Deep Co-Adaptation of Agents with Probabilistic Q-Value-Surrogates

Timofey Abramski

Master of Science
Artificial Intelligence
School of Informatics
University of Edinburgh
2021

Abstract

In nature, morphology and behaviour are co-adapted through the process of natural selection [1] across many generations. In this project, we co-adapt behaviour and design of robots using reinforcement learning in a data-efficient manner, such that design and controller are co-adapted in the minimum number of design evaluations. Similar to previous work by Luck et al. [19], we achieve this by taking advantage of neural networks' generalisation across the space of possible designs, but unlike previous literature, we include the uncertainty associated with neural networks' outputs as a key heuristic to aid in efficiently exploring the design space. This helps us avoid wasting design evaluations for designs which are already associated with high certainty, therefore lending exploration more to the notion of maximising knowledge [7]. In this report, we explore multiple ways of generating neural network uncertainty estimates, including converting the critic in the Soft-Actor Critic algorithm [14] to a probabilistic one, ensembling neural networks, as well as a novel semi-probabilistic approach involving both deterministic and probabilistic (Monte-Carlo Dropout) neural networks [9]. We demonstrate that two of our methods improve on the data-efficiency of previous work by Luck et al. [19] in the Half-Cheetah environment [6], with our semi-probabilistic approach being particularly successful. We show that the generated uncertainty estimates have the expected properties of epistemic uncertainty, ultimately allowing for our novel combined deterministic and probabilistic algorithm to find better design-controller pairs in fewer iterations, while also achieving significantly higher performance than in the previous work [19], even when running the previous model for more design iterations.

Acknowledgements

I would like to thank my supervisor, Kevin Sebastian Luck for taking me on for the project when I was in a tough spot, as well as for his support throughout the project. I have learned a lot, and I am very grateful for his willingness to help me with any problems I had to deal with.
I would also like to thank my family, as well as my significant other, Sara who gave significant feedback and support in the writing of the thesis.

Table of Contents

Chapter 1

Introduction

1.1 Motivation

In nature, both morphology and behaviour are adapted together in unison, and together determine the possible interactions of a species with its environment [1]. The interplay between morphology and behaviour defines the ability of a species to overcome adversity and hence survive. As an example, the number of limbs and their muscular arrangement defines possible behaviours that an animal can take advantage of which explains the diversity of animal locomotion styles. However, typically, in robotics, the design of a robot is considered as being part of the environment that the agent implicitly needs to take into account when learning a movement strategy or solving a task. This raises the question of how and why the design of such a robot was selected. The optimal behaviour for a robot is a function of the environment and the design of a robot. Typically, evolutionary algorithms are used to create a population of robot designs [25, 5], which are all trained independently to convergence in physics simulations, and the best performing designs are selected on an empirical basis to be deployed in a real-life setting. However, this approach suffers from the simulation-to-reality gap, which results in robots performing worse in real life than they demonstrate during simulations [17, 18]. Some argue that progress in these evolutionary algorithms or simulation experiments results in little to no progress to real life robots due to this gap [20]. Producing and training such intermediate robots selected by an evolutionary algorithm in the real world is a seemingly attractive option, but is infeasible due to the production costs and the amount of data needed to be collected to train a population of robots. Secondly, this also ignores the fact that in nature morphology and controller are inherently interrelated and co-adapted across many generations. Adaptation to en-

vironmental challenges can come in changes to an organism's shape as well as changes to the brain and nervous system [1]. The common approach ignores the fact that there is some transferability in behaviour between designs.

In this project, we take this insight and explicitly model the design as part of the Markov Decision Process (MDP), allowing us to bring the optimisation of design inside the reinforcement learning loop. In particular, we investigate this as it applies to legged locomotion, where the goal is to walk/run efficiently in a particular direction [22]. However, since the ultimate goal is to perform the design exploration as part of a single optimisation which occurs in the real world, it is crucial for any such algorithm to be data-efficient. We hypothesise that this data-efficiency can only come from a model's self-awareness regarding its own uncertainty. A model that is aware that it is uncertain is more useful than a model that is certain but is wrong, and can be designed to take steps that minimise its uncertainty, or maximise its knowledge. In this project, we explore multiple methods of estimating model uncertainty and integrate them with previous data-efficient co-adaption of design and controller by Luck et al. [19], in an attempt to make such an algorithm more viable for production in the real world.

1.2 Related Work

In this project, we focus on using reinforcement learning to co-adapt robot design and morphology [19]. While there are other approaches which rely solely on evolutionary algorithms [25] to produce effective designs, deep reinforcement learning allows us to learn more specialised behavioural policies and movement strategies than using only evolutionary algorithms [19]. The following is an outline of some key literature that is relevant to this direction of research.

Work by Schaff et al. [24] is a recent approach that brings the design parameters of a robot ξ into the definition of the MDP, and therefore allows the policy to condition on the design as $\pi(s, \xi)$, with the general goal of optimising the expected reward $\mathbb{E}_\pi[R]$ while conditioning on the design. However, this approach requires maintaining a population of designs to compute policy update gradients for the population-wide policy, making it difficult for such an algorithm to be applied in the real world to find effective morphologies. Work by David Ha [13] also applied a similar method, except using REINFORCE [29] to make updates to the design. However, this work also relies on maintaining a population of designs. Work by Luck et al [19] takes on a unique perspective that avoids some of these issues by training two variants - population and

individual - of the policy (as well as Q-value networks), which have the role of generalising across design space and specialising to particular designs respectively. This approach avoids keeping a population of designs, making it more practical for successive printing of designs, and applying such an algorithm to real-life robots. Crucially, this allows intermediate evaluation of expected design performance, by simply querying the learned functions for new designs, accompanied with state and action. In prior work [19], this is used as a foundation to bring design optimisation inside the reinforcement learning loop, ultimately allowing good designs to be found in less design evaluations than evolutionary methods by orders of magnitude [25]. However, such algorithms are not yet entirely viable to apply to real life robots, due to the time and financial constraints of producing many robots in order to execute such an algorithm to convergence. This makes the improvement of the data-efficiency of such methods vital for progress in the field, and applicability outside of simulations. This could ultimately be achieved using certainty-aware models known as Bayesian Neural Networks [11].

Bayesian Neural Networks (BNNs) are a class of networks which explicitly model the posterior over weights, allowing access to model uncertainty information [11]. This can be very useful in human-in-the-loop supervised learning setting, for example, in the healthcare setting where neural networks can flag cases for human intervention where a model output has high uncertainty [21]. Similarly, in reinforcement learning uncertainty information can be used in tasks where exploration is important [9], allowing a slightly different view on the exploration-exploitation dilemma, where instead the agent can select actions which result in the maximisation of knowledge or minimisation of future surprise as in the Free-Energy Principle [7].

While model uncertainty has not yet been previously used to help effectively explore the space of possible designs in robotics, Monte-Carlo Dropout [9] is a promising method which can estimate model uncertainty. This method has previously been used to guide exploration in other reinforcement learning settings with continuous state-action spaces, such as Maze games [9], Lunar Lander and MinAtar [4], for which it has been very effective. Monte-Carlo Dropout makes networks inherently stochastic and allows uncertainty information to be generated by analysing the variance between the outputs of multiple forward passes. While there exist a whole class of BNNs which can also be used to estimate model uncertainty, Monte-Carlo Dropout's strength is its simplicity, being implementable on any Multi-Layer Perceptron model that can use *standard dropout* [26], instead of requiring different network architecture. In this project, we will apply Monte-Carlo Dropout [9] to the value function in the algorithm

introduced by Luck et al. [19] in an attempt to improve data-efficiency of the co-adaptation of design and controller. In theory, uncertainty information can help the agent avoid wasting design evaluations where it already has high certainty of the expected reward, and instead focus on more promising designs, with the goal of attaining more knowledge about how design maps to design effectiveness.

1.3 Objectives

The main objective of this project is to address the following question: ***Is it possible to improve on the design exploration-exploitation strategy presented by Luck et al. [19] using model uncertainty estimates, in a way to achieve more data-efficient co-adaptation of design and controller?*** We mainly focus on Monte-Carlo Dropout [9], and ensembling methods to estimate uncertainty, and address our hypothesis. The project can be roughly split into the following main objectives:

- Assess whether the Q-value function trained by Luck et al. [19] implicitely learns uncertainty information which could be used to improve design exploration.
- Add Monte-Carlo Dropout to the Q-network in the Soft-Actor Critic algorithm [14] in order to make it probabilistically interpretable.
- Train an ensemble of Q-value functions to generate uncertainty estimates that can be used in design exploration.

1.4 Report Structure

We organise this report as follows: Chapter 1 gives a motivation for the approaches we attempt in this project and outlines some of the key relevant literature. Chapter 2 gives a background to all of the topics which are required to understand the contents of this thesis. Chapter 3 introduces the problem we are attempting to solve in this thesis and explains the different methodologies we use in order to address the problem. Chapter 4 begins with details regarding out experimental set-up, and continues on to discuss the experiments which we performed as part of the project, as well as a discussion explaining the main observations and findings. Finally, in Chapter 5 we give a summary and conclusion to our experimental findings, and suggest further work which have good potential to further improve on our results and understanding of the problem.

Chapter 2

Background

2.1 Deep Learning

The term Deep Learning refers to the process of training neural networks which consist of an input $\mathbf{x}$, multiple hidden layers $\mathbf{h}_1, \mathbf{h}_2, ...\mathbf{h}_L$, and terminate in an output $\mathbf{y}$. Each layer consists of a linear transformation applied to its input as shown in Equation 2.1, with each transformation followed by an activation function.

$$\mathbf{h}_n = \text{activation}(\mathbf{W}_n \mathbf{h}_{n-1} + \mathbf{b}_n) \tag{2.1}$$

The final output $\mathbf{y}$, is computed in the same way, and is the prediction label of the network. Activation functions are non-linear functions, for example, ReLU, sigmoid and tanh non-linear functions, which ensures that multiple layers of the network cannot be expressed as a single weight matrix multiplication, in other words, allowing the network to learn a more complex set of functions. The network parameters are updated by back-propagating gradients using the back-propagation algorithm [23].

In this project, we are mostly interested in Deep Learning being used as a method for function approximation of the Q-value and policy functions in the reinforcement learning setting, described further in Section 2.2.

2.2 Reinforcement Learning

In reinforcement learning, there is an environment in which an agent lives. The agent makes observations from the environment, known as the state s, uses this information to decide its action with a policy function $\pi(s)$ and is returned a reward signal r. The

goal of the agent is to maximise the reward signal over time. This dynamic between the agent and its environment can be seen in Fig 2.1.

More formally, this interaction can be described by the Markov Decision Process (MDP). The MDP is a 4-tuple consisting of (S, A, P, R), where S is the *state space* which encompasses all of the states that an agent may find itself in, A is the *action space* and contains all of the accessible actions from state s, P is the set of transition probabilities conditioned on the state-action pair (s, a) which describes the probabilities of landing in any state s' and R are the rewards attained by moving from one state to another via a specific action a.

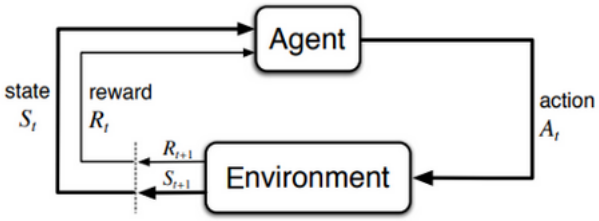

Figure 2.1: RL formulation from [27], depicting an agent's choice to select action, which together with the environment determines the agent's next state S_t and attained reward R_t

Given this formulation, the goal of the agent is to learn an optimal policy π^*, which is a policy that maximises the return $\mathbb{E}_\pi[\sum_{t=0}^{\infty} \gamma^t r(s_t, a_t)]$ where γ is the discount factor in the range $[0, 1]$ and determines how much the agent considers rewards far in the future. The value function $V_\pi(s)$ describes the cumulative expected return from a particular state onwards $\mathbb{E}_\pi[\sum_{t=0}^{\infty} \gamma^t r(s_t, a_t | s_0 = s]$ and a useful quantity to model in order to help learn effective policies. Alternatively, the action value function $Q_\pi(s, a)$ can be used instead of the value function, which conditions on action in addition to state, and therefore denotes the expected cumulative reward after taking action a from state s: $\mathbb{E}_\pi[\sum_{t=0}^{\infty} \gamma^t r(s_t, a_t | s_0 = s, a_0 = a]$. In the case of deep reinforcement learning, the policy, value function and action-value functions can be represented using neural networks. Such networks allow for end-to-end learning, where the internal representations and features are learned directly in a way that best suits the problem at hand. This end-to-end learning occurs with the help of back-propagation [15] which propagates the loss derivative backwards through the layers of the neural networks and updates the model parameters to minimise the loss.

One of the big differences between reinforcement learning and other machine learning paradigms is that in reinforcement learning the agent's decisions affect what data is collected, and there is therefore a constant trade-off between exploration and exploitation. In exploration, the agent attempts to visit novel areas of the MDP with the

goal of attaining more knowledge which will be later helpful in attaining a higher reward, whereas exploitation is where the agent purposely uses only the existing knowledge to select the best currently known actions. An agent which exploits too frequently fails to attain knowledge of the MDP that will help to achieve higher reward in the long run, while an agent that explores too much never takes advantage of the gained knowledge by maximising the reward. This is a constant struggle in the problem of reinforcement learning and is very much task dependant, however, recent work has shown that the use of probabilistic neural networks can be used to effectively explore in an efficient way which is motivated by the agent being self-aware of the gaps in its knowledge, and seeking out areas of the state-space therefore in few iterations gaining maximum knowledge [9]. In this project, the trade-off of exploration and exploitation is important as it applies to finding effective robot design parameters.

2.3 Probabilistic Neural Networks

Probabilistic or Bayesian neural networks (BNN) refer to the extension of standard networks to include posterior inference of model parameters, instead of assuming that the learned weights are deterministic and correct. The posterior is the probability distribution which describes the probability of parameter θ given model evidence X: $P(\theta|X)$, in the form of data. In the training of a standard neural network, the weights of a converged model are equivalent to the maximum likelihood estimate (MLE) of the weights, or $\text{argmax}_\theta P(\theta|X)$. However, it is often the case that this approach is not satisfactory, as it completely ignores the shape of the posterior distribution, and discards any uncertainty information about the output of the neural network, which can be useful in practical applications of supervised learning [8] where humans can intervene when a model is uncertain, or like in this project it can be used as the basis of an exploration strategy in reinforcement learning [9, 8], where exploration is defined in terms of maximising knowledge or reducing model uncertainty and thus expected future surprise [7]. Probabilistic neural networks provide a way to explicitly reason about the weight posterior and therefore model uncertainty. One approach is to use Bayesian neural networks, which explicitly model the posterior for each parameter, resulting in more uncertain outputs from uncertain weights [2]. Alternatively, one can learn an ensemble - train multiple deterministic models instead of one. In the ensemble, each model needs to be trained independently from each other by ensuring that each model starts with its own random initialisation and is trained on different random batches.

