Curriculum Learning through Distilled Discriminators

Master's Thesis submitted to the
Faculty of Informatics of the *Università della Svizzera Italiana*
in partial fulfillment of the requirements for the degree of
Master of Science in Informatics

presented by
Rahul Siripurapu

under the supervision of
Prof. Dr. Jürgen Schmidhuber
co-supervised by
Louis Kirsch

August 2020

I certify that except where due acknowledgement has been given, the work presented in this thesis is that of the author alone; the work has not been submitted previously, in whole or in part, to qualify for any other academic award; and the content of the thesis is the result of work which has been carried out since the official commencement date of the approved research program.

Rahul Siripurapu
Lugano, 28 August 2020

To my parents

From falsehood lead me to truth
Brihadāranyaka Upanishad

Abstract

Automated Curriculum Learning (ACL) is now a key focus of Deep Reinforcement Learning (DRL) research. This is because traditional methods of training agents using Human Engineered Reward Functions (HERFs) work with difficulty in simulation and do not generalise well to real-world settings. Apart from the engineering effort in hand-crafting these functions, they tend to have spurious optima and are challenging to learn from. Current research leans towards using *learned* reward functions which do not share the same problems since they can be learned from the data automatically. Recent works show that it is possible to iteratively train both the RL agents and the reward functions, in a mutually improving fashion, with the adapting reward functions automatically forming a curriculum. These methods aim to discover skills without supervision. This master thesis provides an overview of the core ideas involved in implementing such methods. It demonstrates the generalisation advantages that come with retraining agents in such settings. This retrained agent is then used as a baseline to demonstrate that even better results can be obtained via the use of a curriculum. Given a neural network based reward function for a difficult task, we show how to automatically construct a task of intermediate difficulty by distilling the function into a narrower network. The distilled function and the original function together form a curriculum.

Acknowledgements

First and foremost, I would like to thank my supervisors, Louis Kirsch and Prof. Dr. Jürgen Schmidhuber for the opportunity, the constant guidance and the computational resources without which this thesis would not have been possible. I am very grateful to Louis for all our discussions which kept me motivated throughout. I would like to thank my parents for their support and encouragement. I would also like to thank everyone at NNAISENSE, and particularly Dr. Rupesh Srivastava, Dr. Jan Koutník and Dr. Faustino Gomez for an amazing internship opportunity, which helped me learn a lot of the skills necessary for this thesis. Further, I am very grateful to all of the faculty at USI, for everything that I have learned. Particularly Prof. Dr. Jürgen Schmidhuber and Prof. Dr. Michael Bronstein. Their lectures always had me sitting on the edge of my seat! Finally, I thank my friends and family for always being there for me.

Contents

Figures

Chapter 1

Introduction

1.1 Overview

It is a common experience in life that people are often able to judge whether a movie is good or bad, but unable to teach someone exactly how to make a good movie. People are much better at discriminating or judging than instructing. How can someone learn to make a good movie in a situation where you only have a judge? This also happens to be very relevant to Artificial Intelligence. It is often much easier to judge the actions of an AI than to train it to perform well on any given task, be it playing a game like Chess or maneuvering a drone. This thesis tackles this problem in the context of open-ended learning. In open-ended learning, we want our agents to learn to do everything possible in an environment with no supervision. One of the common methods for open-ended learning is to start with an AI agent, that is used to train an AI judge or discriminator, which in turn trains the *next* AI agent and so on (similar to adversarial curiosity[Schmidhuber, 2020b]). This is built on the assumption that, like humans, AI agents may learn suboptimal policies. Thus after some steps of learning, it may be efficient to start again rather than unlearn and relearn [Ash and Adams, 2019][Igl et al., 2020]. However, a problem with this iterative method is that, as you repeat the process, the discriminator may become more and more complex, and the newly instantiated agents would find it more and more difficult to learn. Improvements would diminish, and the process would halt. One solution to this is to use a meta-learning agent that learns how to learn[Jabri et al., 2019]. However, this also faces the same problem, but on a meta-level. A better solution to this problem is presented in the form of a curriculum [Elman, 1993][Schmidhuber, 2012]. This thesis shows that it is possible to use a simple principle to automatically construct tasks of intermediate difficulty for such an agent, which helps it not only to learn to achieve the task but also generalize well to other "test" tasks.

1.2 Methodology

We will consider the specific case of robotics tasks where we have an agent in a simulated physical environment. The Multi-Joint dynamics with Contact or MuJoCo simulator[Todorov et al., 2012] is used to simulate a flat, open environment where the agent in the form of a cheetah or an ant, must learn to perform complex skills like running, hopping, flipping, etc.

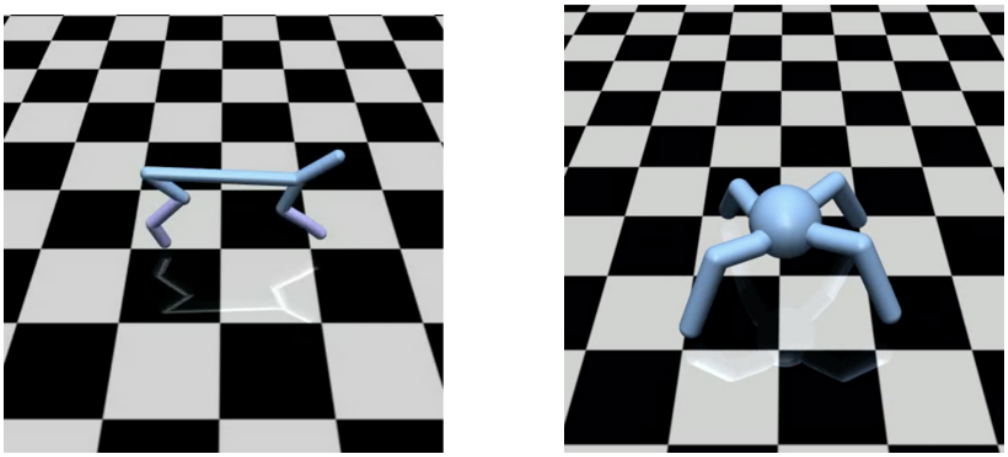


Figure 1.1. Left is the HalfCheetah-v1 Environment, and Right is the Ant-v1 Environment.

The agent's actions are decided by a neural network that knows only the current state of the agent and the skill that it must execute. Another neural network acts as a discriminator which determines whether the skill was executed. This is used to implement an algorithm called Diversity Is All You Need (DIAYN)[Eysenbach et al., 2018], which can learn multiple skills without supervision. This discriminator can then be used to train a new agent from scratch and this process can be continued. We show that distilling the discriminator first into a lower width network makes it much easier for the agent to learn. Distilling is merely the process of compressing the information of one network into another[Schmidhuber, 1992]. *This provides a simple curriculum that helps the agent to achieve the original task and also generalise better to other test tasks.*

1.3 Outline

First, in the chapter on Variational Inference (chapter 2), we discuss a core aspect of learning that underlies Maximum Entropy RL[Eysenbach and Levine, 2019][Levine, 2018] and Empowerment[Salge et al., 2013][Gregor et al., 2016]. In the chapter on Neural Networks (chapter 3), we discuss two of the main principles behind using neural networks. The fact that single-layer neural networks can approximate any continuous function to arbitrary precision[Hornik et al., 1989][Cybenko, 1989] and more precisely the effect that network width has on the approximations, form the basis of the entire method. The chapter on Reinforcement Learning (chapter 4) discusses the rationale behind the framework of using agents and the concepts necessary to implement the agents. Maximum Entropy Reinforcement Learning (chapter 5) discusses the advantages of using a maximum entropy objective, which is very useful in our setting. Finally, the chapter on Automated Curriculum Learning (chapter 6) briefly outlines how curriculum learning has been formulated and provides context for how this work relates to the field.

Next, we will discuss the algorithms used, primarily DIAYN, the distillation, retraining, and fine-tuning algorithms and the MuJoCo environments and benchmark tasks.

Finally, we experimentally validate the benefits of retraining (chapter 11). In the chapter on

curricula (chapter 12), we show that a distilled discriminator serves as a task of intermediate difficulty and can be used to automatically create a curriculum.

1.4 Contributions

This thesis demonstrates how retraining an RL agent can produce better results[1] in terms of generalization.

A method to automatically build a curriculum is introduced, and two of the key benefits of using a curriculum are demonstrated. The algorithm achieves a higher reward on the task it trains for (compared to the retraining as a baseline) and also generalises better to unseen test tasks.

[1] Concurrent work also demonstrates this albeit in different settings. [Portelas et al., 2020b][Igl et al., 2020]

Chapter 2

Variational Inference

Variational inference is a way to formulate a statistical inference problem as an optimisation problem. As with all variational methods, this is useful in situations where we have good optimisation algorithms but no good ways to solve the original problem.

2.1 Bayes' Rule

In the simplest terms, a statistical inference problem can be defined as follows:

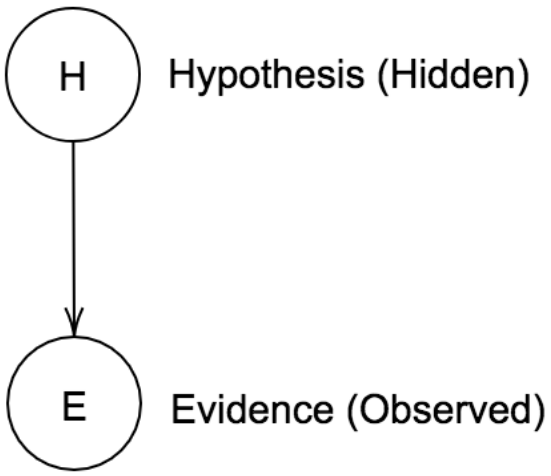


Figure 2.1. The statistical inference problem

We have two random variables H (hypothesis) and E (evidence), where E depends on H. E is observable while H is hidden. The goal is to infer the correct hypothesis given the evidence. Using *Bayes' theorem,* we get the following:

$$p(H \mid E) = \frac{p(E \mid H)p(H)}{p(E)} \tag{2.1}$$

To understand formula (2.1) , consider our evidence to be a mysterious circle in a wheat field. There are two major hypotheses one may arrive at.

1. An alien spaceship landed in the field recently
2. A prankster did it for fun.

The term on the left p(H|E) is referred to as the *Posterior:* The probability of the hypothesis given the evidence at hand. How likely is it, that the crop circle was caused by an alien spaceship? The term on the right p(E|H) is the *Likelihood:* The probability of the evidence given the hypothesis. Here we see that both alien spaceships and pranksters are equally capable in stamping out a circle in a wheat field. p(H) is our *Prior:* Our initial estimate of the probability of the hypothesis. We have probably never seen a spaceship but pranksters are very common. The denominator, the probability of evidence, is computed as follows:

$$p(E) = \sum_{H} p(E \mid H)p(H) \tag{2.2}$$

Since the likelihood and the denominator terms would be the same for both hypotheses, we can tell that the probability of a prankster is quite high, using Bayes' rule.

2.2 Sampling vs Variational Methods

In complicated scenarios, we may have countless hypotheses and it may be difficult to compute the likelihoods for each one. Further, computing the probability of evidence[1], for which we must integrate over all possible hypotheses, is intractable. Hence we need a way to approximate the posterior. Monte Carlo methods like Gibbs Sampling, propose to simulate the process multiple times and estimate the posterior via counting. Such sampling-based estimates are unbiased but have high variance for complicated processes that require unreasonably large sample sizes. Variational inference solves this problem by assuming a known parametric density Q and then optimising Q to be as close as possible to the desired posterior. This will be a biased approximation because of our prior assumption but has a much lower variance and is also much faster to converge.

