{w})$ is used in place of G_t.

**From chapter 10- **