Each model's converged weights can be viewed as a sample of the multi-dimensional posterior. Posterior inference in an ensemble is done by comparing the outputs of the ensemble for a given input and by ensembling more models we are effectively reconstructing the posterior from its samples. If the outputs of the models in the ensemble do not vary significantly, this suggests that the posterior is narrow and there is high model certainty, while if there is high variance in the outputs, there is high model uncertainty. Another equivalent view is that every model in the ensemble overfits to the data in different ways, and the averaging over the ensemble removes or "cancels" these errors. A final approach we will focus on, known as *Monte-Carlo Dropout*, applies *standard dropout* [26] at both training and test time [9], and makes a network inherently stochastic, with each forward pass corresponding to a sample of the posterior (see Section 2.4). This approach is a computationally cheap alternative to training multiple networks like ensembling, where instead of using training separate networks we instead instantiate multiple sub-networks with different dropout realisations. In this project, we will focus on Monte-Carlo Dropout and ensembling as methods to estimate model uncertainty to guide exploration.

2.4 Dropout and Monte-Carlo Dropout

Dropout, or *standard dropout* is a common regularisation technique [26] for neural networks which for each forward pass through the network during training, "drops out" or sets neurons to zero, with a probability determined by the pre-set hyperparameter p, also known as the *dropout rate*. This forces the network's learned representation to be more distributed and introduces noise into training, helping the network avoid overfitting. At test time, dropout does not shut off any neurons, and instead scales the outputs of each hidden layer by $1/(1-p)$ to compensate for the increase in magnitude of the signal passing through the network.

2.4.1 Monte-Carlo Dropout Training

In Monte-Carlo Dropout (MCDO), training occurs in the same way as in *standard dropout* [26], that is for each forward pass, every hidden unit is masked with a value of zero with a probability determined by the dropout rate, p. In general, it is possible for different layers to have varying dropout rates $p_1 ... p_L$ where L is the number of layers. Therefore, the set of zeroed hidden units for a given forward pass, or the *dropout mask*

is equivalent to sampling the set of random variables $\bar{\omega}$ from the joint distribution $\bar{\Omega}$:

$$\bar{\omega} = \{\omega_{lk} : l = 1...L, k = K_1...K_L\} \tag{2.2}$$

$$\omega_{lk} \sim \text{Bernoulli}(p_l), \bar{\omega}_i \sim \bar{\Omega}(p_1...p_L) \tag{2.3}$$

where K_l is the number of units in layer l. In back-propagation, gradients don't pass through the masked hidden units, but otherwise parameters are updated as usual, in the negative gradient direction: $\mathbf{w} \leftarrow \mathbf{w} - \alpha\frac{\partial E}{\partial \mathbf{w}}$ with learning rate α and error function E.

2.4.2 Monte-Carlo Dropout Inference

While training of Monte-Carlo Dropout is the same as in *standard dropout*, inference occurs differently. At inference, MCDO involves computing multiple forward passes through the network for a single input, with dropout enabled [9], such that each set of random variables $\bar{\omega}_1...\bar{\omega}_T$ for each forward pass is sampled from the joint distribution $\bar{\Omega}$. The network's best point estimate is computed as the mean across multiple stochastic forward passes: $\mu_Q \equiv \mathbb{E}_{\bar{\omega}\sim\bar{\Omega}}[Q(s,a,\xi,\bar{\omega})] \approx \frac{1}{T}\sum_{t=1}^{T} Q(s,a,\xi,\bar{\omega}_t)$. The variance of the T outputs $\sigma_Q^2 \approx \frac{\sum_{t=1}^{T}(Q(s,a,\xi,\bar{\omega}_t)-\mu_Q)^2}{T-1}$ is an estimate of the network's uncertainty.

Monte-Carlo Dropout is an approximation of the probabilistic deep Gaussian Process and can be interpreted as a variational approximation of the posterior distribution over the network's parameters [9]. Running a stochastic forward pass through a MCDO network can be viewed as taking a sample from the model's predictive distribution, with the output being the point estimate μ_Q with noise sampled proportional to the model's *epistemic uncertainty*, or uncertainty associated with having a lack of data. This allows us to interpret the output as a distribution rather than a point estimate [8] and hence multiple forward passes are required for posterior and therefore uncertainty inference. In this project, we apply Monte-Carlo Dropout to the Q-function which allows us to probabilistically interpret the outputs of the network. In the reinforcement learning setting like ours, uncertainty information can be used as a guide for exploration. While Gaussian Processes (GPs) are seemingly attractive by giving direct access to uncertainty information, unlike MCDO they have the problem of scaling in size with the amount of training data. In the case of co-adaptation of morphology and controller, we are interested in training a Q-function which generalises well for a wide range of designs (ξ) so that the agent can select the most effective design. In order to generalise well across design space, it is crucial that designs which are associated with

high network uncertainty are evaluated, and thus their uncertainty collapsed.

2.4.3 Soft Actor Critic

We use the Soft Actor-Critic (SAC) algorithm [14] to train the Q-function and policy π in lines 12 and 13 of Algorithm 1. This algorithm belongs to the class of off-policy model-free actor-critic algorithms in which the critic estimates the Q-function, and the actor decides actions based on the policy, and updates the policy in the direction suggested by the critic. In particular, the SAC algorithm is a stochastic algorithm based on the maximum entropy RL framework, and therefore is effective for continuous control problems like ours. The SAC algorithm has two loss functions: the Q-loss and policy loss which are used to update the Q-function and policy network respectively. These losses are computed in alternating order, and are used to incrementally improve one after the other. The original Q-loss in the SAC algorithm is computed as:

$$\text{Q-Loss}(B) = \frac{1}{|B|} \sum_{(s,a,r,s',d) \in B} (Q(s,a,\xi) - Q_{target})^2 \tag{2.4}$$

where B denotes the batch of tuples containing the state s, action a, reward r, the next state s' and whether the state s' is terminal, denoted by d. Q_{target} is the target Q-value network, which acts as a target to which the Q-value function approaches in order to improve its estimates. The Q-target is computed as:

$$Q_{target} = r + \gamma(1-d)(Q_{target}(s',a',\xi) - \alpha log \pi(a'|s')), \;\; a' \sim \pi(s') \tag{2.5}$$

where Q_{target} refers to the target network. The target network updates its parameters at every training iteration as a weighted average of its current parameters and the Q-value network $\mathbf{w}_{target} \leftarrow (1-\tau)\mathbf{w}_{target} + \tau \mathbf{w}_Q$, where $\mathbf{w}_Q$ and $\mathbf{w}_{target}$ are the weights of the Q and target networks respectively, and τ is a hyperparameter in the range (0,1] which determines the rate that the target network moves at. The original SAC policy loss is:

$$\text{Policy Loss}(B) = \frac{1}{|B|} \sum_{s \in B} (min_{i=1,2} Q(s, a = \pi(s), \xi) - \alpha log \pi(a|s)) \tag{2.6}$$

where as in the Q-loss, $\min_{1,2}$ refers to the minimisation over the double Q-value networks. In this loss, the gradients travel through the Q-value function in order to update the policy. Therefore, the policy bootstraps from the Q-value function and updates in the direction that maximises the Q-value function.

2.5 Co-adaptation of morphology and controller

In this thesis, we build off work by Luck et al. [19], by making the Q-value function probabilistic in order to improve the exploration strategy. We treat this work as a baseline for our results, and therefore outline the work as follows.

This thesis focuses on the problem of co-adapting robot morphology and behaviour, and the authors present an algorithm which includes the optimisation of design within the reinforcement learning loop. This involves the use of population Q and policy functions Q_{pop} and π_{pop}, as well as individual Q and policy functions Q_{ind} and π_{ind}. The individual variants of the networks are specialists trained only on data collected from the current design, and are therefore not required to generalise across designs, while the population networks are trained on data collected from all designs which are evaluated and therefore generalise across design space. The population Q-value network Q_{pop} is used to predict effectiveness of a design by evaluating the expected return across n start states s_0: $\mathbb{E}_{s\sim s_0}[Q(s,a=\pi_{pop}(s,\xi),\xi)] \approx \frac{1}{n}\sum_{s=s_0^1}^{s_0^n} Q(s,a=\pi_{pop}(s,\xi),\xi)$.

The Q and π functions are deep neural networks and are trained with the Soft Actor-Critic (SAC) algorithm [14] with data that is collected by the agent. There is a policy and Q-loss which bootstrap from a target Q-value function and policy, respectively, in order to make gradual improvements to both networks, described in Section 2.4.3.

The algorithm presented by Luck et al. [19], (shown in Algorithm 1) involves first training the Q and π functions by evaluating 5 prespecified designs, ensuring that the networks don't overfit to too little data, as well as to ensure sufficient generalisation across the design space. After this, the design exploration-exploitation strategy is instantiated, which alternates between exploration and exploitation phases. The exploration phase involves simply selecting a design completely at random, within the range of possible values. The exploitation phase involves selecting the most effective design is the one that satisfies $\text{argmax}_\xi \frac{1}{n}\sum_{s=s_0^1}^{s_0^n} Q(s,a=\pi_{pop}(s,\xi),\xi)$ as estimated by Q_{pop}, which is computed with particle swarm optimisation [3]. The exploration phase ensures the Q and π functions generalise well to different areas of the design space by evaluating on a wide range of new designs, while the exploitation phase effectively tests the model's prediction of good designs by evaluating the ones which are expected to perform well. Alternating between the two strategies ensures that morphology and controller are co-adapted in a data-efficient way.

In lines 4 and 5 of Algorithm 1, we see that the individual variants of the Q and π networks get initialised to the weights of the population networks. This gives a

jump start to the training and ensures that the individual networks are not too far from convergence. Between lines 7-13, we see the reinforcement learning loop which trains the population and individual variants of the Q and π functions. Since the individual networks specialise on specific designs, and the population networks generalise across design space, we see that the individual networks are trained with a quintuple which does not contain the design ξ, whereas the population variants do. Finally, in lines 15-19 we see the alternation between the exploration and exploitation strategy.

In this approach, it is ultimately the inclusion of the design parameters in the state description and thus the generalisation across design space that allows the data-efficient co-adaptation of morphology and behaviour. This allows no time to be spent on areas of the design space which are predicted to be unfavourable. This contrasts with other methods, which instead focus on a more exhaustive search over design space, and ultimately depend on finding good design-behaviour pairs by using orders of magnitude larger number of design evaluations [12] in simulation. This work is particularly important for co-adaptation in the real world, where there is a strict constraint on the number of morphologies that can be produced. We will focus on using epistemic uncertainty to make this design exploration even more efficient.

Algorithm 2: Pseudocode presented by Luck et al. [19]

1: Initialize replay buffers: $Replay_{Pop.}, Replay_{Ind.} and Replay_{s0}$
2: Initialize first design ξ
3: **for** i $\in$ (1,2,...M) **do**
4: $\quad \pi_{ind.} = \pi_{pop.}$
5: $\quad Q_{ind.} = Q_{pop.}$
6: $\quad$ Initialise and empty $Replay_{Ind.}$
7: $\quad$ **while** not finished optimising local policy **do**
8: $\quad\quad$ Collect training examples $(s_0, a_0, r_1, ...s_T, r_T)$ for design ξ and policy $\pi_{ind.}$
9: $\quad\quad$ Add quadruples $(s_i, a_i, r_{i+1}, s_{i+1})$ to $Replay_{ind.}$
10: $\quad\quad$ Add quintuples $(s_i, a_i, r_{i+1}, s_{i+1}, \xi)$ to $Replay_{pop.}$
11: $\quad\quad$ Add start state s_0 to $Replay_{s_0}$
12: $\quad\quad$ Train networks $\pi_{Ind.}$ and $Q_{Ind.}$ with random batches from $Replay_{Ind.}$
13: $\quad\quad$ Train networks $\pi_{Pop.}$ and $Q_{Pop.}$ with random batches from $Replay_{Pop.}$
14: $\quad$ **end while**
15: $\quad$ **if** *i* is even **then**
16: $\quad\quad$ Sample batch of start states $s_{batch} = (s_0^1, s_0^2, , s_0^n)$ from $Replay_{s0}$
17: $\quad\quad$ Exploitation: Compute optimal design ξ with objective function $max_\xi \frac{1}{n} \sum_{s \in sbatch} Q_{Pop.}(s, \pi_{Pop.}(s, \xi), \xi)$
18: $\quad$ **else**
19: $\quad\quad$ Exploration: Sample design ξ with exploration strategy
20: $\quad$ **end if**
21: **end for**

Chapter 3

Methodology

3.1 Problem Statement

We describe the problem of co-adapting robot design (morphology) and its controller (policy) similarly to what is described by Luck et al. [19]. This optimisation can be written as $\theta^* = \text{argmax}_\theta R|\theta$, where R is the reward attained with variables $\theta = [\xi, \pi]$, where ξ are the morphology or design parameters and π is the policy which the agent uses to select actions. In this project, we extend the state space in the Markov Decision Process (MDP) to include the design parameter ξ in addition to the observable state that a robot finds itself in. Under this formulation, the transition probabilities $p(s_{t+1}|s_t, a_t, \xi)$ and therefore the attained reward $r(s, a, \xi)$ is a function of the design parameter ξ. The policy function $\pi(s, \xi)$ is used to select actions to maximise the expected future return for a particular design:

$$\mathbb{E}_\pi[\sum_{i=0}^{\infty} \gamma^i r(s_{t+i+1}, a_{t+i+1}, \xi)|s_t = s, a_i = \pi(s_i), \xi)] \tag{3.1}$$

where $\gamma \in [0, 1]$ is a discount factor, future states s_{t+i+1} are produced by the transition function and actions a_{t+i+1} are sampled from the policy. The goal is to learn a Q-function and policy function π which in addition to generalising well across states-action pairs, generalises well across the design space. Such a Q-value function can be evaluated for start states for a particular design ξ, giving an output that can be interpreted as the expected future reward for the given design, thus this acts as a measure of the general effectiveness of the design. A Q-value function that generalises well across design space can therefore be used to select effective designs by the following optimisation: $\text{argmax}_\xi(\mathbb{E}_{s \in S_0}[Q(s, a = \pi(s), \xi)])$ where S_0 denotes the set of possible

start states, and the policy π should select actions which lead this design to high future return.

In our approach, we use the algorithm presented by Luck et al. [19], shown in Algorithm 1 as a baseline. The main goal of the project is to improve on the design exploration/exploitation strategy, such that good designs are found earlier in training, and given a limit to the number of design evaluations, we find $\theta = [\xi, \pi]$ parameters which yield larger cumulative return. We aim to achieve this by making the Q-value function certainty aware, and exploiting the certainty information to more efficiently explore the design space. We will use Monte-Carlo Dropout (MCDO) as a method for variational inference, which will provide certainty estimates used in the exploration/exploitation strategy. While the previous method could end up evaluating designs which are similar to ones that have already been tested, that is designs for which there is little uncertainty about their effectiveness, we hope to make a strategy that will automatically select designs which maximise the gained knowledge over design space in few iterations and thus result in more effective designs with correspondingly effective policies.