2.3 Information Projection

Now, to optimise anything, it is essential to have a measure. The most common measure used is the reverse Kullback-Leibler (KL) divergence.

$$KL\left(Q_\phi(H \mid E) || P(H \mid E)\right) = \sum_{h \in H} q_\phi(h \mid e) \log \frac{q_\phi(h \mid e)}{p(h \mid e)} \tag{2.3}$$

The KL divergence is an asymmetric distance measure between probability distributions. Here we try to find the Q that is closest to the posterior P in terms of the above measure. This

[1] Also called the Partition function

is also known as the *Information projection* of P onto the set of parametrised distributions Q. Informally, it measures the amount of additional information in Q. In terms of information theory, one can easily see that this is the additional number of bits required to encode samples from Q using a code optimised for P. Hence to minimise this, Q would try to force itself to become zero wherever P is small (i.e. codes for P are large)

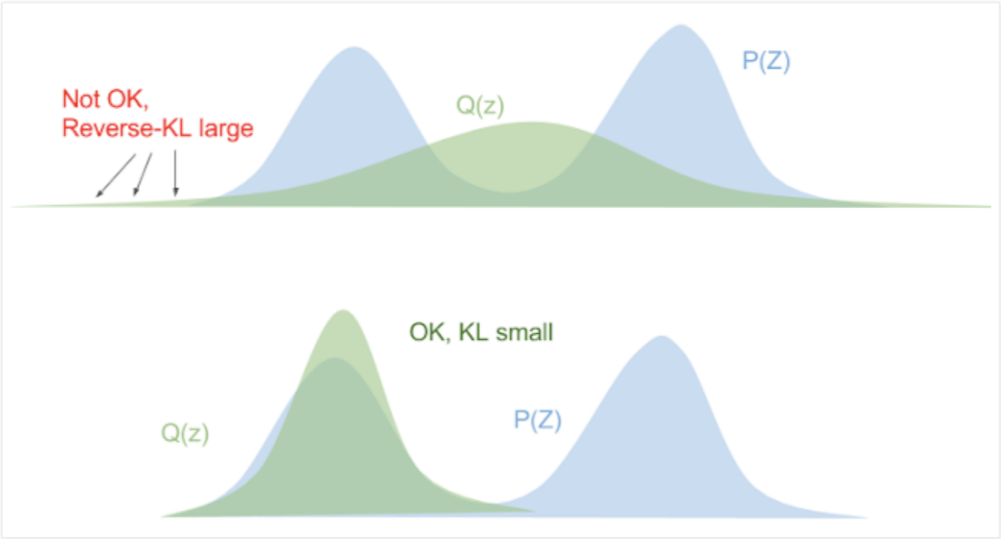


Figure 2.2. Reverse KL. Image taken from https://blog.evjang.com/2016/08/variational-bayes.html

We can see that the reveres KL underestimates the entropy and simply locks on to one of the modes as Q is restricted to a set of gaussians.

2.4 Variational Free Energy

Expanding the KL term as follows, we get an interesting interpretation:

$$\begin{aligned} KL(Q_\phi(H \mid E) \| P(H \mid E)) &= \sum_{h \in H} q_\phi(h \mid e) \log \frac{q_\phi(h \mid e) p(e)}{p(h,e)} \\ &= \sum_{h \in H} q_\phi(h \mid e) \left(\log \frac{q_\phi(h \mid e)}{p(h,e)} + \log p(e) \right) \\ &= \left(\sum_{h \in H} q_\phi(h \mid e) \log \frac{q_\phi(h \mid e)}{p(h,e)} \right) + \left(\sum_{h \in H} \log p(e) q_\phi(h \mid e) \right) \\ &= \left(\sum_{h \in H} q_\phi(h \mid e) \log \frac{q_\phi(h \mid e)}{p(h,e)} \right) + \left(\log p(e) \sum_{h \in H} q_\phi(h \mid e) \right) \\ &= \left(\sum_{h \in H} q_\phi(h \mid e) \log \frac{q_\phi(h \mid e)}{p(h,e)} \right) + \log p(e) \end{aligned} \tag{2.4}$$

$$
\begin{aligned}
-\sum_{h\in H} q_\phi(h \mid e) \log \frac{q_\phi(h \mid e)}{p(h,e)} &= \log p(e) - KL(Q_\phi(H \mid E) \| P(H \mid E)) \\
\mathscr{L} &= \log p(e) - KL(Q_\phi(H \mid E) \| P(H \mid E))
\end{aligned}
\tag{2.5}
$$

We can see that $\mathscr{L}$ lower bounds the log probability of the evidence as the KL term on the right is non-negative. Hence $\mathscr{L}$ is also referred to as the Evidence Lower Bound (ELBO) or the *negative* Variational Free Energy. Hence by maximising this lower bound, we can minimize the KL divergence term. We can rewrite the ELBO as follows:

$$
\begin{aligned}
-\sum_{h\in H} q_\phi(h \mid e) \log \frac{q_\phi(h \mid e)}{p(h,e)} &= \mathbb{E}_{h\sim Q_\phi(H|E)}\left[\log \frac{p(h,e)}{q_\phi(h \mid e)}\right] \\
&= \mathbb{E}_{h\sim Q_\phi(H|E)}\left[\log \frac{p(e \mid h)p(h)}{q_\phi(h \mid e)}\right] \\
&= \mathbb{E}_{h\sim Q_\phi(H|E)}[\log p(e \mid h)] + \sum_{h\in H} q_\phi(h \mid e) \log \frac{p(h)}{q_\phi(h \mid e)} \\
&= \mathbb{E}_{h\sim Q_\phi(H|E)}[\log p(e \mid h)] - KL(Q_\phi(H \mid E) || P(H))
\end{aligned}
\tag{2.6}
$$

The first term is commonly called the *accuracy* term while the second is referred to as the *complexity* term. Hence maximising $\mathscr{L}$ is essentially maximising the likelihood of the evidence given the hypothesis (prediction accuracy) while simultaneously minimising the additional information (complexity) in the posterior with respect to the prior on the hypothesis.

Another way to look at this in terms of Free Energy, is to see that maximising the ELBO is the same as minimising the Free Energy. The Free Energy can be seen as a sum of the Surprise and the complexity term. Since the complexity term is always positive, *minimising Free Energy leads to a minimisation of surprise*!

Chapter 3

Neural Networks

In recent years, Neural Networks have become massively popular due to their wide-ranging applicability and performance scale-ups with compute. Their wide ranging applicability stems from the fact that they are Universal Approximators[Hornik et al., 1989][Cybenko, 1989]. Further, by incorporating priors such as *compositionality* and *smoothness*, they are efficient to implement and optimise.

3.1 Approximating one-dimensional functions

Given a function $y = f(x)$, we can use a neural network to approximate the function as $y = f'(x;\theta)$ where θ are parameters corresponding to the Neural Network. We want to ensure that we can approximate any function f, say the one in figure 3.1:

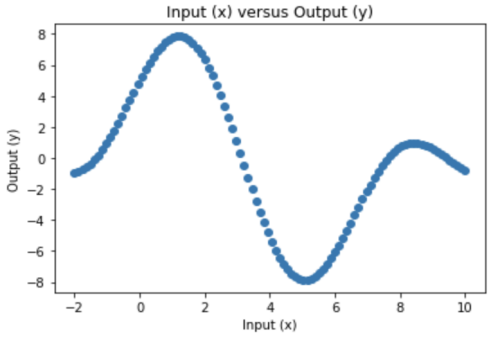


Figure 3.1. A discrete, finite precision representation of a function to be approximated

We see that we need two key elements. First, the curve is non-linear, so we need a non-linearity in our network. Second, the curve is smooth, so we would also require smoothness. To get the simplest possible network, we can use the simplest smooth curve, a line: $y = w.x + c$. Followed by the simplest nonlinearity, a Rectified Linear Unit (ReLU): $y = max(x, 0) = ReLU(x)$.

Using one input unit x, one hidden unit $ReLU(w_1.x + c_1)$ and one output unit $w_2.x + c_2$ as shown in figure 3.3 , we can get the approximation shown in figure 3.4.

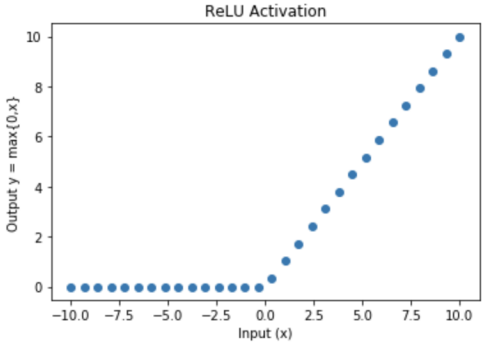


Figure 3.2. ReLU activation

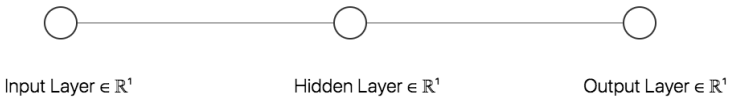


Figure 3.3. A network with 1 hidden unit

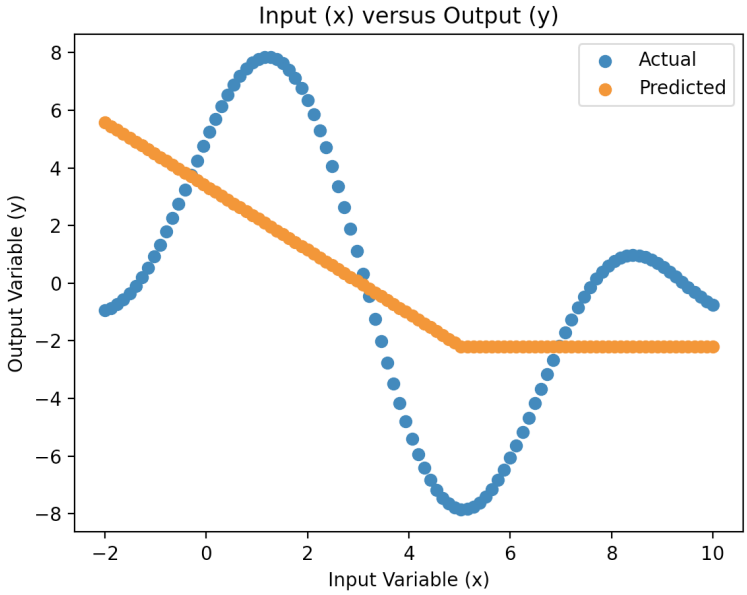


Figure 3.4. An approximation using a 1 hidden unit ReLU network

The approximation is quite bad. Now the one-layer ReLU network has only four parameters and hence controls only four things. First, it can control the slope of the slanting line, by the output layer weight w_2. Second, it can control the y position of the curve, using the output bias c_2. Third, it can control the scale along the x axis. This is given by the hidden layer weight w_1.

Fourth, it can control the location of the elbow along the x axis, through the hidden layer bias c_1. Hence, this is the best it can do.

To get a better approximation, we can simply increase the number of such units and take a linear combination of their outputs. Figure 3.5 shows the approximation for a 100 hidden unit network.

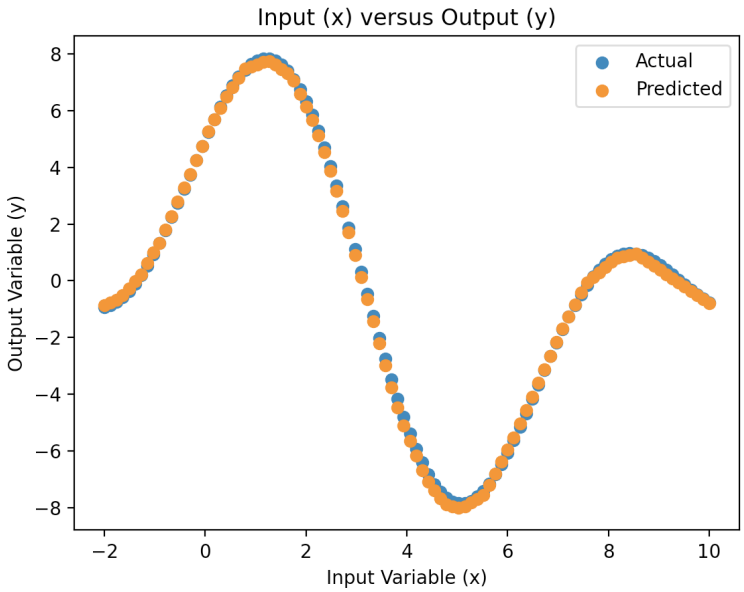


Figure 3.5. An approximation using a 100 hidden unit ReLU network

We can see that the approximation is very good. To understand how the network is able to do this intuitively, we can start by considering the following. Any function such as the one above can be approximated by a set of rectangular bump functions as shown in figure 3.6

Here each bump has a width of 0.04, and we can see that this is quite close to the curve.