3.2 SAC algorithm with probabilistic critic

The SAC algorithm already uses a stochastic actor, which helps the agent deal with the continuous action space. The first method we attempt in this project is to also convert the critic in the SAC algorithm into a probabilistic one. We do this by replacing the Q-value function in the algorithm with a network with dropout applied to each hidden layer, allowing each forward pass to be a stochastic sampling of the posterior of network weights. With the addition of the probabilistic Q-value function, it is also necessary to alter the two original loss functions Equation 2.4 and 2.6 in the SAC algorithm to achieve stable learning and good performance. For the Q-loss, we want to minimise the temoral difference (TD) loss as follows: $\mathbb{E}_{\substack{(s,a,r,s',d)\in B \\ \bar{\omega}_i \sim \bar{\Omega}}}[(Q(s,a,\xi,\bar{\omega}_i) - Q_{target})^2]$, however, we are free to select how the target Q_{target} is computed and whether we should sample different dropouts $\bar{\omega}_i$ for a particular batch B. The most general loss function which is equivalent to this expected loss is as follows:

$$\text{Q-Loss}(B) = \frac{1}{|B|} \sum_{(s,a,r,s',d)\in B} \frac{1}{U} \sum_{u=1}^{U} (Q(s,a,\xi,\bar{\omega}_u) - Q_{target}(s,a,\xi,\bar{\omega}_t))^2 \tag{3.2}$$

In this equation, for each tuple (s, a, r, s', d) in the batch B, we compute U different forward passes through the Q-value function, and optimise the Q-value function in order to best match Q_{target} in each of the U forward passes. Computing just a single forward and backward pass for any given tuple, as in Equation 2.4 corresponds to the special case of $U = 1$, and is also a reasonable choice for this loss function. While theoretically the choice of U should not change the convergence properties of the Q-value function in isolation, we will later explore its effect on convergence given that the Q-function is bootstrapped from by the policy network. With the addition of MCDO, there is also some flexibility in how the Q_{target} is computed. Crucially, the gradients do not propagate through Q_{target}, and only through the Q-value network. While in the SAC algorithm Q_{target} is originally computed as a moving average between the current network and the target network, we will test periodically entirely copying the parameters of the Q-value network to avoid the moving target problem [27]. We also test computing the Q_{target} in Equation 3.2 by averaging multiple forward passes as in MCDO: $\frac{1}{T}\sum_{t=1}^{T} Q_{target}(s, a, \xi, \bar{\omega}_t)$, or by a single forward pass through the network with test variant of *standard dropout*, which involves the outputs of each layer being scaled by $\frac{1}{1-p}$ in order to compensate for all hidden units being active when compared to train time. With MCDO applied to the Q-value function, we also change the point estimate Q value from Equation 2.6 in order to give the following policy loss:

$$\text{Policy Loss} = \frac{1}{|B|}\sum_{s\in B}(min_{i=1,2}(\frac{1}{W}\sum_{W=1}^{W} Q(s, a, \xi, \bar{\omega}_t)) - \alpha log\pi(a|s))^2 \tag{3.3}$$

with the average of W forward passes through the Q-value function in order to compute the monte-carlo estimate of the mean of the output distribution. Again, similar to the Q-loss, we also have the choice to compute the Q-value with the test variant of *standard dropout*. However, unlike in the Q-loss, the policy loss results in gradients flowing backwards through the Q-value network in order to update the policy, so the choice of W, or Q_{test} needs to be made not only based on the quality and stability of the Q value estimate, but also the quality of the backward gradients, as this will also affects the ability for the policy to improve.

3.3 Ensembling Q-value functions

One basic way of generating uncertainty estimates is to train multiple Q-value functions in parallel like the baseline model in Algorithm 1. In this method, we take the baseline Algorithm 1, but change line 13 from training one population Q-value function, to instead train N distinct Q-value functions in parallel. Where the policy bootstraps from a Q estimate, we arbitrarily select any of the trained $Q^1_{pop.}, Q^2_{pop.}, ...Q^N_{pop.}$ to bootstrap from. Similarly, in line 5, where the $Q_{ind.}$ network is initialised with the weights of $Q_{pop.}$, we arbitrarily select any of the trained $Q_{pop.}$ networks to copy weights from. The Q-value functions' best estimate is computed as the mean of the N Q-value function outputs $\frac{1}{N}\sum_{n=1}^{N} Q^n_{pop.}(s,a,\xi)$ corresponding to each of the networks, and the uncertainty of the result is simply the standard deviation of the N outputs: $\sqrt{\sum_{n=1}^{N}(Q^n_{pop.} - \frac{1}{N}\sum_{m=1}^{N} Q^m_{pop.}(s,a,\xi))^2/N}$.

The method of ensembling multiple neural networks method is well theoretically justified, where given that training between each is uncorrelated, each neural network is a true single sample of the posterior of weights. However, this method is computationally expensive, due to the requirement of training multiple Q-value networks. Training a small number of networks results in unrepresentative samples from the posterior and high variance or unreliable uncertainty estimates. In our tests, we have opted to split the original batch of 256 samples into n equal parts, and feed each part to one of the $Q^n_{pop.}$ networks for training. Although this likely affects the performance of any given network, it makes training multiple networks more feasible due to the reduction in computational time per network update.

3.4 Semi-probabilistic approach

In our final method, instead of making the critic in SAC algorithm inherently probabilistic, we focus on a simplified problem which is sufficient for our purpose of design exploration. We notice that the stochasticity of the Q-value function within the training of a single design is what results in instability during training. However, for the exploration of design space, we are not interested in using uncertainty information that spans state-action pairs, for a particular design, but more so uncertainty from design to design in general. Therefore, we attempt to blend the original baseline model, which has good convergence properties, with the SAC algorithm with a probabilistic critic like we we describe in Section 3.2. Our main idea is to train the baseline model with

deterministic non-bayesian Q-network as described in [19], but to add on and train an additional MCDO Q-network throughout the training procedure, which is only responsible for selecting designs between design evaluations. We call this Q-network Q'.

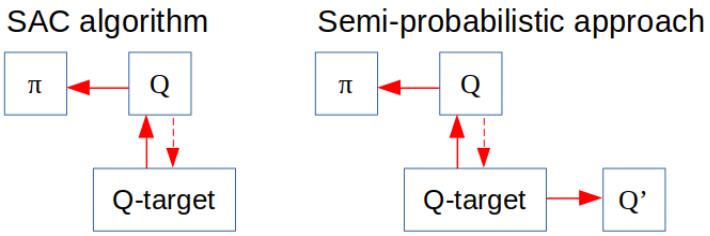

Figure 3.1: Demonstration of the baseline model (left) and our idea (right). Red arrows denote which networks boostrap from others, for example policy π bootstraps *from* a Q estimate. Dotted lines denote copying model parameters from one model to another. In our new method, since the policy does not bootstrap from the new probabilistic network Q', it cannot directly affect stability in training.

Figure 3.1 demonstrates how the addition of the probabilistic Q', does not impact training stability, as no model bootstraps from it. In the computational graph it does not directly lead to the policy network. While this approach attempts to solve a simpler problem than making the SAC algorithm probabilistic, such a treatment theoretically captures all uncertainty information that is needed, and we therefore hypothesise that this method could be workaround for the instability issues associated with making the SAC algorithm a fully probabilistic algorithm.

While this method is more computationally expensive than the baseline model, since we require the training of two Q-value networks, one which is used in estimating across state-action, and one which is used across designs in our extended MDP definition, it is still cheaper than a full Q ensemble approach, which would require training even more Q-value functions simulatenously. Furthermore, unlike ensembling Q-value functions, this approach does not require using different batches across the two Q-value networks, as they are not required to be uncorrelated like ensemble Q-value functions.

The full pseudocode for this algorithm is shown in Algorithm 3. There is only a population variant of the Q' network, since it is only used when selecting the following design, and it gets trained in parallel with the other population networks as in line 13 of Algorithm 3. Unlike the original algorithm, sampling the start state replay buffer in line 17 is required for both exploration and exploitation, since both are dependant on uncertainty estimates. In the following section, we will explain how lines 18 to 25 of the algorithm use the uncertainty estimates from the Q' network in the exploration-exploitation strategy of designs.

3.5 Design exploration

After finishing training a particular design to convergence, the agent must decide on the next design to train on and evaluate, with the overall goal of finding the most effective design ξ^*. We take inspiration from the previous work [19] which used an exploration-exploitation strategy, and hypothesise that a strategy which explicitly takes advantage of model uncertainty estimates will allow us to reach better designs in fewer design evaluations by avoiding exploring designs which are similar to previously attempted ones, or designs which over of the posterior of model parameters is expected to perform poorly. The use of uncertainty estimates allows us to instead focus valuable design evaluations on promising designs which have high potential.

The key quantities which we use in our exploration-exploitation strategy are μ_Q and σ_Q, which are the best point estimate of a design's effectiveness and network uncertainty of the effectiveness respectively. The point estimate μ_Q is computed as:

$$\mu_Q(\xi) = \frac{1}{n}\frac{1}{T}\sum_{s\in sbatch}\sum_{\bar{\omega}_i\sim\bar{\Omega}}^{i=T} Q'(s,\pi_{Pop.}(s,\xi),\xi,\bar{\omega}_i) \tag{3.4}$$

where T is the number of stochastic forward passes, and n is the number of sampled start states from $Replay_{s_0}$. This point estimate is computed by evaluating the Q-value for multiple start states, while σ_Q is the uncertainty for a design and is the standard deviation across each stochastic forward pass, computed as follows:

$$\sigma_Q(\xi) = \sqrt{\frac{\sum_{\bar{\omega}_i\sim\bar{\Omega}}^{i=T}(\frac{1}{n}\sum_{s_j\in s_{batch}}^{j=n} Q'(s_j,a,\xi,\bar{\omega}_i) - \mu_Q(\xi))^2}{T}} \tag{3.5}$$

where in each forward pass we use an entire batch of start states to estimate the expected effectiveness of a design, $\frac{1}{n}\sum_{s_j\in s_{batch}}^{j=n} Q'(s_j,a,\xi,\bar{\omega}_i)$. The main strategy which we attempted is shown in lines 18 to 25 of Algorithm 3, which shows the optimistic exploration strategy which finds the design with the highest potential: $\text{argmax}_\xi(\mu_Q+\sigma_Q)$, and the exploitation strategy which finds the design which the agent is most certain will perform well, expressed as $\text{argmax}_\xi(\mu_Q-\sigma_Q)$. These can be understood as a probabilistic upper confidence bound of expected performance and probabilistic lower confidence bound of expected performance, respectively. Practically, both strategies find the optimal design according to the objective function using particle swarm optimisation [16], which fixes states selecte from the batch s_{batch}, and performs gradient ascent

in design space from 250 different initial random designs. The design that corresponds to the maximum out of the 250 optimisations is chosen as the next design to be trained.

The exploration strategy helps explore new effective designs which are dissimilar to what has previously been evaluated and therefore maximises knowledge, and minimises the KL-divergence between the estimated and true distributions [7], hence helping generalisation across design space. On the other hand, the exploitation strategy can be seen as a pessimistic estimate, or a probabilistic lower bound of the expected performance and helps the agent ensure that what it is certain will perform well, does perform well, and therefore avoids the model from running into delusions. If the agent is wrong about its certainty of the effectiveness of a design, evaluating the design in the exploitation phase will force the Q' to be updated to reflect this new knowledge.

Algorithm 4: Co-adaptation algorithm, with the addition of probabilistic Q' for design exploration

Initialize replay buffers: $Replay_{Pop.}, Replay_{Ind.} and Replay_{s0}$
Initialize first design ξ
Initialise deterministic networks $\pi_{pop.}$ and $Q_{pop.}$
Initialise Monte-Carlo Dropout network Q'
for i $\in$ (1,2,...M) **do**
  $\pi_{ind.} = \pi_{pop.}$
  $Q_{ind.} = Q_{pop.}$
  Initialise and empty $Replay_{Ind.}$
  while not finished optimising local policy **do**
    Collect training examples $(s_0, a_0, r_1, ...s_T, r_T)$ for design ξ and policy $\pi_{ind.}$
    Add quadruples $(s_i, a_i, r_{i+1}, s_{i+1})$ to $Replay_{ind.}$
    Add quintuples $(s_i, a_i, r_{i+1}, s_{i+1}, \xi)$ to $Replay_{pop.}$
    Add start state s_0 to $Replay_{s_0}$
    Train networks $\pi_{Ind.}$ and $Q_{Ind.}$ with random batches from $Replay_{Ind.}$
    Train networks $\pi_{Pop.}$,$Q_{Pop.}$ and Q' with random batches from $Replay_{Pop.}$
  end while
  Sample batch of start states $s_{batch} = (s_0^1, s_0^2, , s_0^n)$ from $Replay_{s0}$
  if *i* is even **then**
    Exploitation: Compute optimal design ξ with objective function $\max_\xi(\mu_Q - \sigma_Q)$
    where $\mu_Q = \frac{1}{n}\frac{1}{T}\sum_{s \in sbatch}\sum_{\bar{\omega}_i \sim \bar{\Omega}}^{i=T} Q'(s, \pi_{Pop.}(s, \xi), \xi, \bar{\omega}_i)$
    and $\sigma_Q = \sqrt{\sum_{\bar{\omega}_i \sim \bar{\Omega}}^{i=T}(\frac{1}{n}\sum_{s_j \in s_{batch}}^{j=n} Q'(s_j, a, \xi, \bar{\omega}_i) - \mu_Q)^2 / T}$
  else
    Exploration: Compute optimal design ξ with objective function $\max_\xi(\mu_Q + \sigma_Q)$
    where $\mu_Q = \frac{1}{n}\frac{1}{T}\sum_{s \in sbatch}\sum_{\bar{\omega}_i \sim \bar{\Omega}}^{i=T} Q'(s, \pi_{Pop.}(s, \xi), \xi, \bar{\omega}_i)$
    and $\sigma_Q = \sqrt{\sum_{\bar{\omega}_i \sim \bar{\Omega}}^{i=T}(\frac{1}{n}\sum_{s_j \in s_{batch}}^{j=n} Q'(s_j, a, \xi, \bar{\omega}_i) - \mu_Q)^2 / T}$
  end if
end for

Chapter 4

Experiments

In this Chapter, we will first describe the experimental set-up which we used, and then in Section 4.2 we present some analysis of the Q-value function as trained by Luck et al. [19]. This will then lead into methods which explicitly model uncertainty information, SAC algorithm (Section 4.3) with probabilistic critic, ensembling Q-value functions (Section 4.4) and our novel merged approach, which uses both deterministic and probabilistic Q-value functions (Section 4.5).

4.1 Experimental Setup

Our experiments are completed in the PyBullet [6] Half-Cheetah environment. As shown in Fig 4.1, this is a simplified physics environment in which we learn the behaviour policy π for the stick-figure of a cheetah, which is constrained to moving in a 2D plane perpendicular to the flat floor. The state of the cheetah is described by a 17-dimensional vector, in which the first 6 numbers describe the position of the robot's joints, the next 6 describe the joint velocities and the final 5 describe the torso position.

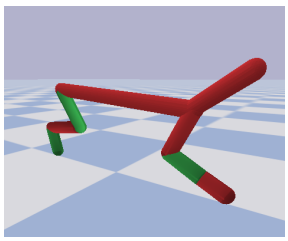

Figure 4.1: Cheetah agent in the Half-Cheetah environment. The agent learns a policy that controls the joint acceleration in order to move horizontally.

In this project, where we extend the state-space to include the design parameters of the robot ξ, the state-space includes an additional 6 dimensions $(\xi_1, \xi_2, ...\xi_6)$ which describes the lengths of the robot's 6 limbs. Each ξ_i is limited to the range $[0.8, 2.0]$, where the final design is computed by $(\xi_1 \cdot 0.29, \xi_2 \cdot 0.3, \xi_3 \cdot 0.188, \xi_4 \cdot 0.29, \xi_5 \cdot 0.3, \xi_6 \cdot 0.188)$. The action that the robot takes is a 6-dimensional vector which gives the joint acceleration values, mimicking the effect of muscle contraction in mammals. The goal

of this environment is to learn a policy π which allows the robot to effectively move in the horizontal direction, and the reward is therefore defined at every timestep as the amount moved in the right direction is $r = \max(\frac{\Delta x}{10}, 0)$ where Δx is the horizontal speed.

We keep the same Soft-Actor Critic set-up, as well as Q and policy architectures as in [19]. All networks have 3 fully connected hidden layers, each with 200 hidden units with ReLU activation applied to each layer. The policy also uses a tanh activation on the final layer, in order to squash the outputs after the elementwise multiplication with the sampled Gaussian. The output is a single unit for the Q-value functions, and a 6-D action vector for the policy networks. The input size varies between the variants of the policy and Q-value networks and are shown in Table 4.2. The population variants have input sizes 6 greater than the individual networks, to account for the 6 design parameters and the Q-value networks have input sizes 6 greater than policy, since they take as input an additional 6-D action vector. We show the set of hyperparameters that we used in the SAC algorithm in table 4.1.

Hyperparameter	Value
Training iterations in pre-set designs	300
Training iterations in explored designs	100
Batch size in exploitation strategy	32
Batch size in training Q and policy	256
Steps per episode	1e3
Discount factor, γ	0.99
Copy weighting factor, τ	5e-3
Q-value function learning rate	3e-4
Policy learning rate	3e-4

Table 4.1: Set of hyperparameters used in our experiments with the SAC algorithm [14].

Network	Input size
$Q_{pop.}$	29
$Q_{ind.}$	23
$\pi_{pop.}$	23
$\pi_{ind.}$	17

Table 4.2: Input sizes of the 4 networks in this project. Each network has 3 hidden layers with 200 units per hidden layer and ReLu activation applied to each layer.

4.2 Variance over start-states for Q-value uncertainty

Before implementing a probabilistic neural network, we hypothesised that it may be that the baseline model by Luck et al. [19] has already implicitely learned uncertainty information, which could be extracted from the Q-value network by evaluating designs over different start states s_0. If this is the case, then this could be directly used to guide a design exploration strategy without the use of probabilistic neural networks. If we assume that there is some standard variance between the Q-values for different start states, which remains below some threshold T for all designs ξ (see Equation 4.1), then the Q-value network must be uncertain regarding designs ξ for which the model

estimates a variance greater than the threshold T.

$$\mathbb{V}_{s\sim s_0}[Q(s,a\sim\pi(s,\xi),\xi)] < T \;\; \forall \;\; \xi \tag{4.1}$$

We motivate our hypothesis by the the fact that we expect - to a first order approximation - that only effectiveness, as measured by $\mathbb{E}_{s\sim s_0}[Q(s,a,\xi)]$, will vary with design, while the variance should remain similar. We generally expect favourable start states - those which place the robot in a horizontal or forward leaning position (giving the robot a head start) to be associated with higher Q-values. On the other hand, unfavourable start states - leaning backwards from the direction of motion to be associated with lower Q-values. However, in both cases we expect the magnitude of variation to not be significantly dependant on design. If an agent finds itself in a very undesirable start state s, it will likely achieve lower return R_t than the expected return $\mathbb{E}_{s\sim s_0}[Q(s,a,\xi)]$, and the difference between the two ΔQ is mostly dependant on the choice of unfavourable state s, not the design ξ as demonstrated in Figure 4.2.

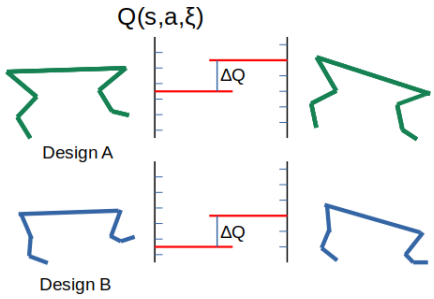

Figure 4.2: Demonstration of our hypothesis. The difference in Q values ΔQ between one start state and another being primarily a function of the two start states being compared, not the design itself. The difference is the same between the two start states for Design A and B.

We tested our hypothesis by evaluating a trained Q-function on two sets of designs, one set being designs on which the Q-function has been trained on, and another set which contains previously unseen designs. Our hypothesis would suggest that trained-on designs would yield Q values with some typical spread that differs significantly to the spread for previously unseen designs. We compared the following set of Q values for the two categories of designs $\{\frac{1}{T}\sum_1^T Q(s_0^1,a\sim\pi_{pop.}(s),\xi),\ldots,\frac{1}{T}\sum_1^T Q(s_0^N,a\sim\pi_{pop.}(s),\xi)\}$, which contains N elements, each corresponding to the evaluation of a unique start state from the batch $s_{batch}\sim Replay_{s_0}$ (as shown in Algorithm 1). We used a sample size of $N=500$ and used the same batch of start states between the set of seen and unseen designs. Since the policy is stochastic, for each start state, it is necessary to average the Q values from T actions sampled from the learned population policy $\pi_{pop.}$, where we selected $T=50$ to ensure low variance in the results. We used a Q-value function which had been trained on 10 designs to ensure that it had not attained good enough generalisation for all designs, that is, there is significant uncertainty variation across design space, and it is still necessary to explore effectively.

Fig 4.3 shows the Q-values for two designs, corresponding to the set described above. As our next step, we computed such sets for 130 seen and unseen designs, and

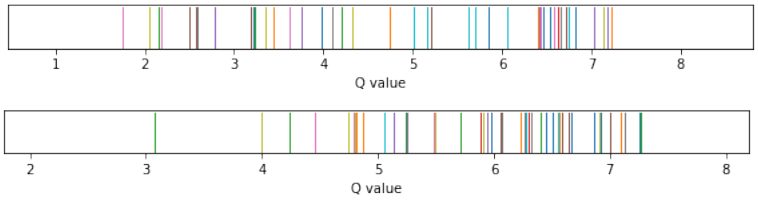

Figure 4.3:
Example of Q-value distribution across start states sampled from s_{batch} from an evaluated design (top) and previously unseen design (bottom).

analysed the mean and standard deviations across different designs. Figure 6.1 in the Appendix, shows the distribution of means of Q values across start states for a set of unseen designs as well as seen designs. As expected, the two distributions of means differ significantly, since this is dependant on the designs in the untrained and trained sets. On the other hand, Figure 4.4 demonstrates that the standard deviation across start states does not vary significantly between the set of seen and unseen designs, despite the sets of designs being different. In support of our hypothesis, we observe that for trained designs, most designs (90%) fall below a threshold of 1.1, while there are a significant number of untrained designs which have a standard deviation greater than 1.1. Furthermore, the average standard deviation for untrained designs is higher, at 1.11 versus 0.79 for trained designs.

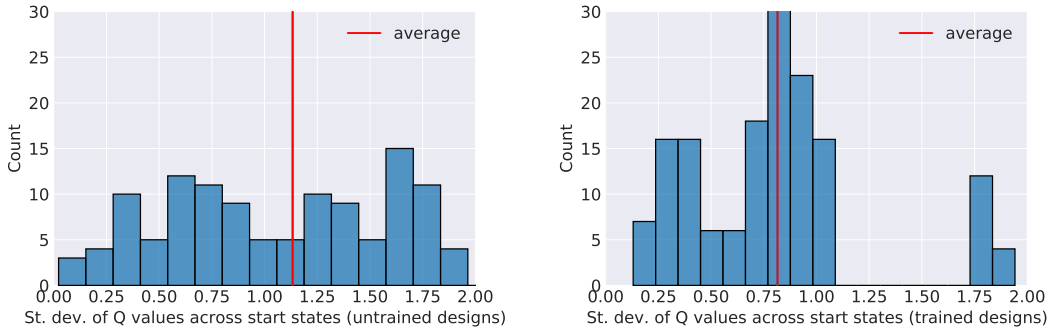

Figure 4.4:
Histograms of standard deviation across start states computed as $\sqrt{\frac{1}{n}\sum_{s_j \in s_{batch}}^{j=n}(Q(s_j,a,\xi) - \frac{1}{n}\sum_{s_i \in s_{batch}}^{i=n} Q(s_i,a,\xi))^2/n}$ for untrained (left) and trained (right) designs. We use $N = 500$, over a sample size of 130 designs. The right figure demonstrate that there are certain characteristic standard deviations which are generally to be expected (less than 1.1), while the left figure shows presence of all standard deviations, due to the networks poorly extrapolating for designs which it hasn't been trained on.

Re-running this analysis with different random seeds, as well as different trained Q-value functions produces the same results, suggesting that our result is statistically significant, which is due to the high number of sampled start states $(N = 500)$. We observe that to a large extent our hypothesis holds true, that designs with standard deviation across start states above 1.1 are significantly more likely to be an untested

design. This is likely because there is high model uncertainty associated with these designs, and the model by chance outputs Q-values for different start-states which are overly varied. However, multiple issues arise regarding the use of the standard deviation as a heuristic for model uncertainty.

Firstly, while an exploration strategy that selects designs with the largest standard deviation would mostly select new designs, it would avoid selecting 55% of the new designs which happen to have have lower standard deviations than 1.1. Secondly, we see that there are outliers that do not conform to our hypothesis, in this case there are seen designs with standard deviations of ~ 1.8 across start states which would be selected early in the exploration strategy, effectively wasting an evaluation. This is due to our assumption that the variance of Q-values across designs depends only on start states, and not on design, while in reality some designs may be particularly well or poorly suited to certain start-states. Finally, it is also not clear that designs which have higher standard deviations are ones which are associated with the most model uncertainty. Therefore, such a heuristic makes for a brittle and flawed strategy which would not optimise well for gaining the most knowledge in the fewest iterations. For the remainder of the project, we focus our efforts instead on building a probabilistic Q function which explicitly models uncertainty in order to explore the design space.

4.3 SAC algorithm with probabilistic critic

We altered the Q-value function to allow for a probabilistic interpretation of the ouputs, within the Soft-Actor Critic [14] algorithm. This involved adding dropout to each of the 3 hidden layers of the Q-network. We altered the Q-loss to the one shown in Equation 3.2 and policy loss to Equation 3.3, to account for the stochasticity of the outputs. We empirically found that $T = 10$ forward passes through the target network, $U = 10$ updates to the Q-value network, and $W = 1$ Q evaluation in the policy loss worked effectively. Our first goal is to achieve stable learning and similarly effective policies to Luck et al. [19], which we will now refer to as the baseline model.

4.3.1 Dropout rate variation

As our first experiment, we varied the dropout rate p in the Q-value network and measured cumulative reward attained by the trained policy. Allowing for exploration of the design space adds a lot of variation to the results and thus makes it difficult to investi-

gate the stability of learning, so we instead compared the MCDO variant of the SAC algorithm to the baseline while training on 5 pre-selected designs. The same 5 designs were chosen for both algorithms. This was set to be part of a single run, so that we rely on the $\pi_{pop.}$ network to generalise in design

Figure 4.5 shows the cumulative episodic rewards attained by agents with different dropout rates, with shaded regions showing the standard deviation across 5 runs. Firstly, we notice that the baseline with no dropout is the most effective method across all of the 5 tested designs and results in very little variation across the runs. On the other hand, dropout rates ranging between 0.1 and 0.4 give successively worse and worse performance and have higher variance between runs. We see that a very small dropout rate of $p = 0.01$, which on average shuts off only 2 out of 200 hidden units per hidden layer, results very similar performance to the baseline which does not use dropout, showing that the reduction in performance is continuous with respect to p. While we are comparing the effectiveness of the learned policy and not the Q-value function, the Q-value function is crucial in the training process as the policy boostraps from the Q-value function through backpropagation in the policy loss.

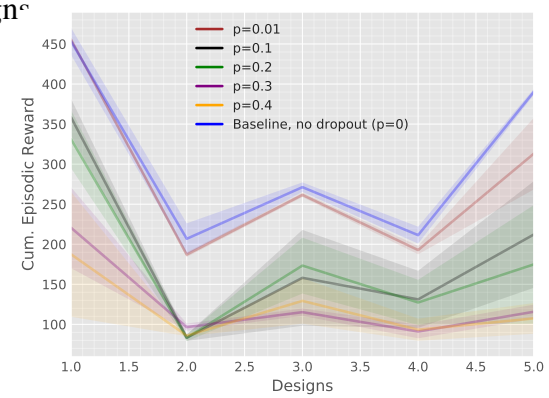

Figure 4.5: Cumulative reward attained by models with varying dropout rate p. Increasing the dropout rate decreases stability of the algorithm and therefore reduces performance. Standard deviation is shown across 5 runs. $p = 0$ replicates the original baseline results.