Now it is possible to use four ReLU hidden units to approximate one such bump. First let's approximate the first half of the bump, i.e. a step function. To do this we can use two ReLU units with extremely large input weights and oppositely signed output weights with the biases shifted by a small number to get the output in figure 3.7.

We can now produce the inverse of this curve using two more ReLUs and then combine the two curves to get our bump function. The features of our bump function can be precisely controlled by the weights and biases of the neural network shown in figure 3.8.

Thus it's not hard to see how using many more ReLUs, it's possible to approximate any curve in this fashion. The sloped regions aren't a problem as adjacent sloping sections can be perfectly overlapped to produce a smooth transition from one step to another. (Reasonably assuming the function is bounded) However this is a very inefficient way to approximate the function and NNs trained with gradient descent usually find much more efficient solutions.

The above discussion only outlines how NNs are universal approximators for single input single, output functions. If we had multiple inputs and multiple outputs. We can extend the same reasoning as shown in the following section.

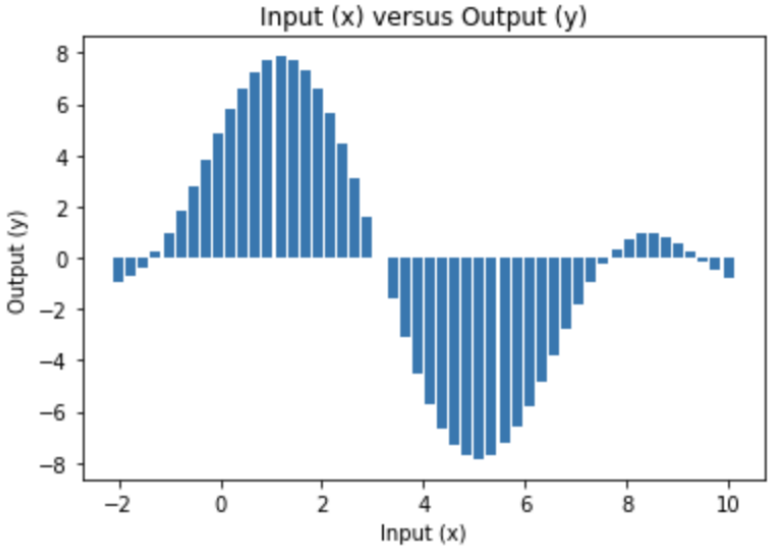


Figure 3.6. An approximation using bump functions

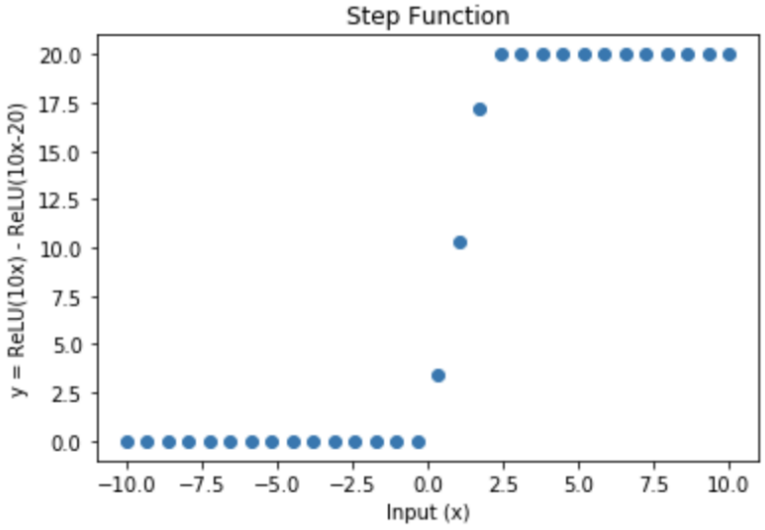


Figure 3.7. A step function using a 2 hidden unit ReLU network

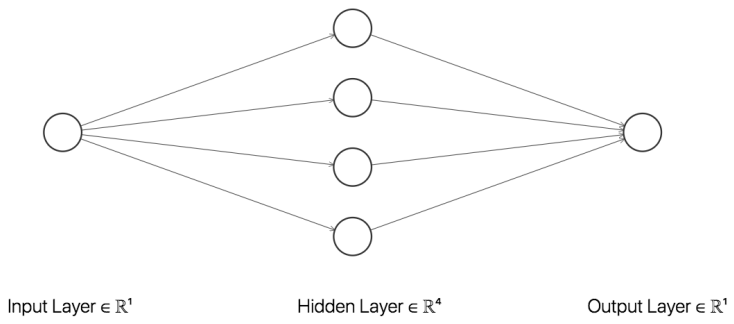


Figure 3.8. A four hidden unit ReLU network can produce a bump function

3.2 Approximating multi-dimensional functions

First for multiple inputs. We simply need to be able to approximate multi-dimensional bumps and then place them at arbitrary locations to be able to approximate a multi-dimensional curve. The two dimensional version of the bump function produced by four hidden units, when only one of the inputs(x) affects the output is shown in figure 3.9.

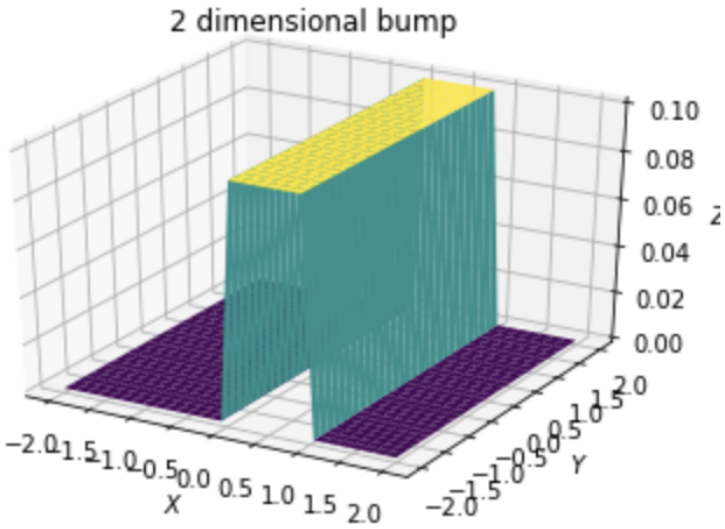


Figure 3.9. A 2 dimensional bump function

By having both inputs affect the output, we can rotate this bump along any angle in the x-y plane. We can combine such bumps along two perpendicular angles to create a square bump at a desired location of intersection. To ensure that this function is zero everywhere else, we can merely adjust the bias of the output neuron to be negative enough to nullify any output outside the square bump.

Thus it is possible to approximate any multi-dimensional curve. For multiple outputs we can just treat each dimension as one such multi-dimensional curve to be approximated. Again,

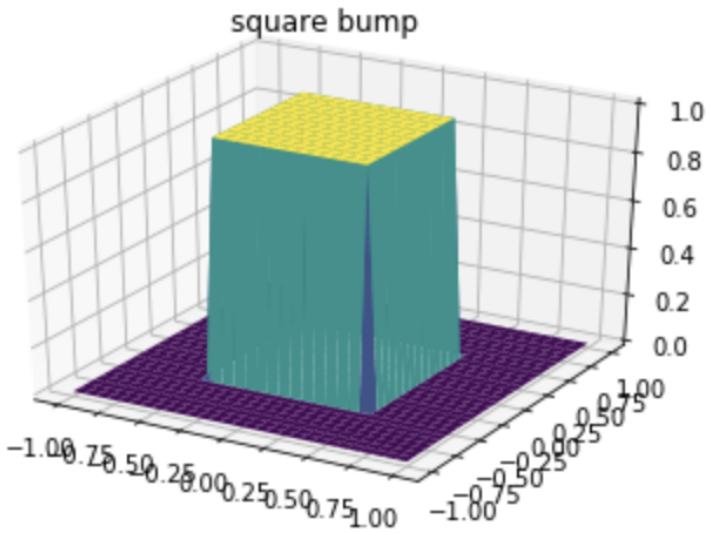


Figure 3.10. A square bump function

NNs trained by gradient descent tend to find more efficient solutions than this. Now that we've understood that we can approximate a multi-dimensional function with a single-layered ReLU network, we need to understand the function of the two main hyper-parameters used in NN design: the width and the depth.

3.3 Width vs Depth

Clearly, wider and deeper networks are bound to fit more complex functions than their narrower and shallower counterparts. However, the complexity that comes with width is different from the complexity that comes from depth, and they are not always interchangeable. Each added layer in a network, is merely a linear transformation (matrix multiplication) followed by a translation (bias addition) and a rectification. Thus, depth represents the number of such non-linear operations performed. The width of the network represents the dimensionality of the space in which these operations are being performed.

Say we had a network of width two and we need to classify the data shown in figure 3.11 into two sets (The final output is 1D, say positive is blue and negative is orange). The final output neuron simply functions as a line dividing the two sets. *The blue patch of space in the centre cannot be moved by continuous transformations in two dimensions out of the orange circular patch*. Thus it is not possible to divide it with a line and the network is not able to generalise.

In three dimensions however, it is easy to push the blue patch out along the third dimension and then divide with a plane along the third dimension.

Figure 3.12 shows the effects of changing the width. On the left is the output of a single layer, 3 hidden neuron network while on the right is the output of a 2-wide,6-deep network. Now the interesting thing to note here is that there is more blue area in the 2-wide model. This will prove to be a key aspect we rely on later.

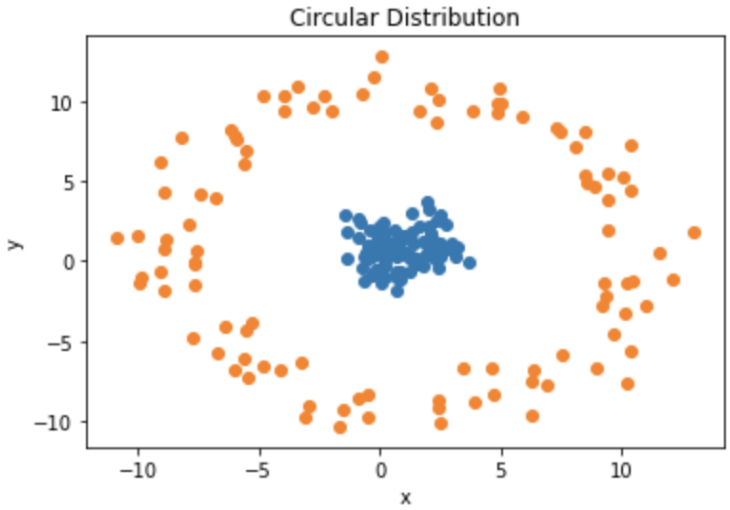


Figure 3.11. A dataset that cannot be classified by a 2-wide network

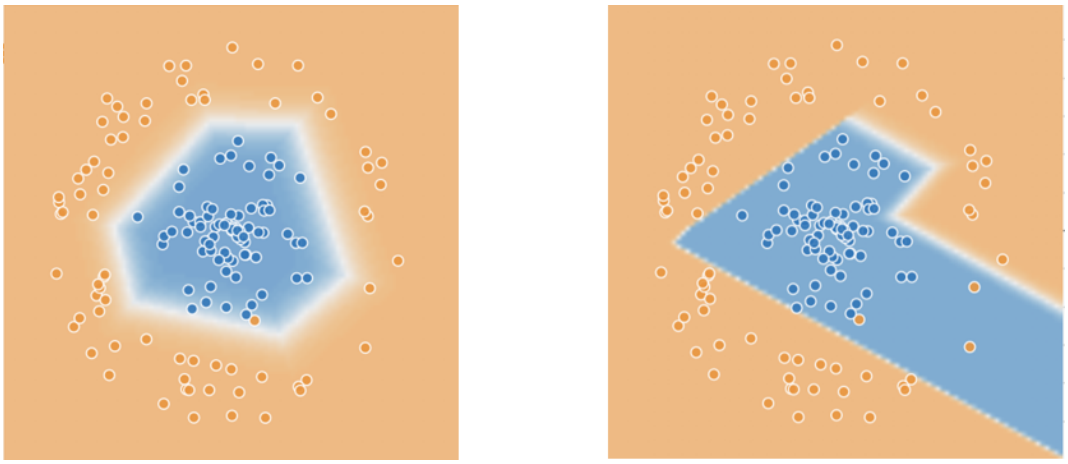


Figure 3.12. Classification boundary comparison: on the left is the boundary produced by a 1-deep, 3-wide ReLU network, on the right is the boundary produced by a 6-deep, 2-wide ReLU network. Image produced using Tensorflow Playground.