4.3.1.1 Loss function analysis

Figure 4.6 shows some of the typical Q-loss and policy loss plots for various dropout rates. The x-axis denotes which design is being trained on, with every design being trained for 30,000 iterations. For the $p = 0.1$ experiment, we see similar Q and policy loss to the baseline, except during the training of the 3rd design, where there is a divergence in the both losses. This suggests that at some point during training, some Q estimates became very far from the Q-target, resulting in large updates to the Q-value network, which is naturally followed by large updates to the target Q-value network. This begins a cycle of diverging Q values. This divergence results in astronomically high Q-loss, and the policy loss plot shows the corresponding very large and negative loss due to the selection of actions which result in unreasonably high Q estimates. Such high Q estimates are not representative of actual performance, as the policy bootstraps

from the Q-function and therefore assumes that the Q-values are correct. The presence of weight decay in our network is likely what eventually stops the model from diverging too far away, however by this point it is likely that much of what the network has learned has been forgotten due to large variations in the weights, and the agent begins learning from scratch. Such divergences don't occur in the baseline model, pointing towards dropout's stochasticity making learning unstable. For higher dropout rates of $p = 0.3, 0.4$ we see a different problem emerge, whereby the Q-loss becomes much smaller than the baseline's throughout all of training, meaning the Q-value network always matches the target network's outputs. This points towards an early convergence problem in which the Q-value network and target-Q-value network are very similar, therefore resulting in no updates to either network. In the policy loss plots we see that the higher the dropout rate, generally the higher the loss that is achieved. Apart from diverging cases, we see that the policy loss is generally stable, so the agent has converged to sub-optimal policies. We expect that with stable training, baseline performance should be achievable with a probabilistic Q-value function.

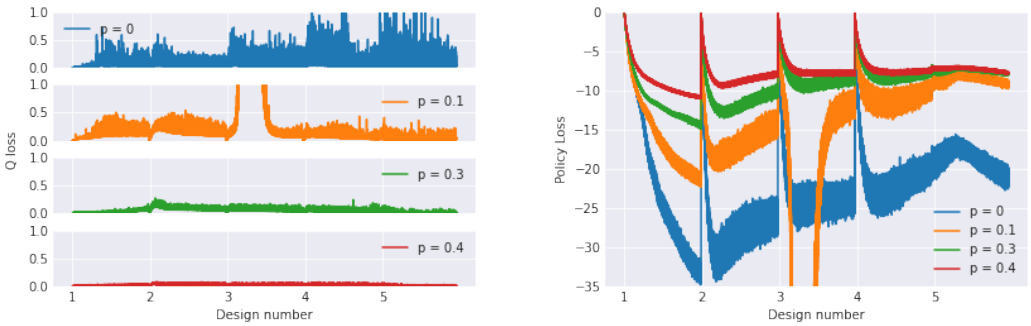

Figure 4.6:
Plots of Q-loss and Policy loss while training on 5 pre-determined designs, with varying dropout rates p. These plots give us some insight into some of the convergence problems associated with using a stochastic Q-value function.

These loss plots as well as the strong variation between runs suggest that the introduction of dropout has caused instability in the learning process of the Q-value function, as well as potentially the policy. Even the addition of what is generally considered a very low dropout rate of 0.1 [4] results in significantly lower performance. While MCDO has been shown to result in stable learning in the literature [9], the difficulty of our method is that we are integrating dropout with the SAC algorithm, an algorithm which has three components: Q-value network, Q-target network, and policy network, which bootstrap from each other. The Q-loss is computed with the Q-target, a network which has its parameters move in the direction of the Q-value network, thus effectively the Q-value function bootstraps from itself and relies on updates being stable. How-

ever, with a stochastic Q-value function, this can prevent the Q-value network itself from converging as normal. Similarly, the policy update aims to update the policy in the gradient direction that results in an action with the highest Q-value. If the policy updates are heavily biased by the dropout mask in a particular forward pass, this can result in a policy with subpar performance. In the following sections, we will describe our attempts to resolve the instability to retain the high performance of the baseline, while also having access to model uncertainty.

4.3.2 Improving stability through Q-loss and Q-target variation

Knowing that the stochasticity of the Q-function is what is causing the instability, it is imperative alter the Q-loss function in an attempt to stabilise training of the Q-value function. The Q-loss function which we have used up until now is shown in Equation 3.2. The parameter U in the Q-loss function determines the number of dropped out forward passes that are done through the Q-value network. Having too few forward passes results in too few weights in the network being trained to a datapoint, while using too many forward passes makes the network overfit to a single datapoint by doing multiple gradient steps to minimise its loss. Using too few forward passes U can result in more noisy updates, which is acceptable in the supervised learning setting, however, in our case since the policy bootstraps from the Q-value function at every iteration, it is important that the Q-value function remains stable from iteration to iteration.

The Q-targets that are used in training the Q-value function are also a key factor in the Q-loss which has the potential to destabilise training. Up until now, the target Q-value network was set up to update in the same way as in the baseline [19], which is to update at every iteration with $\tau = 0.005$, as discussed in Section 2.4.3. With the addition of MCDO to the Q-value network, it is significantly more likely for individual gradient updates to the Q-value function to increase the overall loss of the Q-value function with respect to the target, which is due to bias introduced by sampling only some of the possible dropout masks $\bar{\omega}_i$. A poor update, which on average increases loss across all forward passes $\bar{\omega}_i$, may also result in a poor policy update, as well as a poor Q-target update which makes the following Q update also more unstable. We only expect that MCDO will converge across multiple forward passes, so long as there is a stable target, for example, in supervised learning where labels are stationary [9].

The moving target problem [28] is a typical problem in the RL setting, although in the case of a stochastic Q-value function like ours, it may be that the target needs

to be even more slow-moving than the current $\tau = 0.005$ configuration. We attempt to improve on the stability by updating the Q-target network every N iterations by entirely copying the weights from the Q-value network (equivalent to $\tau = 1$). Locally, between target-Q-value network updates the target network is entirely stationary, giving the Q-value function a better chance to improve before the target is updated. Finally, we test the targets being produced by the target network as set in inference mode in *standard dropout* as originally proposed by [26], which results in no stochasticity in the output of the network, with the goal of achieving a stable target. For these experiments, we use $p = 0.1$ since this is considered to be a low dropout rate [4], which sets a low bound for required stability within our algorithm to achieve uncertainty estimates.

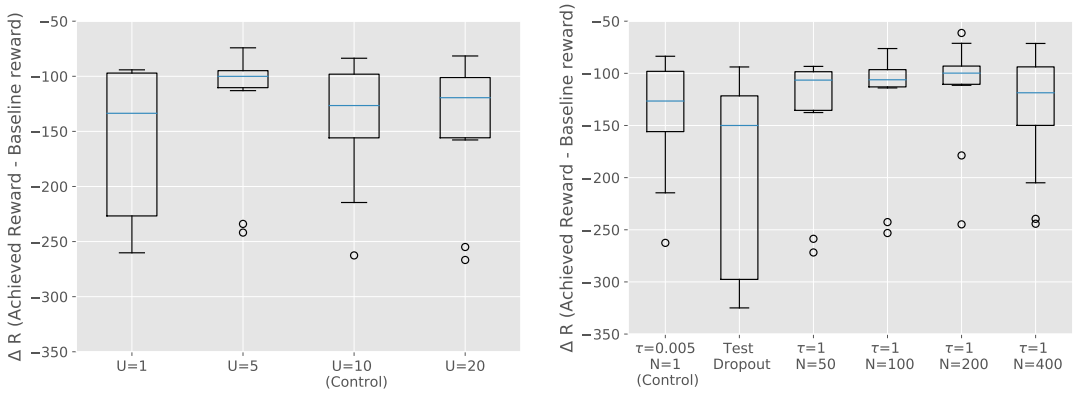

Figure 4.7:
Decrease in attained reward per design when compared to baseline, across 5 runs (each with 5 designs). We see that all of our dropout variations are negative, meaning they average lower performance than baseline. On the left, we see variations to U, number of forward passes in the Q-loss, and on the right, we see variations to the target-Q-value network. $U = 5$ and a slow moving target $\tau = 1, N = 200$ achieves highest performance as well as least variance, while also entirely avoiding divergences, demonstrating improvements in stability of learning.

4.3.2.1 Analysis of Q-loss variation

Figure 4.7 shows the performance across 25 designs for different number of Q evaluations compared to the baseline model. We see that 1 evaluation in the loss function appears to do significantly worse than 5, 10 and 20 evaluations, which is as expected due a significant number of weights not being updated through the network, approximately $(1-p)^2$, or 81% of all model parameters. On the other hand, it appears that U from 5 up to 20 evaluations are similarly effective, with $U = 5$ leading to the best performing policies, with the highest average return, most closely matching the baseline,

as well as the lowest variance, suggesting that this setting has incrementally stabilised training. $U > 5$ likely performs worse due to the larger number of performed gradient steps. The Q and policy losses in Appendix Figure 6.2 don't significantly vary between $U = 5, 10$ and 20.

4.3.2.2 Anaylsis of Q-target variations

Changes to the way in which the Q-target is computed are shown in Figure 4.7 (right), with absolute performance for each design is shown in Appendix Figure 6.4. We see that using the test/inference mode of dropout as originally proposed by [26] performs very poorly. Although this method is often used in practice and typically works effectively, we note that there is no theoretical justification for the different use of the network at test time [9]. In our case where the target network is updated as a weighted average of itself and the Q-value network, we observe that dropout in test mode breaks down and results in very poor performance. On the other hand, we observe that copying the network parameters from the Q to target Q-value network ($\tau = 1$) instead of using a weighted average, appears to significantly improve the mean reward, while also reducing the variance, which again suggests that the training is becoming more stable. The most effective learned policy resulted from copying the network parameters every 200 steps. The stabilisation of the Q-value function results in the policy also attaining more stable updates during the bootstrap. While making the Q-target more slowly moving improves the stability, this comes at the cost of requiring more iterations to converge to a similarly effective policies, due to slower Q convergence, hence the worse performance attained by $N = 400$. The attained losses are shown in Appendix Figure 6.7, and demonstrate that our changes have removed divergences. Appendix Figure 6.4 shows our attempts to stabilise the target further by using 50 forward passes through the target network. We see no difference in the results, suggesting that 10 forward passes is a large enough sample size to produce a stable target.

Based on these results, it appears that some of our changes have incrementally stabilized training, and we have achieved more consistent and higher results with a similar set-up. However, we note that even with these changes the final performance is still significantly lower than the baseline model, on average, achieving approximately 100 lower reward per design. In the next experiments, we continue with $U = 5$ forward passes in the Q-loss function, and a slow moving Q-target updating every 200 steps.

4.3.3 Alternative stabilisation strategies

In Section 4.3.2, we discovered that through some changes to the Qloss function, it is possible to achieve more stable learning, avoid divergences, and therefore learn more effective policies. In this section, we will try experiments to stabilise training further.

Increasing iterations and removing dropout from the final layer: We attempt increasing the number of training iterations on the 5 initial designs, since it is possible that a stochastic critic in the SAC algorithm simply needs more training time before convergence. Secondly we test stabilising the SAC algorithm further by removing dropout from the final hidden layer of the Q-value function. While this makes MCDO a worse approximation to the Deep Gaussian process [9], it may give us some stability at the cost of less accurate uncertainty estimates. In Appendix Figure 6.5, we see that increasing the iterations gives a significant improvement up to 600 iterations (compared to 300 in the baseline). In Appendix Figure 6.6 we show the effect of removing dropout from the final layer. We see significantly higher performance overall, although surprisingly, we still achieve high variance between runs, and pay the price of getting inaccurate uncertainty estimates. We continue our future experiments with 600 iterations of training, and dropout applied to all layers.

Policy Loss variations: Similarly to the Q-loss, in the policy loss (Equation 3.3), we have the choice to vary W, the number of dropout forward passes through the Q-value function. However, it is important to note that unlike the Q-loss, where we train the Q-value function such that each of the forward passes matches the target, the policy loss instead attempts to maximise the average Q value across the W forward passes, based on the selected action. While using more W forward passes improves the accuracy of the Q estimate, as it is equivalent to taking more samples of the posterior, this can introduce an optimisation problem whereby some of the Q evaluations result in dominant backward gradients, which may have the effect of diminishing the gradients from other forward passes. Paradoxically, using more forward passes can therefore result in more bias from individual sampled dropout masks $\bar{\omega} \sim \bar{\Omega}$, rather than smoothing out any bias introduced by any particular $\bar{\omega}_i$. We will investigate the effect of this parameter, as well as using the test mode of dropout in the Q-value function within only the policy loss. While in section 4.3.2 we found that this did not work effectively for the target Q-value network due to the way it is updated, in the policy loss the test mode of

dropout is applied directly to the Q-value function. While backpropagating gradients through dropout in test mode has not been described in the literature (hence the name test), it is particularly appealing to test in the policy loss for two reasons. Firstly, it may reduce bias introduced by the selection of any particular mask $\bar{\omega}_i$, and secondly it avoids using multiple forward passes through the Q-value function, and hence avoids the problem of dominating gradients.

In Figure 4.8, we see the effects of these variations, with more detailed results shown in in Appendix Figure 6.8. We observe that there is no significant difference between any of these settings. It appears that $W = 5, 10$, as well dropout in test mode achieve similarly high results, although we note that there is still significant variation between runs. The small changes to the results from varying the policy loss suggests that the policy updates are not a limiting factor, leaving instability in the Q updates as the major bottleneck, preventing us from achieving higher and more stable performance. Despite our attempts to stabilise training, the performance is still consistently lower than what is achieved by the baseline, by ~ 50 reward per design. In the following sections, we explore other methods for achieving uncertainty estimates while keeping training stable.

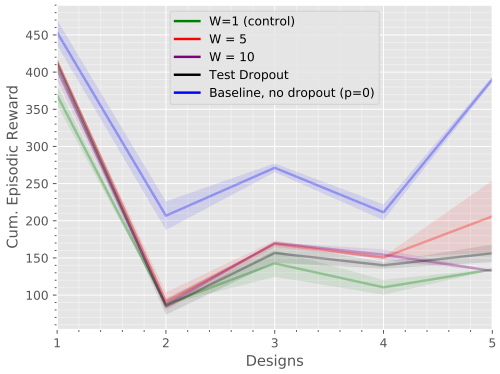

Figure 4.8: Variations of the number of forward passes through the policy loss (W), as well as dropout in test mode in the policy loss. $W = 5, 10$, and dropout in test mode all achieve similarly high results.

4.4 Uncertainty estimates from ensembling Q functions

In this section, we take a detour from our previous approach making the critic in the SAC algorithm probabilistic, and instead focus on a simpler method of acquiring uncertainty estimates. We alter the baseline algorithm to train multiple Q-value functions in parallel, with each model acting as a sample from the posterior. Unlike our previous approach which made the critic a probabilistic network, with this method we can guarantee convergence due to the training occuring exactly the same as in the baseline, only with additional Q-value networks. The exploration-exploitation strategy alternates between $\text{argmax}_\xi(\mu_Q + \sigma_Q)$ and $\text{argmax}_\xi(\mu_Q - \sigma_Q)$, as described in Section 3.5.