Chapter 4

Reinforcement Learning

Reinforcement Learning (RL) is a subfield of machine learning which tackles the learning of tasks by an agent in an environment where there is no direct supervision but only a scalar reward function. The reward may be sparse and delayed. Further the agent's actions affect the environment sometimes even irreversibly and the environment may change over time. The goal of the agent is to maximise the expected total reward.

4.1 The RL framework

One of the key hypotheses that RL is built on is the *Reward hypothesis*. It states that any goal can be described as the maximisation of a scalar reward. While this may be true for any computable goal, one of the challenging problems we face in RL, is how to define the goals that we want the agent to achieve. In many cases it turns out that either the goal we defined is too difficult to achieve or the optimum the agent finds is not one that we intended.

Figure 4.1 depicts the standard RL setup. We have an agent which receives an observation O_t from the environment at time t and then takes an action A_t to get a reward R_t and a new observation O_{t+1}. This is termed as one *interaction or transition or step*. The process repeats until the *episode* ends. An episode is usually terminated by the environment when the agent reaches a terminal goal state (Fulfills the task, dies or exceeds an interaction limit). We will assume environments never terminate for simplicity. Terminating or episodic environments are then merely special cases where $R_t = 0$ after termination.

Assuming the environment is computable, we have a *Markov state*: a state which describes everything we need to know about the environment, to simulate ahead. In other words, conditioned on the Markov state, the future is independent of the past. However the agent may not have access to this state and only has access to the *history* of observations, actions and rewards.

$$O_1, A_1, R_2, O_2, A_2, \ldots$$

It may be possible to infer the state of the environment using only this history, but not in all cases. An important point to note here is that partial observability is the sole source of randomness, assuming a deterministic environment.

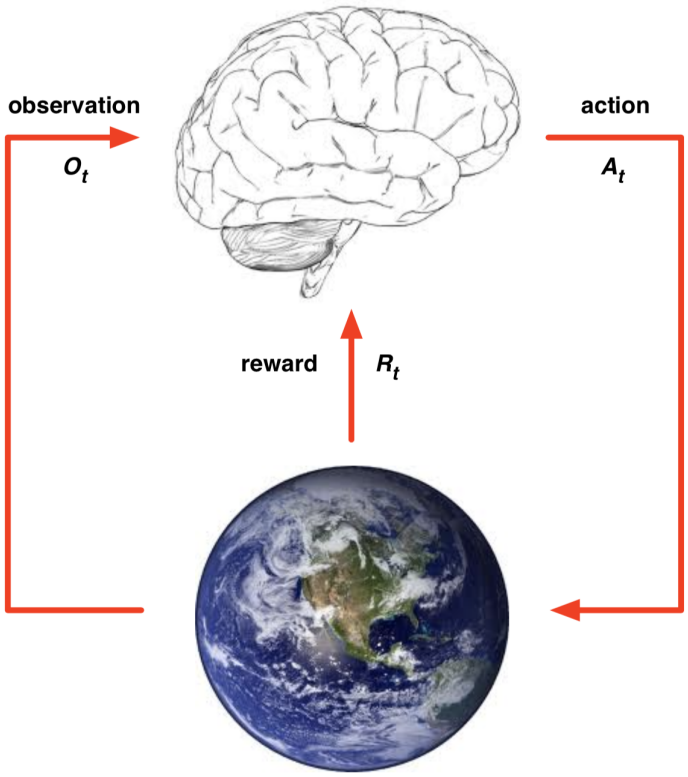


Figure 4.1. Standard Reinforcement Learning Setup [Silver, 2015]

Hence, the standard problem in RL is framed as a Partially Observable Markov Decision Process or POMDP. A POMDP is formally represented by a seven tuple:

$$(\mathscr{S}, \mathscr{A}, \mathscr{O}, \mathscr{P}, \mathscr{R}, \mathscr{Z}, \gamma)$$

1. $\mathscr{S}$ is the set of states that the environment can take
2. $\mathscr{A}$ is the set of actions available to the agent
3. $\mathscr{O}$ is the set of possible observations the agent can make
4. $\mathscr{P}$ is the state transition function for the environment
5. $\mathscr{R}$ is the reward function
6. $\mathscr{Z}$ is the observation function that determines what the agent observes
7. γ is the discount factor to deal with unbounded sums of rewards.

Note that this is a very general representation and supports even stochastic environments. For simplicity we will assume full observability and attribute the randomness to the environment itself. Hence our observation space collapses into the state space and the observation function is subsumed by a probabilistic state transition function.

4.2 Policies, Value Functions and Models

Now, An RL agent can have three major components: A *policy*, A *value function* and a *model*.

The policy is a function that maps each state to an action. This is how the agent decides what to do at any point in time. It may be either deterministic or stochastic. When it is deterministic, it outputs a single action for each state. In the stochastic case, it outputs a probability distribution over all possible actions, and an action is sampled accordingly. Represented as follows:

$$\pi(a \mid s) = \mathbb{P}[A_t = a \mid S_t = s]$$

The Value function determines the value of a certain state. It has two common implementations. One, as a function of only the state, called the state-value function (V). Two, as a function of both the state and the action, called the action-value function (Q). The value function estimates the expected total reward computed as follows:

$$G_t = R_{t+1} + \gamma R_{t+2} + \cdots = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1}$$

The discount factor $\gamma \in [0, 1]$ is needed as otherwise, the sum may not converge. Further we prefer future rewards to be discounted as the future may be uncertain. There are alternatives where the environment may be reward summable. In that case we may avoid the discount factor.

The state and action value functions are then written as follows:

$$V_\pi(s) = \mathbb{E}_\pi[G_t \mid S_t = s]$$

$$Q_\pi(s,a) = \mathbb{E}_\pi\left[G_t \mid S_t = s, A_t = a\right]$$

The value function is a distillation of the agent's knowledge of the environment and its own behaviour to produce a measure of the value of a given state (and action). This helps the agent learn faster as the agent doesn't have to wait till the end of the episode to decide how well it is doing and further the Q function can help the agent decide which is the best action to take. In the game of chess for example, it's often difficult to predict how to checkmate the opponent, but it is far easier to achieve control of the centre which is known to make winning easier.

The difference between the Q function and the V function is termed as the action advantage

$$A_\pi(s,a) = Q_\pi(s,a) - V_\pi(s)$$

Finally, a third component that an agent may have is a model of the environment. The environment is usually modelled by the state transition probabilities for every state and action.

$$P^a_{ss'} = P\left(S_{t+1} = s' \mid S_t = s, A_t = a\right)$$

Again, the tuple (s,a,r,s') is one transition. Further the reward may also be modelled in a similar fashion.

While in theory it's possible to build agents using just one of these three components, an agent which can efficiently use all three has significant performance advantages. In this work, we consider only *Actor-Critic* algorithms, which use only the first two: a policy (actor) and a value function (critic).

4.3 The Bellman Equations

At the heart of RL are the Bellman Equations. The *Bellman Expectation* equations define a recursive relationship between the state and action-value functions as follows:

$$\begin{aligned}
V_\pi(s) &= \sum_{a \in \mathscr{A}} \pi(a \mid s) Q_\pi(s,a) \\
Q_\pi(s,a) &= R(s,a) + \gamma \sum_{s' \in S} P^a_{ss'} V_\pi\left(s'\right) \\
V_\pi(s) &= \sum_{a \in \mathscr{A}} \pi(a \mid s) \left(R(s,a) + \gamma \sum_{s' \in S} P^a_{ss'} V_\pi\left(s'\right) \right) \\
Q_\pi(s,a) &= R(s,a) + \gamma \sum_{s' \in S} P^a_{ss'} \sum_{a' \in \mathscr{A}} \pi\left(a' \mid s'\right) Q_\pi\left(s', a'\right)
\end{aligned} \tag{4.1}$$

The *Bellman Optimality* equations which define the relationships for an optimal agent that achieves the maximum possible return, are similar to the ones above but with all the expectations over actions replaced by a maximum over the actions.

4.4 Temporal Difference Learning

While there are many approaches to solving the RL problem, in this work we primarily consider Temporal Difference (TD) Learning. TD learning starts with estimates of the value function and then updates the value function towards better estimates after each interaction with the environment. The update is done using the temporal difference between the current estimate of the value function and the estimate from the next step, the TD target, as follows:

$$V(S_t) \leftarrow V(S_t) + \alpha(R_{t+1} + \gamma V(S_{t+1}) - V(S_t)) \tag{4.2}$$

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha(R_{t+1} + \gamma Q(S_{t+1}, A_{t+1}) - Q(S_t, A_t)) \tag{4.3}$$

Here the term in the brackets is the temporal difference error. If the value function is already correct, we can see that on average the temporal difference would be zero, making no change. The parameter α is a learning rate parameter which governs how large the updates are. Hence TD learning *bootstraps* itself based on the fact that the value functions must be consistent with their one-step future predictions. Further it can learn online while the episode is still unfolding. Hence it is efficient in using information from the environment. The TD target is a biased but low variance estimate of the value as it only contains one time step of sampling.

One thing to note here is that the above equations make no assumptions about how the Q and V-functions are implemented. If the state and action spaces are finite and discrete, we can easily use a table to store all the values. For infinite spaces, we will need the universal approximation powers of a neural network as we will have to generalise to values that we will never have seen before.

4.5 Exploration and Exploitation

Any RL task can be subdivided into two tasks, Exploration and Exploitation. Exploration is important for the agent to gather information about the environment and the goals. Exploitation is then maximising the reward using the gathered information. Efficient RL requires a fine balance between the two. We don't want to have agents that keep exploring the usually infinite space of states and actions and barely maximise the reward and neither agents that try a few things and then decide the task is impossible or that they can't do any better. Since the action space is usually infinite, it is important to use certain priors for exploration, to wean down the search space. These priors can come in many forms. We will only consider the simplest exploration strategy here and leave the others for later.

4.6 Off-Policy Learning

The simplest way to implement exploration is to randomly sample an action with a probability of ϵ at every time step (ϵ-greedy). However doing this would clearly reduce the estimates of the return as some of the actions are not picked by the policy, but are random. Q-learning solves this problem by using the maximum of the Q-function while updating the estimates. This is commonly termed as Off-Policy learning as the Q-function learned is for the optimal policy, and not for the current policy which is free to explore.

$$Q(S_t,A_t) \leftarrow Q(S_t,A_t) + \alpha\Big(R_{t+1} + \gamma \max_{a \in \mathcal{A}} Q(S_{t+1},a) - Q(S_t,A_t)\Big) \tag{4.4}$$

DQN [Mnih et al., 2015] combines Q learning with function approximation via Neural Networks to scale up Q learning to high dimensional, continuous state and action spaces.

Although the generalisation advantages of neural networks make the algorithm very powerful, combining Off-policy learning with Bootstrapping and Non-linear function approximation is known to cause instabilities [Sutton and Barto, 2018]. DQN solves this problem using two simple ideas. They use a Replay Buffer to store past interactions so that the agent doesn't forget past experiences and repeat past mistakes. A method called Experience Replay [Lin, 1992]. In short this ensures that anything the agent does, counts either as different or as progress. Further they freeze the Q network and only update it periodically to avoid feedback oscillations. As the target Q network updates more slowly than the policy, short-term changes in the policy do not affect the target Q network cutting off the possibility of feedback oscillations.

DQN starts with an empty Replay Buffer of a fixed size N. Every epoch may consist of multiple episodes of interaction of an ϵ-greedy policy with the environment. With every interaction, the transition tuple (s,a,r,s') is stored in the replay buffer. And simultaneously a random mini-batch of samples is fetched from the buffer and for each sample, the target for the Q function is set as follows:

$$y = \begin{cases} r & \text{if episode terminates at the next step} \\ r + \gamma \max_{a'} Q_{target}\left(s', a'; \phi^T\right) & \text{otherwise} \end{cases}$$

Here Q_{target} represents the target Q function with parameters ϕ^T. Using this as the TD target, the squared TD error is used to update the Q function parameters via gradient descent. We have the following gradient estimator at step i:

$$\nabla_{\phi_i} L(\phi_i) \quad = \mathbb{E}_{s,a,r,s'}\left[\left(r + \gamma \max_{a'} Q_{target}\left(s', a'; \phi_i^T\right) - Q(s,a;\phi_i)\right)\nabla_{\phi_i} Q(s,a;\phi_i)\right] \tag{4.5}$$

The above algorithm takes advantage of the fact that there are a limited number of discrete actions and it is easy to maximise the Q-function by simply having it output the Q-values for all the actions and picking the maximum. However, when the action space is high dimensional or continuous, we require a way to estimate the actions that maximise the Q-function. For this, again we use a Neural Network to approximate a policy function that picks the best action given a state as input. We parametrise the policy with parameters θ as:

$$\pi_\theta(a \mid s) = \mathbb{P}[a \mid s, \theta]$$

This also provides us with the advantage that the action probabilities are differentiable with respect to the policy parameters allowing us to use gradient descent! This leads us to Policy Gradient Methods.

4.7 Policy Gradient Methods

We can define the objective function as follows[1]:

$$\mathscr{J}(\theta) = V_{\pi_\theta}(s_0) \tag{4.6}$$

Here s_0 is the initial state, for episodic environments. For continuing environments which don't have a fixed start state, we can just use the average value, using the law of the unconscious statistician:

$$\mathscr{J}(\theta) = \sum_{s \in S} d_{\pi_\theta}(s) V_{\pi_\theta}(s) = \sum_{s \in S} \left(d_{\pi_\theta}(s) \sum_{a \in \mathscr{A}} \pi_\theta(a \mid s; \theta) Q_{\pi_\theta}(s, a) \right) \tag{4.7}$$

Here $d_{\pi\theta}$ is the stationary distribution induced by the policy on the environment. Policy gradient algorithms simply try to optimize the policy to maximise this objective. Hence at each step they update the parameters θ by the standard gradient update:

$$\theta_t = \theta_{t-1} + \alpha \nabla_{\theta_{t-1}} J(\theta_{t-1}) \tag{4.8}$$

Now we have the Policy Gradient theorem which states that for any differentiable policy $\pi_\theta(a|s)$, for either of the objectives discussed above, the policy gradient is (equal for the continuing case, proportional for the episodic case):

$$\nabla_\theta \mathscr{J}(\theta) = \mathbb{E}_{\pi_\theta} \left[\nabla_\theta \log \pi_\theta(s, a) Q^{\pi_\theta}(s, a) \right] \tag{4.9}$$

We can easily see why this is in the episodic case:

$$\begin{aligned}
&\nabla_\theta V_{\pi_\theta}(s) \\
=&\nabla_\theta \left(\sum_{a \in \mathscr{A}} \pi_\theta(a \mid s) Q_{\pi_\theta}(s, a) \right) \\
=&\sum_{a \in \mathscr{A}} \left(\nabla_\theta \pi_\theta(a \mid s) Q_{\pi_\theta}(s, a) + \pi_\theta(a \mid s) \nabla_\theta Q_{\pi_\theta}(s, a) \right) \\
=&\sum_{a \in \mathscr{A}} \left(\nabla_\theta \pi_\theta(a \mid s) Q_{\pi_\theta}(s, a) + \pi_\theta(a \mid s) \nabla_\theta \sum_{s', r} P\left(s', r \mid s, a\right) \left(r + V_{\pi_\theta}\left(s'\right)\right) \right) \\
=&\sum_{a \in \mathscr{A}} \left(\nabla_\theta \pi_\theta(a \mid s) Q_{\pi_\theta}(s, a) + \pi_\theta(a \mid s) \sum_{s', r} P\left(s', r \mid s, a\right) \nabla_\theta V_{\pi_\theta}\left(s'\right) \right) \\
=&\sum_{a \in \mathscr{A}} \left(\nabla_\theta \pi_\theta(a \mid s) Q_{\pi_\theta}(s, a) + \pi_\theta(a \mid s) \sum_{s'} P\left(s' \mid s, a\right) \nabla_\theta V_{\pi_\theta}\left(s'\right) \right)
\end{aligned} \tag{4.10}$$

Thus we get a recursive equation. Substituting it within itself, we get:

$$= \sum_{x \in \mathscr{S}} \sum_{k=0}^{\infty} \Pr(s \to x, k, \pi_\theta) \sum_a \nabla \pi_\theta(a \mid x) Q_{\pi_\theta}(x, a)$$

Where $\Pr(s \to x, k, \pi_\theta)$ represents the probability of reaching state x in k steps starting from s. This allows us to brush the derivative of the Q function under the carpet of infinity.

[1] Note that this is the same objective of maximising the cumulative total reward, but using the value function

$$
\begin{aligned}
&= \sum_s \left(\sum_{k=0}^{\infty} \Pr(s_0 \to s, k, \pi_\theta) \right) \sum_a \nabla \pi_\theta(a \mid s) Q_{\pi_\theta}(s,a) \\
&= \sum_s \eta(s) \sum_a \nabla \pi_\theta(a \mid s) Q_{\pi_\theta}(s,a) \\
&= \sum_{s'} \eta(s') \sum_s \frac{\eta(s)}{\sum_{s'} \eta(s')} \sum_a \nabla \pi_\theta(a \mid s) Q_{\pi_\theta}(s,a) \\
&= \sum_{s'} \eta(s') \sum_s d_{\pi_\theta}(s) \sum_a \nabla \pi_\theta(a \mid s) Q_{\pi_\theta}(s,a) \\
&\propto \sum_s d_{\pi_\theta}(s) \sum_a \nabla \pi_\theta(a \mid s) Q_{\pi_\theta}(s,a)
\end{aligned}
\tag{4.11}
$$

The constant term behind the proportionality is just the average length of an episode as a sum of probabilities of reaching all states in k steps is just the probability of reaching the kth step and summing this over all possible k, just gives us the average length of the episode. For the continuing case this is simply 1 and thus we have equality.

Further we can use the *likelihood ratio trick* to get the final result:

$$
\begin{aligned}
\nabla_\theta J(\theta) &\propto \sum_{s \in \mathcal{S}} d_{\pi_\theta}(s) \sum_{a \in \mathcal{A}} Q_{\pi_\theta}(s,a) \nabla_\theta \pi_\theta(a \mid s) \\
&= \sum_{s \in S} d_{\pi_\theta}(s) \sum_{a \in \mathcal{A}} \pi_\theta(a \mid s) Q_{\pi_\theta}(s,a) \frac{\nabla_\theta \pi_\theta(a \mid s)}{\pi_\theta(a \mid s)} \\
&= \mathbb{E}_{\pi_\theta} \left[Q_{\pi_\theta}(s,a) \nabla_\theta \ln \pi_\theta(a \mid s) \right]
\end{aligned}
\tag{4.12}
$$

As mentioned earlier, Actor-Critic algorithms use both a policy as well as a value function. Specifically, both are approximated using neural networks. Thus the policy gradient is approximate, as the Q value is only an approximation with parameters ϕ. This approximation leads to a bias in the estimates, and we want to have the following instead:

$$
\begin{aligned}
\mathbb{E}_{\pi_\theta} \left[Q_{\pi_\theta}(s,a) \nabla_\theta \log \pi_\theta(s,a) \right] &= \mathbb{E}_{\pi_\theta} \left[Q_\phi(s,a) \nabla_\theta \log \pi_\theta(s,a) \right] \\
\mathbb{E}_{\pi_\theta} \left[\left(Q_{\pi_\theta}(s,a) - Q_\phi(s,a) \right) \nabla_\theta \log \pi_\theta(s,a) \right] &= 0 \\
\mathbb{E}_{\pi_\theta} \left[\left(Q_{\pi_\theta}(s,a) - Q_\phi(s,a) \right) \nabla_\phi Q_\phi(s,a) \right] &= 0 \\
\nabla_\phi \mathbb{E}_{\pi_\theta} \left[\left(Q_{\pi_\theta}(s,a) - Q_\phi(s,a) \right)^2 \right] &= 0
\end{aligned}
\tag{4.13}
$$

Thus by the *compatible function approximation theorem*, if the value function approximation minimises the mean squared error with respect to the true value function and has the same gradient as the log policy

$$
\nabla_\phi Q_\phi(s,a) = \nabla_\theta \log \pi_\theta(s,a) \tag{4.14}
$$

Then we have no bias. However we still have variance to tackle. The trick to dealing with the variance is to subtract a correlated variable that has zero mean! This reduces the variance without affecting the expectation. One easy to estimate correlated variable is the state value function. Thus subtracting we get the advantage function discussed earlier and the associated low variance gradient estimator:

$$\begin{aligned} A_{\pi_\theta}(s,a) &= Q_{\pi_\theta}(s,a) - V_{\pi_\theta}(s) \\ \nabla_\theta J(\theta) &= \mathbb{E}_{\pi_\theta}\left[A_{\pi_\theta}(s,a)\nabla_\theta \log \pi_\theta(s,a)\right] \end{aligned} \tag{4.15}$$

Now we are ready to understand Maximum Entropy RL methods

Chapter 5

Maximum Entropy Reinforcement Learning

Maximum Entropy Reinforcement Learning (MaxEnt RL) is an alternative formulation of the standard Reinforcement Learning problem. Instead of merely maximising the expected total reward, MaxEnt RL also maximises the entropy of the policy as well. Ignoring the discount factor, we can see that standard RL has the following objective:

$$J(\pi) = \sum_{t=0}^{T} \mathbb{E}_{(\mathbf{s}_t, \mathbf{a}_t) \sim \pi} [r(\mathbf{s}_t, \mathbf{a}_t)] \tag{5.1}$$

Max Entropy RL modifies the objective as follows:

$$J(\pi) = \sum_{t=0}^{T} \mathbb{E}_{(\mathbf{s}_t, \mathbf{a}_t) \sim \pi} [r(\mathbf{s}_t, \mathbf{a}_t) + \alpha \mathscr{H}(\pi(\cdot \mid \mathbf{s}_t))] \tag{5.2}$$

The coefficient α of the entropy term controls how important the entropy is with respect to the reward. Where standard RL finds a single best deterministic policy, MaxEntRL tends to find a stochastic policy that achieves the best reward. Further it finds a stochastic policy that maximises the number of different actions the agent can take at each step. It does so by matching the trajectories with the distribution of the exponentiated reward per trajectory. This is in fact a very common strategy in nature called Probability Matching.

5.1 Probability Matching

It has been found that many animals, from ants to humans, engage in this behaviour. This is what explains why humans don't always choose a single best option all the time. Studies have found that when presented with repeated trials of choices, where option A produces a reward with 0.8 probability and option B with 0.2 probability, humans tend to pick option A roughly 80% of the time and option B 20% of the time [Vulkan, 1998]. Clearly the maximising strategy is to pick option A 100% of the time but most humans match probabilities until they become aware of the fact that the maximising strategy is superior. As to why intuitively we adopt the

probability matching strategy is easy to understand. If the sequence being predicted was a predictable sequence, the best solution would be one which was matched in probability! Hence trying to match probabilities may help us build mental predictive models that accurately model the phenomenon. Experiments have even shown that probability matchers are more likely to identify patterns in non-random sequences. This could be explained by the fact that trying to predict a random sequence helps you compute an error from which you can update your model and learn. Whereas using the maximisation strategy isn't exactly a prediction as you're only trying to maximise your returns and hence there is no learning[1]. Further the strategy seems to be optimal in two particular situations[Eysenbach and Levine, 2019]. The first situation is one where the reward may be varying over time. For example, if an agent learned a deterministic policy of picking only one option it would never discover that the other option is sometimes much better. The second situation is one where the reward is picked adversarially. For example if there were two lakes, one with 70% probability of catching a fish and one with 30% probability, a group of agents who all go to the same lake will likely catch less fish than if they were to split themselves between the two.