We attempt an ensemble of 3 Q-value networks, as shown in Figure 4.9. We see that the exploration is more effective in the ensemble model than the baseline in early designs, with an average cumulative reward of 450 achieved by design 8 which is only

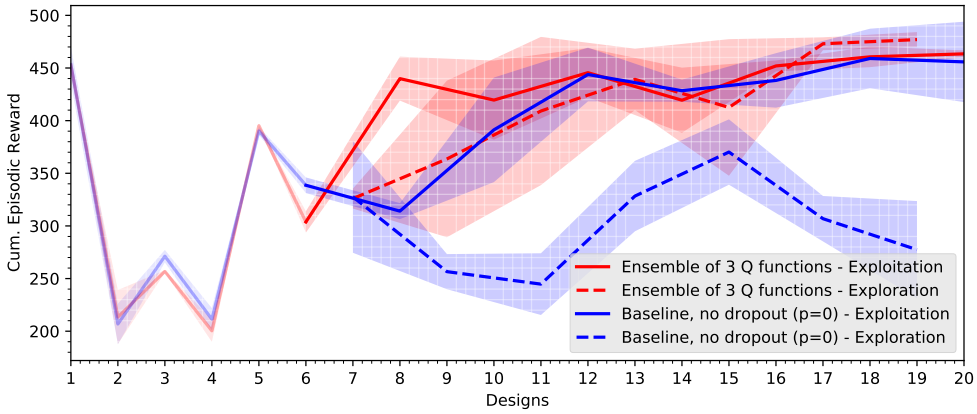

Figure 4.9: The baseline model refers to work by Luck et al. [19], which uses an exploration strategy that selects designs at random, and an exploitation strategy that selects the design with the highest expected reward over start states: $\frac{1}{n}\sum_{s=s_0^1}^{s_0^n} Q(s, a = \pi_{pop}(s,\xi),\xi)$. Our model (red) relies on uncertainty estimates attained by ensembling multiple Q-value functions, with an optimistic exploration strategy that selects $\text{argmax}_\xi(\mu_Q + \sigma_Q)$ and exploitation strategy that selects $\text{argmax}_\xi(\mu_Q - \sigma_Q)$.

reached by the baseline by design 12. However, we see that the ensemble method quickly levels off in cumulative reward and is matched by the baseline, albeit with less variance. While this is a positive result, we will later learn in Section 4.5 that significantly higher reward is attainable in fewer steps, with higher quality uncertainty estimates. In Section 4.5 we will also further describe the baseline results in a more in-depth comparison of results. Since the ensemble method is well theoretically justified, we conclude that its inability to achieve better design-policy pairs is down to the high variance associated with computing the mean or standard deviation of 3 samples, as well as the required reduction in batch size. Since the variance in uncertainty estimates scales with $\frac{1}{\sqrt{n}}$, where n is the number of ensembled Q-value functions, it is infeasible to train enough functions to attain stable uncertainty estimates. In the following section we will instead attempt a new method in which we aim to generate uncertainty estimates in a way that is computationally cheaper, with lower variance.

4.5 Semi-probabilistic approach

Having attempted multiple different strategies to stabilise training probabilistic critic in the SAC algorithm, we realise that the problem is very challenging, and there is no simple solution that will immediately make the SAC algorithm into a stable certainty aware algorithm for exploration. Furthermore, while our attempts still resulted in reasonably effective policies, our goal is to lose as little performance as possible, in return

for a design certainty-aware model. To address these problems, as well as the computational expense of training an ensemble of Q-value functions like in Section 4.4, we use Algorithm 3 to achieve stable learning like in the baseline model, with quality uncertainty estimates using MCDO applied to the Q' network. In this method we only use uncertainty information in regard to selecting the next design, ensuring stability during the training process.

In the following experiment, we trained the Q' probabilistic Q-value function as described, and used it during the design selection process. We use an exploration-exploitation strategy where exploration selects the design with the highest optimistic Q-value estimate across start states: $\text{argmax}_\xi(\mu_Q + \sigma_Q)$ where μ_Q, σ_Q are the network's best point estimate and uncertainty respectively, as described in Section 3.5, and are computed across multiple stochastic passes through Q'. The goal of the exploration phase is to evaluate designs which have high potential. In most cases, the agent will learn that the design was less optimal than the optimistic estimate suggested, but this strategy helps to ensure a wide variety of the state-space is explored, particularly areas that show promise based on the Q' function generalisation. The exploitation phase selects a design according to the following objective function: $\text{argmax}_\xi(\mu_Q - \sigma_Q)$ and ensures that the designs which the agent is certain are effective, are as effective as expected. This also gives the agent a chance to realise if it is overconfident, in which case it will continue to explore more of the design space in the next exploit phase.

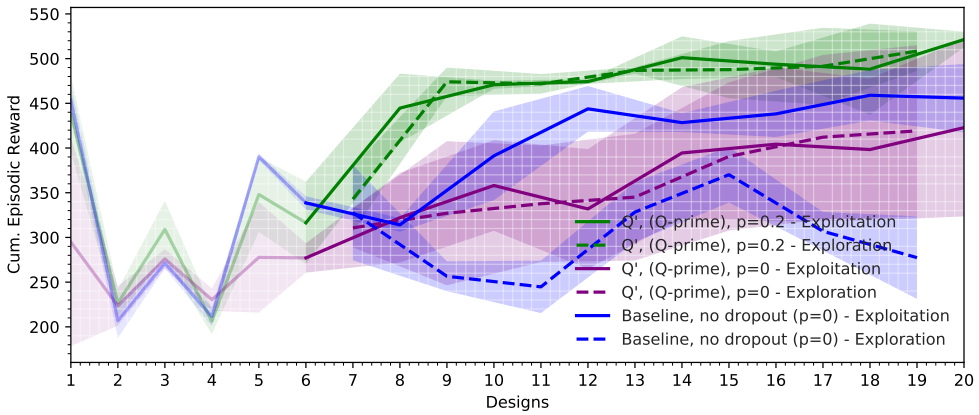

Figure 4.10: Lines in green (our novel approach) achieves the highest cumulative reward in the fewest design evaluations, when compared to using a dropout rate of 0 (purple) and the baseline model (blue). Appendix Figure 6.9 shows the Q' (q-prime) compared to baseline, up to the 50th design evaluation, and shows that our model converges to a higher reward than what is attained by the baseline model. All models are compared across 5 runs.

Q' with $p = 0.2$: In Figure 4.10 we see the cumulative episodic reward attained by three different algorithms. The first 5 designs are the same 5 preset designs that we

have used thus far for testing, but after these designs each algorithm begins its own exploration/exploitation strategy, with dotted lines denoting the exploration and solid lines exploitation. We see the baseline model's exploitation (in solid blue) gradually increase until 450 reward, after which point the performance largely saturates. The dotted blue line, its exploration strategy performs much worse on average because it selects designs from the design space entirely at random, which typically results in ineffective designs. In green, we see our novel proposed Q' model, which uses uncertainty estimates from MCDO in the exploration/exploitation , with $p = 0.2$. As expected, the exploration (dotted green line) performs much better than the baseline's exploration, because the agent purposely selects designs with high potential. This results in gaining more relevant knowledge at a much more rapid rate, resulting better designs even very early on before the 10th design evaluation. Our method not only explores more effectively, resulting in design-policy pairs that achieve higher return earlier on, but manages to achieve a reward of 510 by the 14th design evaluation, which is not surpassed by the baseline model, even in 50 design evaluations, as shown in Appendix Figure 6.9. In Appendix Figure 6.9, we see our Q' method continue to increase performance even after 50 design evaluations, at which point it reaches reward of 550. Our approach also has low variance across runs, showing that the uncertainty estimates are reliable enough to repeatedly use as a heuristic for exploration. In Appendix Figure 6.10, we see that our method also performs better than the baseline in the Walker2 environment, demonstrating the universality of our method, although we note that the difference in performance between the baseline and our novel approach is smaller likely due to the design space being only 5-D, instead of 6-D, making it significantly easier for the baseline's random exploration strategy.

Control experiment with $p = 0$: In purple we show another control experiment, which uses the same Q' model set-up, but has a dropout rate of $p = 0$. With $p = 0$, the model is certainty-unaware, and results in $\sigma_Q = 0$ for all designs. With this in mind, the exploration and exploitation strategy both maximise the same quantity $\text{argmax}_\xi(\mu_Q)$, which is equivalent to the (blue) baseline model running a simple exploitation-only strategy. While this model sometimes achieves very high reward, almost reaching that reached by the Q' model, it does so very rarely and exhibits very high variance across runs. This likely occurs because without a true exploration strategy, the knowledge that is gained by the agent is highly dependant on the design trajectory, and the agent commonly ends up with knowledge gaps. The agent, unaware of its certainty will often be-

lieve it is repeatedly selecting the best design, but unknowingly is missing out on many designs which are in actuality more effective. Since the model with $p = 0$ performs significantly worse to the model with $p = 0.2$, we conclude that it is the uncertainty estimates attained by Monte-Carlo Dropout that allows the agent to effectively explore and quickly converge to good design-policy pairs. In the following sections we will further analyse our results and gain more insight about these uncertainty estimates.

4.5.1 Uncertainty and design selection analysis

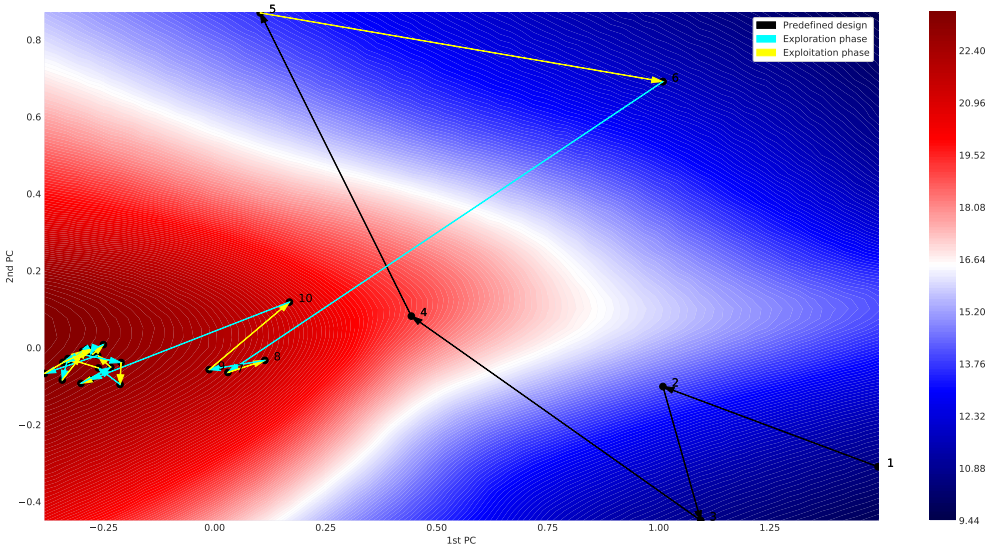

Figure 4.11:
Expected Q value computed by the trained Q' network as the mean over 10 forward passes across 256 start states as follows $\frac{1}{10}\frac{1}{256}\sum_{\bar{\omega}_i \sim \bar{\Omega}}^{i=10}\sum_{s_j \sim s_0}^{j=256} Q(s_j, a, \xi, \bar{\omega}_i)$. This quantity is a measure of the expected effectiveness of the design, also called μ_Q and is used in both the exploration and exploitation phases to compute optimistic and pessimistic estimates.

In Figure 4.11, we get some insight into what the Q-value function looks like. This figure shows a contour plot of the Q values on a 2-D plane in the 6-D design space, which is the plane that best best fits the 30 selected designs by the Q' model with $p = 0.2$. The Q-value shown is the expected Q-value based on 10 forward passes across 256 start states for designs that fall on the plane. Figure 4.12 instead shows the uncertainty estimate, computed as the standard deviation for the same Q estimates on the same plane. In both plots, blue denotes low values and red denotes high values. The first 5 black arrows show the transition to preset designs which are not part of the exploration-exploitation strategy, which are intended to give the agent a head-start and give the Q-value functions some level of generalisation. Yellow arrows show exploitation steps, and cyan arrows show exploration. We see that the algorithm has largely narrowed

down its preferred design within the first 12 steps, which is consistent with Figure 4.10 and Appendix Figure 6.9, beyond which the reward improves very slowly. It's important to note that our visualisation does not fully show the design trajectory, since the real design traversal is in 6 dimensions, outside of the plotted plane. As expected, we see the exploration strategy tend to make bigger, bolder steps, for example from design 7 to 8, 10 to 11, which is where the agent at that time saw high potential based on its optimistic view of the Q' function. On the other hand, the exploitation steps (yellow) tend to be more incremental changes to the design after large design changes, for example, 7 to 8, 9 to 10, due to the agent's pessimistic view.

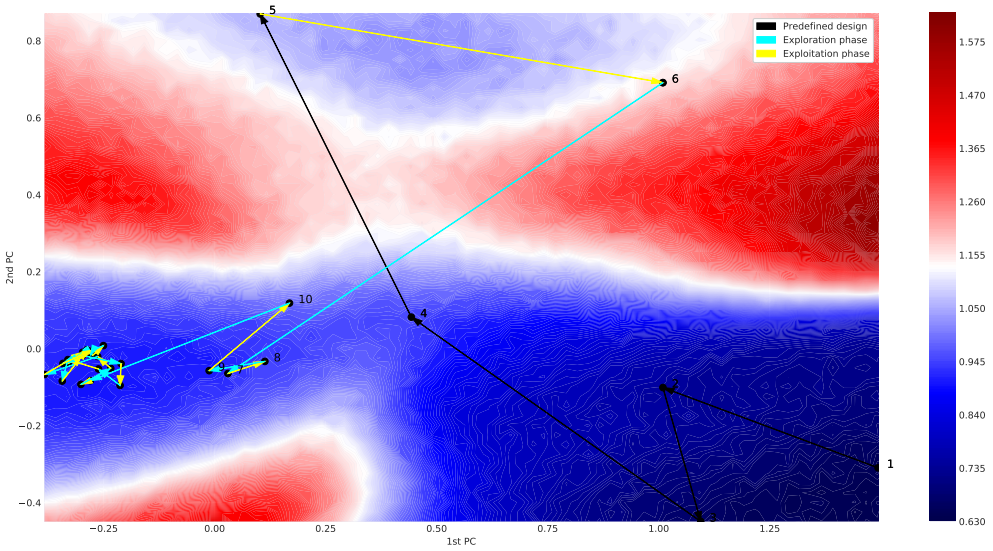

Figure 4.12:
Estimated uncertainty by Monte-Carlo Dropout, known as σ_Q, computed by the trained Q' network as the standard deviation over 10 forward passes across 256 start states, as follows: $\sqrt{\sum_{\bar{\omega}_i \sim \bar{\omega}}^{i=10} (\frac{1}{256} \sum_{s_j \sim s_0}^{j=256} Q(s_j, a, \xi, \bar{\omega}_i) - \mu_Q)/10}$. It describes the model unceratainty for a design based on an approximation of the posterior. σ_Q is added to or subtracted from μ_Q in order to give the optimistic and pessimistic design effectiveness estimates respectively.

In Figure 4.12, we see that in general the uncertainty is the lowest for regions which are near the explored designs. The overall range of uncertainty in the chosen 2-D plane is between 0.45 and 1.35. While we may expect close to 0 uncertainty in the bottom left region where many designs are evaluated, all evaluated designs do not directly lie on the 2D plane, and therefore the Q-value function is relying on some level of generalisation to evaluate points on the plane, leading to non-negligible uncertainty. As the agent converges to its preferred design in the bottom left region (after design 10), we see that there are regions with high uncertainty (in red) above and below it. However, looking at the same regions in Figure 4.11, we see that the expected Q value there is

significantly lower. As a result of this, the agent avoids these designs, since even an optimistic estimate is lower than what the agent is currently exploring. This leads to the agent's convergence in design. To further verify that the uncertainty estimates behave as expected, we plot the difference between uncertainty estimates for a design before and after training on each of the designs across 3 runs in Figure 4.13. As expected, we see that the vast majority of the time (85%) the uncertainty for a design decreases after training and we conclude that the uncertainty estimates work as expected. In some cases the uncertainty increases, which could be due to the design having completely different effectiveness to its expected value μ_Q, or equivalently contradicting information from generalisation across design space.

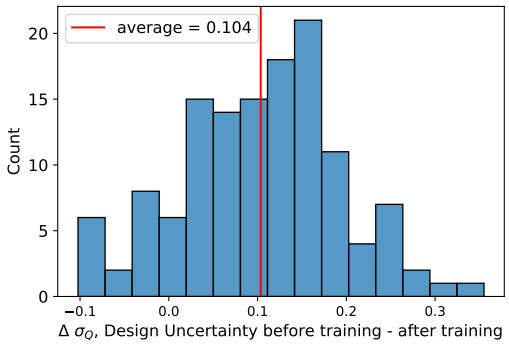

Figure 4.13: Difference in estimated uncertainty for a design σ_Q before and after training on the design. The bias for positive values shows that uncertainty decreases after training.

We show each of the visited designs from Figure 4.11 and 4.12 using the Q' method in Figure 4.14. The even designs are designs chosen by the exploitation strategy, while the odd designs are ones chosen by the exploration strategy. As consistent with the asymptoting of reward in Figure 4.10, as well as the convergence in Figure 4.11, we see that most of the exploration and thus improvements occur in the first 20 designs. Most designs after 20 appear very similar. Typically, we see the odd numbered designs to be quite novel and unique design choices, for example design 19 with long hind legs. Even number designs generally appear similar to ones that have previously been evaluated (8 is similar to 7, 10 is similar to 9).

Figure 4.14: Designs selected during the exploration/exploitation strategy, after the initial 5 pre-set designs are trained on. These designs correspond to the ones selected in Figure 4.11 and 4.12. Even numbered designs are ones selected by the exploitation strategy, while odd numbered designs are ones selected by the exploration strategy.

Chapter 5

Conclusions

5.1 Summary

The main hypothesis that has motivated this project can be summarised as follows:

> Is it possible to improve on the design exploration-exploitation strategy presented by Luck et al. [19] using neural network uncertainty estimates, in a way to achieve more data-efficient co-adaptation of design and controller?

We began to address this hypothesis by performing some analysis on the outputs of the Q-value function as trained in [19], in order to assess whether the existing model had implicitely learned uncertainty information which could be extracted for the purpose of an exploration strategy. While we found some correlations between the variation in Q-values across start-states and novelty of a design, we found it to be unreliable and brittle for the purposes of design exploration.

In our next method, we attempted to explicitely model uncertainty estimates by making the critic in the SAC algorithm [14] probabilistic with the use of Monte-Carlo Dropout in the Q-value network, and by altering the loss functions of the algorithm. However, we found that the addition of stochasticity to the output of the Q-value network resulted in instabilities during the training process and convergence to sub-optimal policies. We attempted multiple strategies to stabilise training, such as by using a more slowly moving Q-target network, tuning the correct number of stochastic forward passes in our loss functions and removing dropout from the final hidden layer. Some of our changes showed signs of stabilisation, achieving lower variance results and increasing performance, while also fixing our diverging Q-value function problem. However, despite the improvements to the stability, our algorithm still converged

to significantly worse performing policies, by approximately 50 reward per design, which is a price not worth paying to generate uncertainty estimates.

As our next step, we trained an ensemble of Q-value functions as an approximation of sampling the posterior, and used the agreement between Q-value functions to model output certainty. While this method showed some positive results, demonstrating that exploration in design space can be done in a slightly more data-efficient manner, the computational expense of the method as well as the high variance in the uncertainty estimates ultimately limited us from scaling our algorithm and achieving even higher cumulative reward from better design-controller pairs.

To address these problems with training an ensemble of Q-value functions, as well as the stability issues which we struggled with when training the SAC algorithm with a probabilistic critic, we came up with a novel semi-probabilistic approach which trains a probabilistic Q-value function which is used in design exploration, in parallel with the original deterministic baseline as in [19], allowing us to keep the benefits of both the ensemble and probabilistic critic approaches. With this approach, we achieved significantly more data-efficient exploration of design space than the baseline model, while also finding highly effective designs in few design evaluations (less than 18) which were not matched by the baseline model even in over 50 designs. We achieved a cumulative reward of 555 in the Half-Cheetah environment, while still increasing after the 50th design evaluation, while the baseline converged to 505. Furthermore, we have analysed some design trajectories, and shown that the agent's design decisions are reasonable and are an efficient method to rapidly converge to effective designs. Finally, we analysed the properties of the uncertainty estimates, and confirmed that they have properties of epistemic uncertainty, and are reasonable grounds for design exploration.

Our data-efficient design exploration shows great promise for deploying and co-adapting morphology and controller in robots in real-life, with morphology changes that are well motivated by a neural network's uncertainty estimates, due to the fewer morphologies that would need to be produced. This would effectively bring the RL loop out of simulation and into the real world, allowing us to entirely avoid the simulation-to-reality gap.

5.2 Future Work

While we have addressed the main hypothesis behind this project, there is more work to be done to further improve our method.

One such example is the implementation of Concrete Dropout [10] within the Q' network. Concrete dropout is a relaxation of the dropout, and is designed for use with Monte-Carlo Dropout, such that the dropout rate p can be tuned during training through an extra term in the loss function. Apart from this saving time on hyperparamter tuning, such a set-up allows to seamlessly tune separate dropout rates for each layer, something which we have not attempted in this project. Doing this in an automatic way has the potential to increase the accuracy of the uncertainty estimates and therefore further improve on the exploration strategy.

Another interesting direction to further improve on the data efficiency of the exploitation-exploration strategy, is to attempt only exploring, without an exploitation strategy. In the baseline model by Luck et al. [19], exploitation is necessary to ensure that the agent prevents delusions, whereby it believes a design is highly effective when in reality it is not. Testing such a design in the exploitation phase challenges the agent's beliefs and updates its knowledge. However, with the addition of uncertainty estimates, this is theoretically unnecessary, since designs which the agent is wrong about, should also be tied to high uncertainty, and therefore would be selected by an exploration strategy regardless. This approach could prevent wasting design evaluations when the agent is certain of a design in the exploitation phase. Such a set-up, with only exploration lends itself well to how exploration is presented in the Free Energy Principle [7], in which exploration is the maximisation of knowledge. We already have some preliminary results in Appendix Figure 6.11 which demonstrates that an agent which selects two exploration steps for every one exploitation step, performs similarly well, or even slightly outperforms our previous best model, which alternates between one exploration and one exploitation step, suggesting that an exploration only approach is a promising direction.

Finally, testing such a system on real robots, where every successive robot is printed and trained in the real world, would be of scientific interest, and would also demonstrate what shortfalls our approach has when applied outside of simulations. It would also demonstrate how reliable uncertainty estimates can be in another setting setting, given our current set-up, and whether the data-efficiency of our approach carries over to the real world.

Chapter 6

Appendix

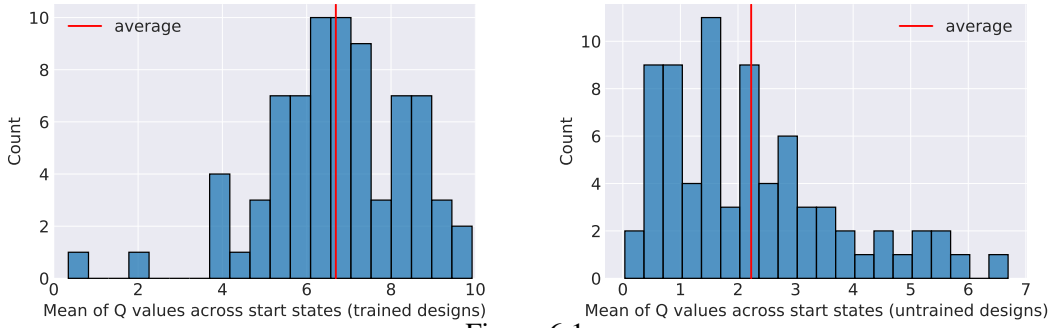

Figure 6.1:
Histogram of mean of Q-values across start states, computed as $\frac{1}{n}\sum_{s_i \in s_{batch}}^{i=n} Q(s_i, a, \xi)$, for trained designs (left) and untrained designs (right). These distributions differ significantly, as they are a function of the choice of designs in both sets.

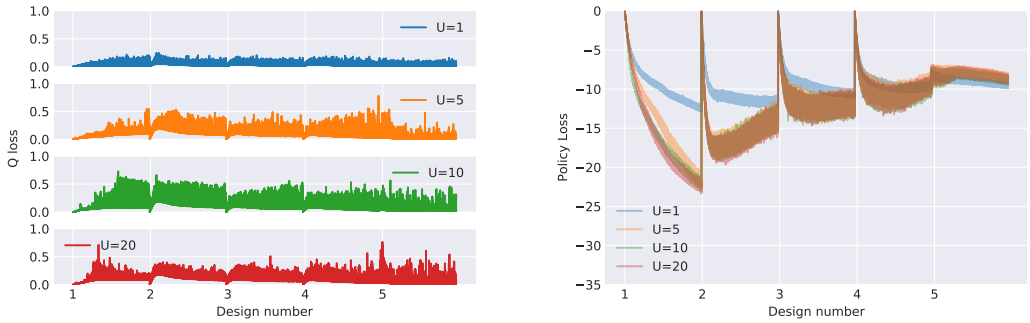

Figure 6.2:
Q-loss and policy loss for differing numbers of forward passes U through the Q-value function in the Q-loss.

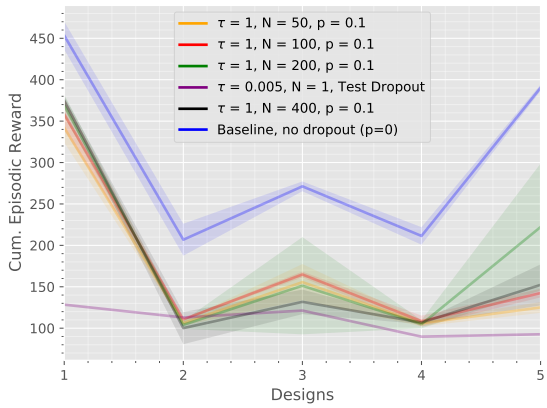

Figure 6.3:
Cumulative reward for 5 runs of 5 pre-set designs with different Q target configurations. All experiments are run with $p = 0.1$.

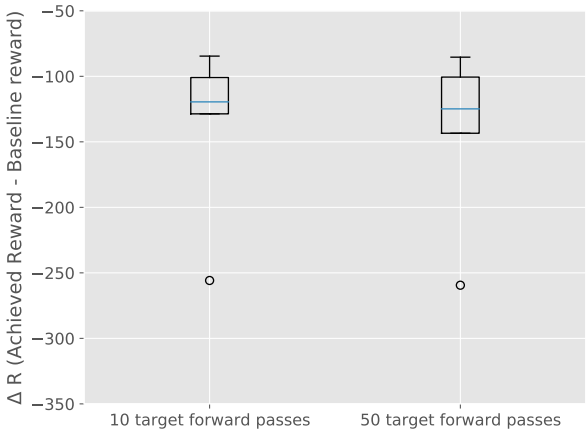

Figure 6.4:
Variation of number of forward passes through the target Q-value network to pass through before setting the mean as the target for the Q-value function. We see that 50 forward passes performs no better than 10 forward passes, suggesting that 10 is a sufficient sample size to achieve a stable target.

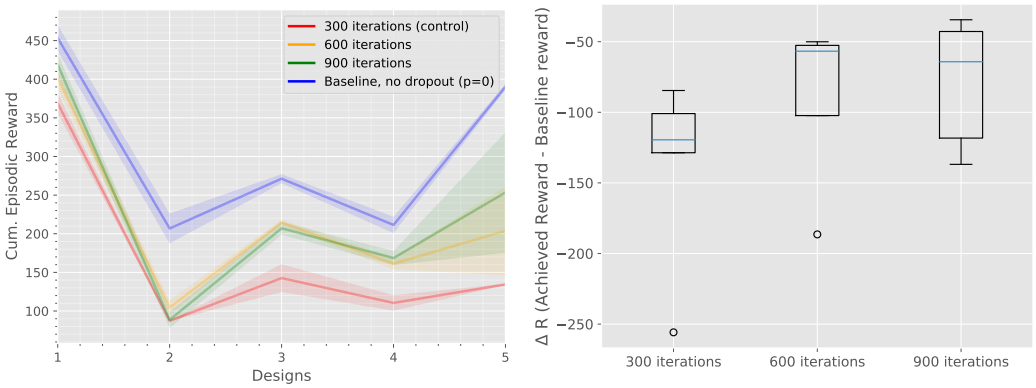

Figure 6.5:
Investigation of number of training iterations on convergence of the probabilistic SAC algorithm. We see that increasing iterations improves the performance to a point, with most benefits being gained at 600 iterations. Statistics shown over 5 runs, of 5 designs each.

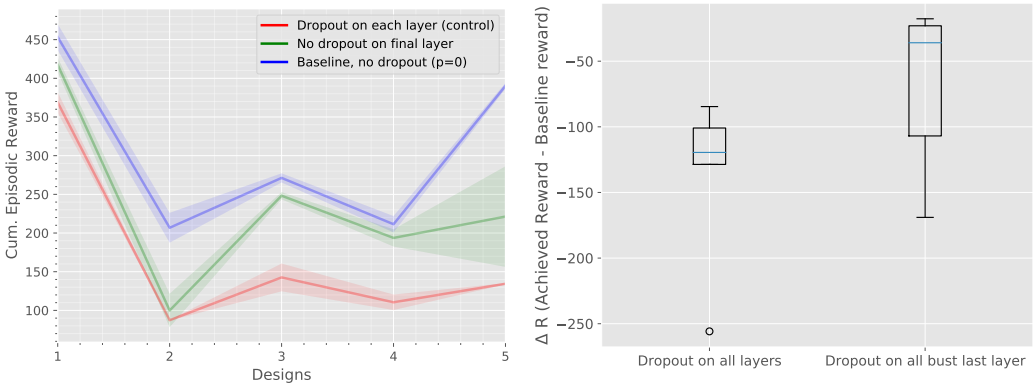

Figure 6.6:
Here we compare using dropout on all layers to all but the final layer. We observe significant improvements to stability when removing dropout from the final layer, but interestingly, we still cannot achieve baseline performance. Statistics shown over 5 runs, of 5 designs each.