Other explanations have also been provided which show that this is merely an irrational heuristic. For example by Kahneman and Frederick in 2002, who note that we may be internally substituting the difficult question "which pattern of options to pick?" with the easier "which pattern of rewarding options do we expect to see" [Kahneman and Frederick, 2002].

The main reason why the reward matching strategy is useful is because of its optimality under a changing reward. For most complex real world tasks it is a common observation that fixed reward functions are easily gamed by agents. For example, mortgage brokers who were incentivised to originate loans did so by extending loans to people with no means to pay back, leading to the 2008 financial crisis or more commonly, lawyers paid by the hour, who will ensure that your case continues forever. Hence in the absence of knowledge about what the true reward function must be, we have no choice but to make successive approximations. Arriving at the problem of having constantly changing reward functions!

Another way to understand why MaxEntRL methods are successful in standard RL tasks is through the fact that the entropy term makes optimization of the standard RL task much easier. In most of the standard RL algorithms with value function approximators, the value function approximations can be considered as maintaining implicit prior beliefs on the reward function. Actor Critic algorithms in particular have a part of the agent's reward determined by the value function estimator. So in that sense the agent is truly faced with a changing reward function in the form of the intrinsic value function based beliefs about the reward changing over time. This would certainly be an advantage for Soft Actor-Critic [Haarnoja et al., 2018], a MaxEnt RL based Actor-Critic algorithm. This would be an alternative way to explain the benefits of SAC as opposed to standard literature claims that it is due to the exploration that results purely from a stochastic policy. Since there are other RL algorithms which have stochastic policies (like SVG [Heess et al., 2015] for example) but do not enjoy the same advantages of SAC.

[1] Of course if it were possible to predict internally but externally pick the optimal choice, that would be the best of both worlds

5.2 Soft Actor-Critic

SAC is formulated based on soft policy iteration. Soft policy iteration like policy iteration methods involves two steps: *Policy Evaluation*, were the value of the policy is estimated for each state and then followed by *Policy improvement*, where the value function is used to update the policy to maximize the value it attains.

In the Policy evaluation step, a modified bellman backup operator is used to perform soft policy evaluation:

$$\mathscr{T}^{\pi}Q(\mathbf{s}_t,\mathbf{a}_t) \triangleq r(\mathbf{s}_t,\mathbf{a}_t) + \gamma \mathbb{E}_{\mathbf{s}_{t+1}\sim p}[V(\mathbf{s}_{t+1})] \tag{5.3}$$

$$V(\mathbf{s}_t) = \mathbb{E}_{\mathbf{a}_t\sim\pi}[Q(\mathbf{s}_t,\mathbf{a}_t) - \log \pi(\mathbf{a}_t \mid \mathbf{s}_t)] \tag{5.4}$$

The difference being the incorporation of the entropy term into the value function.

In the Policy Improvement step, the policy is iteratively improved using the following objective:

$$\pi_{\text{new}} = \arg\min_{\pi'\in\Pi} \mathrm{D}_{\mathrm{KL}}\left(\pi'(\cdot \mid \mathbf{s}_t) \,\|\, \frac{\exp(Q^{\pi_{\text{old}}}(\mathbf{s}_t,\cdot))}{Z^{\pi_{\text{old}}}(\mathbf{s}_t)}\right) \tag{5.5}$$

Basically the new policy is chosen such that it matches the distribution (normalised by Z) of the exponentiated Q function. It has been shown that a repeated application of these two steps converges to an optimal policy whose Q value is never less than that of any other policy in the set of all policies [Haarnoja et al., 2018].

Soft Actor-Critic is an approximation of Soft Policy iteration using neural networks to parametrize the Policy (Actor) and the value functions (Critic) and since it's difficult to run each step until convergence, both the policy and value functions are jointly optimised as is done in Actor-Critic methods.

The soft value function is parametrised by ψ and uses a mean squared error objective based on the soft bellman update rule:

$$J_V(\psi) = \mathbb{E}_{\mathbf{s}_t\sim\mathscr{D}}\left[\frac{1}{2}\left(V_{\psi}(\mathbf{s}_t) - \mathbb{E}_{\mathbf{a}_t\sim\pi_{\phi}}\left[Q_{\theta}(\mathbf{s}_t,\mathbf{a}_t) - \log \pi_{\phi}(\mathbf{a}_t \mid \mathbf{s}_t)\right]\right)^2\right] \tag{5.6}$$

For each update, a mini-batch of interactions (s,a,r,s') can be sampled from the replay buffer D. The target value doesn't depend on ψ and we can backpropagate through the soft value function to get the gradient with respect to ψ. Further the target value can be directly estimated using the Q function and the current policy and hence we have an unbiased gradient estimator for ψ.

The soft Q function is parametrised by θ and again uses a mean squared bellman error objective:

$$J_Q(\theta) = \mathbb{E}_{(\mathbf{s}_t,\mathbf{a}_t)\sim\mathscr{D}}\left[\frac{1}{2}\left(Q_{\theta}(\mathbf{s}_t,\mathbf{a}_t) - \left(r(\mathbf{s}_t,\mathbf{a}_t) + \gamma \mathbb{E}_{\mathbf{s}_{t+1}\sim p}\left[V_{\bar{\psi}}(\mathbf{s}_{t+1})\right]\right)\right)^2\right] \tag{5.7}$$

Here $\bar{\psi}$ represents an exponential moving average of the target value function which is one of the stability improving tricks discussed earlier (section 4.6). This target function is updated as follows at every step:

$$\bar{\psi} = \bar{\psi} * (1-\tau) + \psi * \tau \tag{5.8}$$

Where τ is usually set to something small like 0.01.

The policy parametrised by ϕ is optimised by directly minimising the reverse KL divergence:

$$J_\pi(\phi) = \mathbb{E}_{\mathbf{s}_t \sim \mathscr{D}}\left[\mathrm{D}_{\mathrm{KL}}\left(\pi_\phi(\cdot \mid \mathbf{s}_t) \| \frac{\exp\left(Q_\theta(\mathbf{s}_t, \cdot)\right)}{Z_\theta(\mathbf{s}_t)}\right)\right] \tag{5.9}$$

Here we can use the *reparametrization trick* to get a gradient estimator for ϕ. We reparametrize the sampling done by the policy as a function of the state and a random sample from a normal distribution:

$$\mathbf{a}_t = f_\phi(\epsilon_t, \mathbf{s}_t)$$

With this, we can differentiate through the action samples from the policy to get the gradients with respect to ϕ. Thus we can expand the objective as follows:

$$\begin{aligned} J_\pi(\phi) &= \mathbb{E}_{\mathbf{s}_t \sim \mathscr{D}}\left[D_{\mathrm{KL}}\left(\pi_\phi(\cdot \mid \mathbf{s}_t) \| \exp\left(Q_\theta(s_t, .) - \log Z_\theta(s_t)\right)\right)\right] \\ &= \mathbb{E}_{\mathbf{s}_t \sim \mathscr{D}}\left[\mathbb{E}_{a_t \sim \pi}\left[-\log\left(\frac{\exp\left(Q_\theta(s_t, a_t) - \log Z_\theta(s_t)\right)}{\pi_\phi(a_t \mid s_t)}\right)\right]\right] \\ &= \mathbb{E}_{\mathbf{s}_t \sim \mathscr{D}}\left[\mathbb{E}_{a_t \sim \pi}\left[\log \pi_\phi(a_t \mid s_t) - Q_\theta(s_t, a_t) + \log Z_\theta(s_t)\right]\right] \end{aligned} \tag{5.10}$$

And finally, adding in the reparametrization and ignoring the Z term as it doesn't affect the gradient:

$$J_\pi(\phi) = \mathbb{E}_{\mathbf{s}_t \sim \mathscr{D}, \epsilon_t \sim \mathscr{N}}\left[\log \pi_\phi\left(f_\phi(\epsilon_t, \mathbf{s}_t) \mid \mathbf{s}_t\right) - Q_\theta\left(\mathbf{s}_t, f_\phi(\epsilon_t, \mathbf{s}_t)\right)\right] \tag{5.11}$$

Giving us the following gradient estimator:

$$\hat{\nabla}_\phi J_\pi(\phi) = \nabla_\phi \log \pi_\phi(\mathbf{a}_t \mid \mathbf{s}_t) + \left(\nabla_{\mathbf{a}_t} \log \pi_\phi(\mathbf{a}_t \mid \mathbf{s}_t) - \nabla_{\mathbf{a}_t} Q(\mathbf{s}_t, \mathbf{a}_t)\right) \nabla_\phi f_\phi(\epsilon_t, \mathbf{s}_t) \tag{5.12}$$

Thus we have the following algorithm for SAC:

Algorithm 1 Soft Actor-Critic

Initialize parameter vectors ψ, $\bar{\psi}$, θ, ϕ.
for each iteration **do**
  for each environment step **do**
    $\mathbf{a}_t \sim \pi_\phi(\mathbf{a}_t|\mathbf{s}_t)$
    $\mathbf{s}_{t+1} \sim p(\mathbf{s}_{t+1}|\mathbf{s}_t, \mathbf{a}_t)$
    $\mathcal{D} \leftarrow \mathcal{D} \cup \{(\mathbf{s}_t, \mathbf{a}_t, r(\mathbf{s}_t, \mathbf{a}_t), \mathbf{s}_{t+1})\}$
  end for
  for each gradient step **do**
    $\psi \leftarrow \psi - \lambda_V \hat{\nabla}_\psi J_V(\psi)$
    $\theta_i \leftarrow \theta_i - \lambda_Q \hat{\nabla}_{\theta_i} J_Q(\theta_i)$ for $i \in \{1, 2\}$
    $\phi \leftarrow \phi - \lambda_\pi \hat{\nabla}_\phi J_\pi(\phi)$
    $\bar{\psi} \leftarrow \tau\psi + (1 - \tau)\bar{\psi}$
  end for
end for

Figure 5.1. SAC algorithm. Image taken from [Haarnoja et al., 2018]

Chapter 6

Automated Curriculum Learning

Automated Curriculum Learning (ACL) deals with algorithms that can automatically build curricula to train agents to achieve any goal with as few trials as possible. It can be seen as a special case of open-ended learning where we enforce the use of a curriculum. Many works have already addressed the problem of open-ended learning. For example, through Artificial Curiosity [Schmidhuber, 1991], where an agent is intrinsically motivated to learn more about the environment. And learning about the environment is useful in solving any task. Or Adversarial Artificial Curiosity[Schmidhuber, 2020a] , where a "controller" network is rewarded for finding novel phenomena that the "world model" network cannot model, while the world model is trained to model everything it encounters. Optimal Ordered Problem Solver (OOPS) [Schmidhuber, 2002], tries to solve a sequence of problems optimally by solving the easiest ones first and reusing parts of the solution to solve others. POWERPLAY [Schmidhuber, 2012][Srivastava et al., 2012] doesn't need a sequence of problems and can invent its own problems, specifically the simplest yet unsolved problem at each step. Thus it builds its own curriculum and learns.

There is one main reason motivating ACL. We need to be able to take in vast amounts of information from the environment to retrace the planetary-scale search[1] that has resulted in the development of human-level intelligence. Millions of lifelong interactions by trillions of lifeforms over billions of years [Hud and Fialho, 2019] have been distilled into the human genome. Further, a few million years of learned behaviours have been transferred from parents to children as a part of a cultural evolution [Elman, 1993]. This information or *priors* form the basis for human intelligence [Wolpert and Macready, 1997]. These priors include the spectral ranges of our senses, the higher-level structure of our brains, the shared visual illusions, psychological biases, culture, language etc. An indispensable aspect of our cultural evolution happens to be an adapting curriculum that helps train every new generation. If Albert Einstein was born on an otherwise uninhabited island, he would be no more intelligent than a mere ape. The curriculum is what makes us as intelligent as we are. The adapting curriculum is what accounts for the continued rise of our capabilities over time. Curriculums have been our ultimate tools in transferring the large quantities of knowledge necessary to reach adult human intelligence. Thus we can expect that ACL will be useful in reproducing the planetary-scale information transfer needed to build human-level AI.