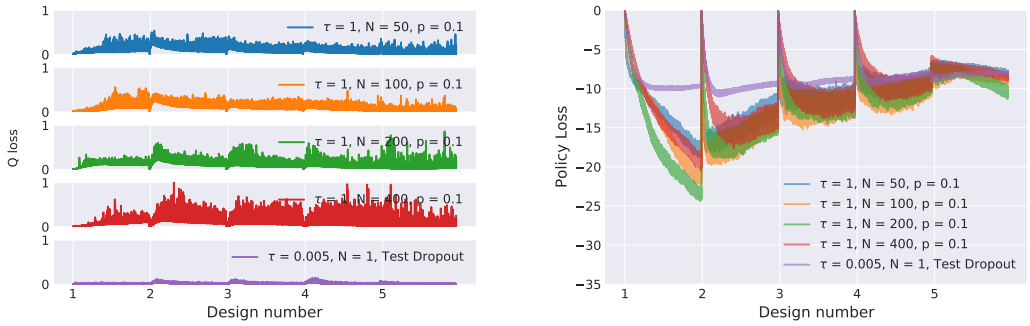

Figure 6.7:
Q-loss and policy loss for differing Q-target configurations. We see healthy Q-losses for all but the test mode of dropout variant of our experiment. We also not that our changes have removed the diverging loss problem.

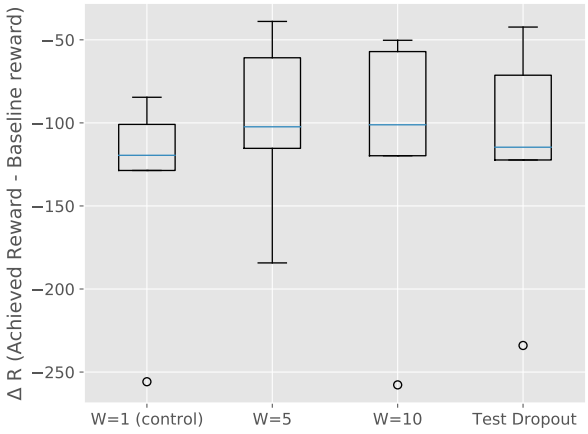

Figure 6.8:
Policy loss changes, where W is the number of forward passes through the Q-value function in the policy loss. We see that $W = 5, 10$, as well as using the test variation of dropout all achieve similar results, although with significant variance between the 5 runs of 5 designs each.

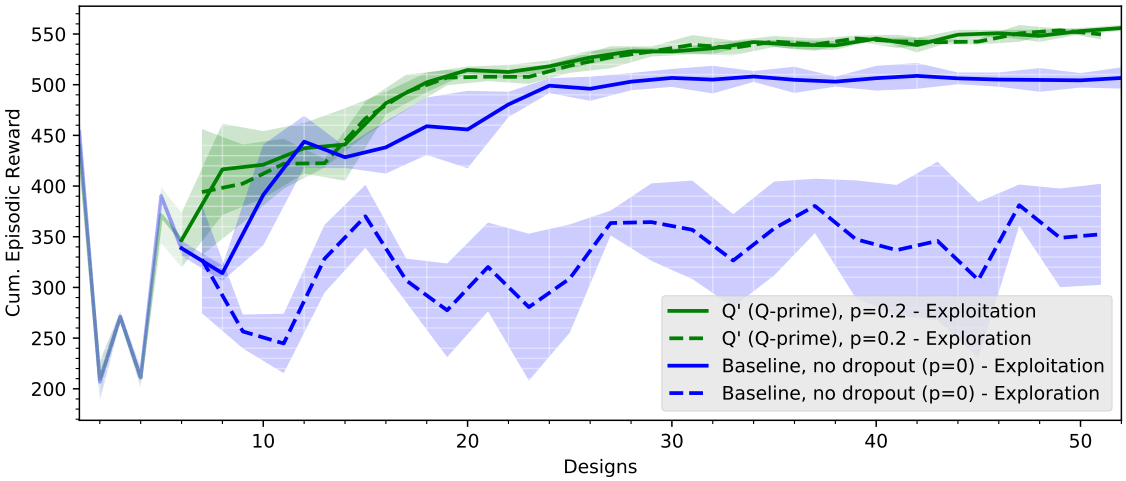

Figure 6.9:
Here we see our Q' method which combines deterministic Q-value function with a Monte-Carlo Dropout probabilistic Q-value function for design exploration, in green. The baseline model is in blue. We see that after 50 design evaluations the baseline appears to have completely converged to a reward of ~ 505, while our approach reaches 555, and appears to still be gradually increasing. Both models are compared over 5 runs each.

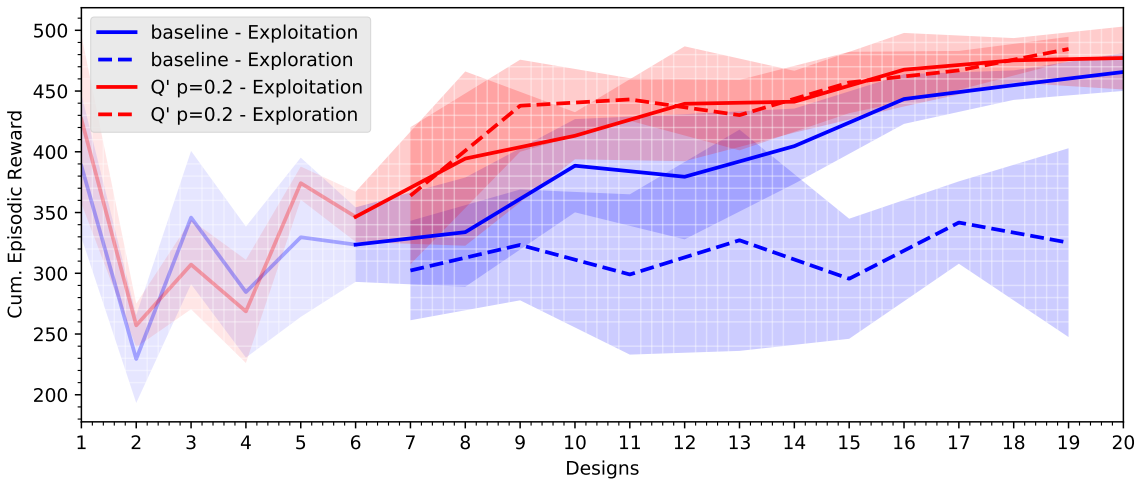

Figure 6.10:
Demonstration of the baseline model and our Q' method in a different environment, known as Walker2. In this environment, a walking stick figure is rewarded for learning effective policies allowing it to walk in a 2-D plane. Like in the Half-Cheetah environment, our method is more data-efficient than the baseline, although we note that the difference is smaller between the two models likely because of the reduction in the number of design dimensions, thereby making even the baseline's random exploration strategy relatively effective.

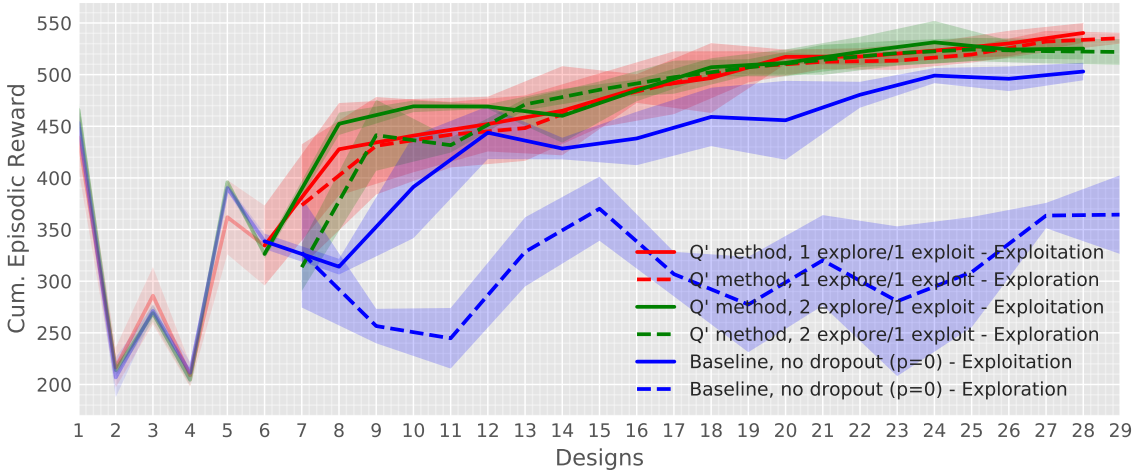

Figure 6.11:
Demonstration of our Q' method, in which we learn a probabilistic Q' in addition to a deterministic Q-value function, but where we explore twice for every exploitation step, instead of once as in the baseline. We achieve very competitive results with this set-up, possibly beating the 1 to 1 strategy, suggesting an exploration only set-up may be effective as potential future work.

Bibliography

[1] Rinaldo C Bertossa. Morphology and behaviour: functional links in development and evolution, 2011.

[2] Charles Blundell, Julien Cornebise, Koray Kavukcuoglu, and Daan Wierstra. Weight uncertainty in neural network. In *International Conference on Machine Learning*, pages 1613–1622. PMLR, 2015.

[3] Mohammad Reza Bonyadi and Zbigniew Michalewicz. Particle Swarm Optimization for Single Objective Continuous Space Problems: A Review. *Evolutionary Computation*, 25(1):1–54, 03 2017.

[4] Andrea Cini, Carlo D'Eramo, Jan Peters, and Cesare Alippi. Deep reinforcement learning with weighted q-learning. *arXiv preprint arXiv:2003.09280*, 2020.

[5] Francesco Corucci, Marcello Calisti, Helmut Hauser, and Cecilia Laschi. Novelty-based evolutionary design of morphing underwater robots. In *Proceedings of the 2015 annual conference on Genetic and Evolutionary Computation*, pages 145–152, 2015.

[6] Erwin Coumans and Yunfei Bai. Pybullet, a python module for physics simulation for games, robotics and machine learning. 2016.

[7] Karl Friston. The free-energy principle: a unified brain theory? *Nature reviews neuroscience*, 11(2):127–138, 2010.

[8] Yarin Gal. *Uncertainty in Deep Learning*. PhD thesis, University of Cambridge, 2016.

[9] Yarin Gal and Zoubin Ghahramani. Dropout as a bayesian approximation: Representing model uncertainty in deep learning. In *international conference on machine learning*, pages 1050–1059. PMLR, 2016.

[10] Yarin Gal, Jiri Hron, and Alex Kendall. Concrete dropout. *arXiv preprint arXiv:1705.07832*, 2017.

[11] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016. `http://www.deeplearningbook.org`.

[12] Agrim Gupta, S. Savarese, S. Ganguli, and Li Fei-Fei. Embodied intelligence via learning and evolution. *ArXiv*, abs/2102.02202, 2021.

[13] David Ha. Reinforcement learning for improving agent design. *Artificial life*, 25(4):352–365, 2019.

[14] Tuomas Haarnoja, Aurick Zhou, Pieter Abbeel, and Sergey Levine. Soft actor-critic: Off-policy maximum entropy deep reinforcement learning with a stochastic actor. In *International conference on machine learning*, pages 1861–1870. PMLR, 2018.

[15] Henry J Kelley. Gradient theory of optimal flight paths. *Ars Journal*, 30(10):947–954, 1960.

[16] James Kennedy and Russell Eberhart. Particle swarm optimization. In *Proceedings of ICNN'95-international conference on neural networks*, volume 4, pages 1942–1948. IEEE, 1995.

[17] Sylvain Koos, Jean-Baptiste Mouret, and Stéphane Doncieux. Crossing the reality gap in evolutionary robotics by promoting transferable controllers. In *Proceedings of the 12th annual conference on Genetic and evolutionary computation*, pages 119–126, 2010.

[18] Sylvain Koos, Jean-Baptiste Mouret, and Stéphane Doncieux. The transferability approach: Crossing the reality gap in evolutionary robotics. *IEEE Transactions on Evolutionary Computation*, 17(1):122–145, 2012.

[19] Kevin Sebastian Luck, Heni Ben Amor, and Roberto Calandra. Data-efficient co-adaptation of morphology and behaviour with deep reinforcement learning. In Leslie Pack Kaelbling, Danica Kragic, and Komei Sugiura, editors, *Proceedings of the Conference on Robot Learning*, volume 100 of *Proceedings of Machine Learning Research*, pages 854–869. PMLR, 30 Oct–01 Nov 2020.

[20] Kevin Sebastian Luck, Joseph Campbell, Michael Andrew Jansen, Daniel M Aukes, and Heni Ben Amor. From the lab to the desert: Fast prototyping and learning of robot locomotion. *arXiv preprint arXiv:1706.01977*, 2017.

[21] Scott McLachlan, Kudakwashe Dube, Graham A Hitman, Norman E Fenton, and Evangelia Kyrimi. Bayesian networks in healthcare: Distribution by medical condition. *Artificial Intelligence in Medicine*, 107:101912, 2020.

[22] Tønnes F Nygaard, Charles P Martin, Eivind Samuelsen, Jim Torresen, and Kyrre Glette. Real-world evolution adapts robot morphology and control to hardware limitations. In *Proceedings of the Genetic and Evolutionary Computation Conference*, pages 125–132, 2018.

[23] David E Rumelhart, Geoffrey E Hinton, and Ronald J Williams. Learning representations by back-propagating errors. *nature*, 323(6088):533–536, 1986.

[24] Charles Schaff, David Yunis, Ayan Chakrabarti, and Matthew R Walter. Jointly learning to construct and control agents using deep reinforcement learning. In *2019 International Conference on Robotics and Automation (ICRA)*, pages 9798–9805. IEEE, 2019.

[25] Karl Sims. Evolving virtual creatures. In *Proceedings of the 21st annual conference on Computer graphics and interactive techniques*, pages 15–22, 1994.

[26] Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. Dropout: a simple way to prevent neural networks from overfitting. *The journal of machine learning research*, 15(1):1929–1958, 2014.

[27] Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning: An Introduction*. A Bradford Book, Cambridge, MA, USA, 2018.

[28] Andrew Trask. *Grokking Deep Learning*. Manning Publications Co., USA, 1st edition, 2019.

[29] Ronald J Williams. Simple statistical gradient-following algorithms for connectionist reinforcement learning. *Machine learning*, 8(3):229–256, 1992.