Further, it is also becoming increasingly clear, that human-engineered reward functions (HERFs) are easily gamed, as they tend to have spurious optima. And they're also not easy

[1] If not galactic-scale, considering the Anthropic principle and Fermi's paradox

to learn from, requiring far too many samples. So it seems natural to let the agents design their own tasks or reward functions and consequently their own curricula.

The main objective of ACL has been formulated as follows[Portelas et al., 2020a]:

$$J_D = \max_{\mathscr{D}} \int_{T \sim \mathscr{T}_{\text{target}}} P_T^N \, \mathrm{d}T \tag{6.1}$$

The objective is to maximize the performance of an agent after N training steps P^N over a set of tasks from a target task distribution. This must be done by finding the optimal task selection function D, that selects the N tasks based on the interaction history. Essentially, we are trying to select N tasks such that we gain maximum information about the task distribution. This prior information should help us perform well on the target task distribution.

The first question to ask here: Is organizing the task sequence a way to improve this objective? It has been shown that using a good curriculum not only leads to faster convergence but also better generalization [Weinshall et al., 2018] and further using an anti-curriculum can lead to worse results as well!

[Portelas et al., 2020a] identify that ACL can be applied in two contexts. An agent needs to *learn* from the information it *collects* from the environment. In on-policy algorithms, the *learning* and *collection* are done by the same agent. In off-policy algorithms, the data may be collected by other mechanisms, separating the learning process from the collection process. Now ACL can benefit both of these processes simultaneously.

There are five major ways that traditional literature deals with collecting data from the environment in the context of ACL [Portelas et al., 2020a].

1. Modifying the agent start states to be closer to the goals [Florensa et al., 2018],
2. Modifying the reward function so that novel and diverse states are searched for. [Eysenbach et al., 2018][Shyam et al., 2019][Jabri et al., 2019]
3. Conditioning agents on goal representations, so that novel representations lead to novel goals via generalisation. [Reinke et al., 2020][Racaniere et al., 2020]
4. Generating environments based on Procedural Content Generation[Wang et al., 2019][Risi and Togelius, 2020]
5. Generating opponents who can lead the agent to novel states via adversarial or cooperative games [Sims, 1994][Silver et al., 2017a]

Hybrid approaches are also possible such as Paired Open-Ended Trailblazers (POET),[Wang et al., 2019] where the generated environments play a cooperative game with the agents! Basically all of these can be targeted by ACL methods to improve the usefulness of the data collected. However there is a key consideration to keep in mind, since most of the ACL literature deals with collecting data from simulation, it is important to be able to identify which information is useful in the real world. AlphaGo Zero[Silver et al., 2017b] for example has discovered novel Go strategies but sadly they do not generalise much to the real world outside of Go.

To efficiently learn from the collected data[2], we must maximize the information we get from each unit of compute spent. There are two general ways to do this, attention and modelling. Some environment transitions may be more informative than others. Thus, Prioritized

[2]Here we assume environment data is collected in the form of state transitions.

Experience Replay (PER) [Schaul et al., 2016], prioritizes more informative transitions during training. Thus it is a type of attention mechanism that helps us focus more on informative data. Modelling is another general way to maximize the information we gain, as we can update our model using the data. To a person who needs to build an AGI, a book on watering plants is not very informative, but if the person were to pretend that their goal was to water plants then the book can become very informative. This is precisely what Hindsight Experience Replay (HER)[Andrychowicz et al., 2018] does by pretending that the goal that was reached, was the intended goal. This way they can update their model of how goals map onto policies. This updated model should be better at finding the policy that achieves the required goal. Curriculum guided HER (CHER) [Fang et al., 2019] is a hybrid approach that combines both attention and modelling.

Many surrogate objectives have been identified for ACL such as reward, intermediate difficulty, learning progress, surprise, diversity etc[Portelas et al., 2020a]. We can optimise the selection process to obtain more rewarded interactions. We can optimise for always selecting tasks that are neither too hard nor too easy, but of intermediate difficulty so that we ensure we are making progress. Learning progress is a dual of the previous objective, where the progress of the agent is directly measured rather than measuring the difficulty of the task. The advantage that learning progress provides over intermediate difficulty is that there may be tasks that are of intermediate difficulty and yet the agent fails to learn anything: consider the task of tossing a coin to get heads! Maximising surprise is a more direct way, where surprise can either be measured via prediction error or via model disagreement[Schmidhuber, 1991][Shyam et al., 2019]. Model disagreement is better as it avoids the "noisy TV" problem (Noisy TVs are hard to predict making them always surprising) since all models can predict the mean output. In other words, the models act as a baseline for what the agent knows and a lack of disagreement suggests that although the outcome is surprising, there is nothing new to learn. Hence model disagreement is a measure of learning progress. In essence, all of these are merely ways to maximize information gain [Schmidhuber, 1991].

Thus curriculum learning is one of the priors that would be extremely important in designing agents that can learn in an open-ended way.

Chapter 7

Distributions Shifts and Warm Starting

The reason gradient descent works is that non-convex loss functions can be optimized effectively using local updates based on the gradient of the loss function. The gradient of the loss function only locally identifies the direction in which the network parameters must move to minimise the loss. And one often finds that performing a sufficient number of such local updates results in a good optimum. However computing the gradient over the full batch of training data is computationally expensive. A randomly sampled mini-batch is sufficient to obtain an unbiased estimate of the gradient. This makes stochastic gradient descent extremely efficient but with the added requirement that we must be able to uniformly sample the true data distribution. In online settings, this proves to be a problem as the data are no longer independent and identically distributed. The data distribution may change over time. And thus the gradient estimates are no longer unbiased making training difficult. As RL is also an online learning setting, this is a major problem.

[Ash and Adams, 2019] show that even when the data are drawn from the same distribution, updating models using new data is worse than retraining from scratch. More precisely, they show that models generalise better when trained on the full dataset for 350 epochs, rather than training first on 50% of the data for 350 epochs and continuing further for another 350 epochs on 100% of the data.

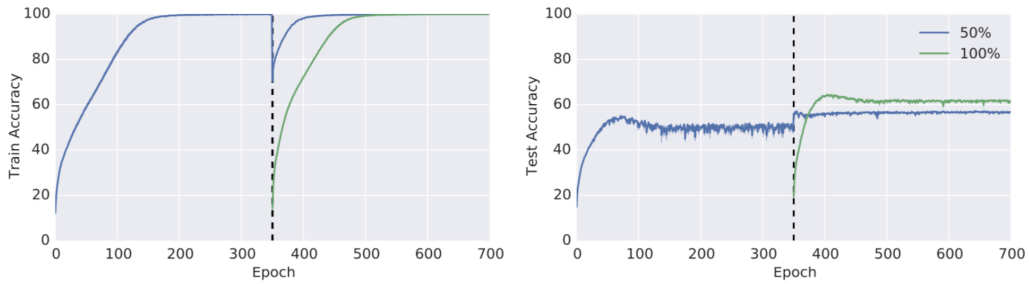


Figure 7.1. The left image depicts the training accuracy over time, while the right image depicts the test accuracy over time. Blue curve corresponds to a model trained first on 50% of the data for 350 epochs and then further trained on 100% of the data for another 350 epochs. The Green curve corresponds to a model trained from scratch on the full dataset for 350 epochs. Image taken from [Ash and Adams, 2019]

The latter method is commonly known as “Warm starting”(WS), where the network weights are pre-trained on data that might be relevant for the task rather than randomly initialising. This method is popular especially in few shot learning settings, where there is limited data for the actual task. Hence pre-training on any additional relevant data helps the networks make much better use of the limited task data. However as they show in their paper, this comes at the cost of a lower asymptotic performance. i.e. as more data is available for the task, the pre-training turns from an advantage into a liability.

This is in stark contrast to the results from curriculum learning [Weinshall et al., 2018] where we see that the performance on data drawn from a different distribution improves due to the WS. Both the facts can be reconciled as experiments using "anti-curricula" show worse results than randomly sorted data. So we may infer that the generalization gap arises since splitting the data in half can act as an anti-curriculum. Further, curriculum learning is being used successfully in RL as the sparse-reward environments may create scenarios where the agent is unable to receive any gradients until after it solves a sub-task successfully. But the above results suggest that an RL agent being trained via curriculum learning may need periodic distillation into randomly initialized networks to help improve generalization to downstream tasks.

[Nagarajan and Kolter, 2019] suggests that networks whose weights remain close to their original initializations, generalize better than ones whose weights have moved far away. In other words deviating from good initializations is bad for training on new tasks. However, [Ash and Adams, 2019] shows that deviating from a good initialization even in the direction of the task may be bad. They clearly show that any correlation of the final weights to the WS initialization indicates low performance. This problem provides an interesting perspective on Meta-learning approaches such as Model-Agnostic Meta-Learning (MAML) [Finn et al., 2017], suggesting that their approach to learning the initial weights may be flawed.

Although SGD is implicitly regularized in terms of nuclear norm [Li et al., 2019][Gunasekar et al., 2017], this implicit regularization hurts when used on smaller fractions of the data. This can be explained as an oversimplifying bias in the network that hinders it from finding good solutions while randomly initialized networks don’t suffer from this bias. However this doesn’t seem to be a problem when the curriculum is aptly designed! [Weinshall et al., 2018]

Unlike unsupervised pre-training, the goal of warm starting is not to address limited data issues in downstream tasks, but rather to speed up model updates without hurting generalization, as more data are collected. [Ash and Adams, 2019] venture that the warm start problem is one of the most significant problems in tackling resource consumption for training. They suggest that Polyak averaging, KFAC and AdaGrad may provide plausible solutions to this problem. However the proposed solutions are “fixes” without a deep theoretical motivation. As discussed previously, this is more easily solved via curriculum learning.

Chapter 8

Diversity Is All You Need?

Diversity Is All You Need [Eysenbach et al., 2018](DIAYN) is an unsupervised skill discovery method that uses a maximum entropy policy in conjunction with a set of learned reward functions to maximise an objective related to empowerment[Salge et al., 2013]. Empowerment can be seen as a measure of control over the environment. And skills here, refer to distinct predictable ways in which the state space can be manipulated. Thus empowerment is a measure defined on the number of such skills.

The objective in DIAYN is designed to meet three requirements. The first requirement is that the skills the agent uses and the states the agent visits have high Mutual Information. This ensures both, that the agent maximizes the entropy of the skills it uses, as well as minimizes the uncertainty about which skill was used given the state the agent is in. The second requirement is that the agent must try to take as diverse actions as possible at each state. This ensures that the policy is a maximum entropy policy thus enabling it to explore the environment and adapt quickly to changing reward functions. The third requirement is that the agent tries to use all possible actions to build each skill. This ensures that the skills are decoupled directly from the actions and specifically linked to a sequence of actions that lead the agent to a diverse region of the state space.

$$\begin{aligned} J(\theta) &\triangleq I(S;Z) + \mathcal{H}[A \mid S] - I(A;Z \mid S) \\ &= (\mathcal{H}[Z] - \mathcal{H}[Z \mid S]) + \mathcal{H}[A \mid S] - (\mathcal{H}[A \mid S] - \mathcal{H}[A \mid S,Z]) \\ &= \mathcal{H}[Z] - \mathcal{H}[Z \mid S] + \mathcal{H}[A \mid S,Z] \end{aligned} \tag{8.1}$$

Thus expanding and rearranging, we see that we end up with three terms. The first term corresponds to maximizing the entropy of the skills. The second term ensures that it's easy for a discriminator to identify the latent skill z given the state s. The third term ensures that the agent tries all possible actions for each skill to ensure that each skill uses a diverse sequence of actions. The third term is maximized by the entropy term of MaxEnt RL. The second term is intractable so they use a discriminator to approximate it, producing a variational lower bound to the objective. Hence the first two terms constitute the reward for the MaxEnt RL agent and it tries to maximize the discriminability of the states. Following is the derivation:

$$
\begin{aligned}
J(\theta) &= \mathcal{H}[A \mid S,Z] - \mathcal{H}[Z \mid S] + \mathcal{H}[Z] \\
&= \mathcal{H}[A \mid S,Z] + \mathbb{E}_{z\sim p(z),s\sim\pi(z)}[\log p(z \mid s)] - \mathbb{E}_{z\sim p(z)}[\log p(z)] \\
&= \mathcal{H}[A \mid S,Z] + \mathbb{E}_{z\sim p(z),s\sim\pi(z)}[\log q_\phi(z \mid s)\frac{p(z \mid s)}{q_\phi(z \mid s)}] - \mathbb{E}_{z\sim p(z)}[\log p(z)] \\
&= \mathcal{H}[A \mid S,Z] + \mathbb{E}_{z\sim p(z),s\sim\pi(z)}[\log q_\phi(z \mid s)] + \mathbb{E}_{z\sim p(z),s\sim\pi(z)}[\log \frac{p(z \mid s)}{q_\phi(z \mid s)}] - \mathbb{E}_{z\sim p(z)}[\log p(z)] \\
&= \mathcal{H}[A \mid S,Z] + \mathbb{E}_{z\sim p(z),s\sim\pi(z)}[\log q_\phi(z \mid s)] + KL(p(z \mid s)||q_\phi(z \mid s)) - \mathbb{E}_{z\sim p(z)}[\log p(z)] \\
&\geq \mathcal{H}[A \mid S,Z] + \mathbb{E}_{z\sim p(z),s\sim\pi(z)}\big[\log q_\phi(z \mid s) - \log p(z)\big] \triangleq \mathcal{G}(\theta,\phi)
\end{aligned}
\tag{8.2}
$$

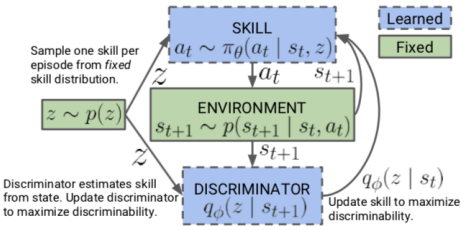
This gives us an objective $\mathcal{G}$ that can be implemented easily with the tools we have discussed earlier. The policy is parametrised as $\pi_\theta(a \mid s,z)$. Basically it's a neural network with parameters θ that outputs a distribution over actions given the state s and latent skill variable z as input. To have a parametrised distribution as an output, the network can output the k weights and the $k * actiondimension$ means and variances necessary to construct a mixture of k diagonal gaussians. The action can then be sampled from this mixture model.

The discriminator q_ϕ is modelled by a classifier that takes the state as input and predicts a softmax output over a finite set of discrete skills. Hence, the second term is merely the log probability of the target skill from the classifier. The distribution over the skills is fixed to be a uniform categorical distribution to avoid problems with the "Matthew effect" [Merton, 1968][Eysenbach et al., 2018] or "rich get richer, poor get poorer". Not fixing the prior distribution causes the algorithm to repeatedly pick only from a subset of the skills. Thus if we have 50 skills, the third term in the objective is merely a constant 3.91 that's added to the reward of the agent.

Hence we have the following intrinsic reward to maximize at each time step:

$$
r_z(s,a) \triangleq \log q_\phi(z \mid s) - \log p(z) \tag{8.3}
$$

The first term in the objective $\mathcal{G}$ is already incorporated in the MaxEnt formulation so it doesn't appear in the reward. Thus we get the algorithm in figure 8.1



Algorithm 1: DIAYN

while *not converged* **do**
  Sample skill $z \sim p(z)$ and initial state $s_0 \sim p_0(s)$
  for $t \leftarrow 1$ **to** *steps_per_episode* **do**
    Sample action $a_t \sim \pi_\theta(a_t \mid s_t, z)$ from skill.
    Step environment: $s_{t+1} \sim p(s_{t+1} \mid s_t, a_t)$.
    Compute $q_\phi(z \mid s_{t+1})$ with discriminator.
    Set skill reward $r_t = \log q_\phi(z \mid s_{t+1}) - \log p(z)$
    Update policy (θ) to maximize r_t with SAC.
    Update discriminator (ϕ) with SGD.

Figure 8.1. The DIAYN algorithm. Image taken from [Eysenbach et al., 2018].

This method is able to discover skills in an unsupervised manner and explore the state space in a structured way. The initial policy produces random trajectories and the discriminators learn

to identify some distinct features in those trajectories. As the discriminators learn, they guide the skill conditioned policy towards more distinct regions of the state space. The training of multiple skills simultaneously along with the joint training of the discriminators seems to automatically form a curriculum as the agent learns skills that it *doesn't otherwise learn with reward functions that correspond explicitly to those skills*. Figure 12.1 shows that the joint-training is particularly more useful than just multi-skill training. The benefits of training on multiple tasks simultaneously have already been shown [Finn et al., 2017][OpenAI et al., 2019]

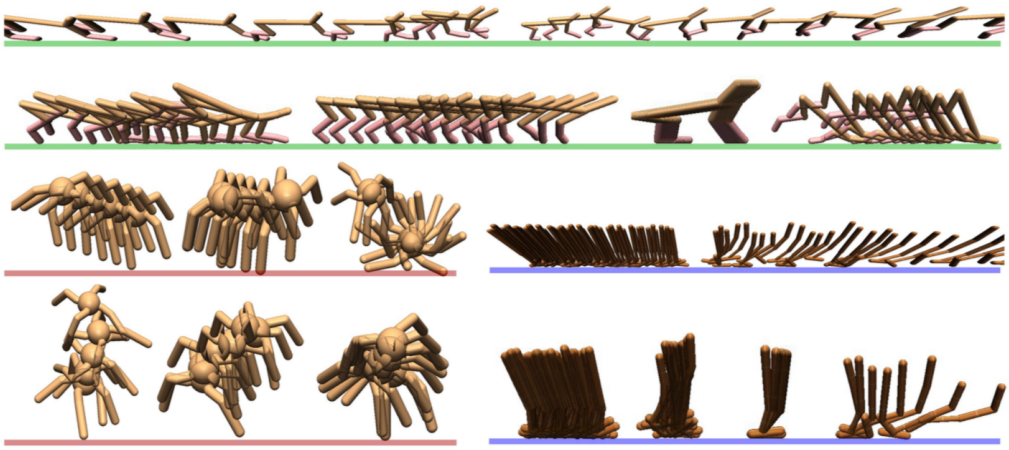


Figure 8.2. A visualisation of the skills learned by DIAYN, taken from [Eysenbach et al., 2018].

One of the key advantages of DIAYN is that it's formulated as a cooperative rather than an adversarial game. This helps avoid some instabilities. DIAYN not only discovers diverse skills that solve various tasks without ever having seen the task reward, but the DIAYN policy parameters turn out to be a good initialization when training on downstream tasks. Figure 8.3 shows the experimental results from the paper demonstrating this.

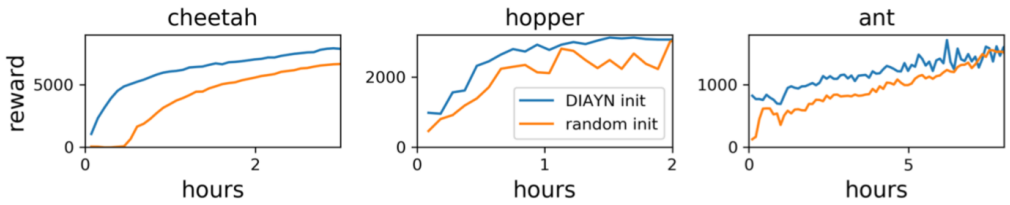


Figure 8.3. Generalization benefits of DIAYN: An RL agent initialized with the DIAYN parameters outperforms randomly initialized agents on multiple tasks. Image taken from [Eysenbach et al., 2018]

In essence the DIAYN policy produces two key benefits,

1. The discovered skills help the agent search the state space effectively to gather information about the task. This can be seen as a direct benefit of exploration by using empowerment as a prior.

2. The parametrization of multiple distinct action primitives in a single policy helps make the policy readily adaptable to any task. Experiments (Figure 11.1) show that the DIAYN policy generalises better to a running task even if it never learns a running skill.

Another advantage is that if the learned discriminator is modelled as an NN that takes s, z as inputs and predicts the probability that the skill z produced state s. This would prove to be very useful in defining tasks. The discriminator with a fixed z can behave as a reward function that guides an agent to learn the skill corresponding to z. [Gupta et al., 2019][Jabri et al., 2019] show that discriminators can be used to define a task distribution for an adaptive curriculum.

Unsupervised Curricula for visual meta-RL (CARML)[Jabri et al., 2019] uses an iterative approach where the state distribution of the agent is modelled by a generative-discriminative model (Deep-Cluster) [Caron et al., 2019]and this model is then used as a target for the agent to learn and adapt the distribution further. They expect this adapting distribution to generate a curriculum. They motivate this iterative process as a variational Expectation-Maximization algorithm. The expectation step corresponds to modelling the state distribution, while the maximisation step corresponds to maximizing an objective defined on the state distribution model. The objective encourages the agent to not only fit the task state distribution but also diversify it.

Apart from the main benefit of being able to formulate it as variational EM, the use of a learned reward function in the form of a discriminator network rather than a HERF has other benefits too. HERFs are more prone to local or spurious optima than these learned reward functions which are trained jointly with the agents on the generated data[1] and it also makes them feasible by default.

A major drawback of this approach is that the agent generates the curriculum for itself as it learns. And because of this the learning path may not be optimal, i.e. the agent may learn something that may not eventually prove to be useful and instead hinders the agent from learning more (succeeding at some downstream tasks). Further, a limiting factor of this iterative process, is the agent's learning ability[2]. The algorithm would stop improving at the point where the agent is barely able to achieve the task objective.

It may be better to use the final task state distribution of the agent to produce a curriculum for training the next agent from scratch, which should make it both easier to learn and produce generalization benefits [Weinshall et al., 2018][Ash and Adams, 2019][Graves et al., 2017][Schmidhuber, 2009][Schmidhuber, 2012] . This process if iterated, may lead to successively improved generalization, similar to the way generations of students becoming teachers, have been able to raise the threshold of human capability with every iteration. In the coming chapters we will see that indeed, a curriculum makes it easier to learn and produces generalization benefits in this context! However, wether such an iterative process can be sustained by incorporating this curriculum is left for future work.

[1] This is also why Variational Auto-Encoders with a reconstruction loss perform much worse than Generative Adversarial Networks, where the discriminator is trained jointly on the generated data.

[2] Despite the fact that CARML actually uses a meta-learning agent.

Now the key question is, how the curriculum may be generated given the final task distribution in the form of a discriminator. I propose a simple solution to generate this curriculum by distilling the trained discriminator into smaller networks. The fully-trained discriminator discriminates a small set of valid states from the rest and an agent training from scratch may fail to get any reward signal to learn from. However if the fully-trained discriminator is distilled into a smaller network; in the most extreme case, a single hidden neuron; then it would represent a hyperplane in the state space. The agent should find it much easier to cross this hyperplane than to reach the small set of states that correspond to the skill. As shown in the section on Width vs Depth (section 3.3) , the decision boundaries of less wide networks cover larger areas. By using a set of distilled discriminators that increase in size until the original discriminator, it should be possible to train an agent from scratch much faster than the original agent and get better generalization. I show that this process works for the simplest curriculum of training on one distilled discriminator, followed by training on the original discriminator. As a control, I show that this curriculum performs better than an agent trained directly on the trained DIAYN discriminator. This demonstrates the advantages of the curriculum in terms of both convergence and generalization.

Chapter 9

Mujoco Environments

MuJoCo [Todorov et al., 2012] or Multi-Joint dynamics with Contact is a physics simulator developed at the Movement Control Laboratory at the University of Washington. It comes with a set of benchmark continuous control tasks that are widely used by the RL community. It is designed specifically to be run faster than real time and optimiser proof, as optimisers are known to find and exploit bugs in physics engines. For this thesis, we will not delve into the implementation of MuJoCo but will only consider two environments which have been used to experimentally validate the results: HalfCheetah-v1 and Ant-v1.

9.1 HalfCheetah-v1

The HalfCheetah-v1 is a simulation of a 2-dimensional (half) cheetah with a metre long torso with two legs at either end. Each leg has 3 movable joints that the cheetah can actuate to run. The cheetah is free to move in the x and z directions while it's position along the y axis is fixed at 0. Gravity acts in the negative z direction. The state space of the HC env consists of 18 variables, shown in figure 9.2

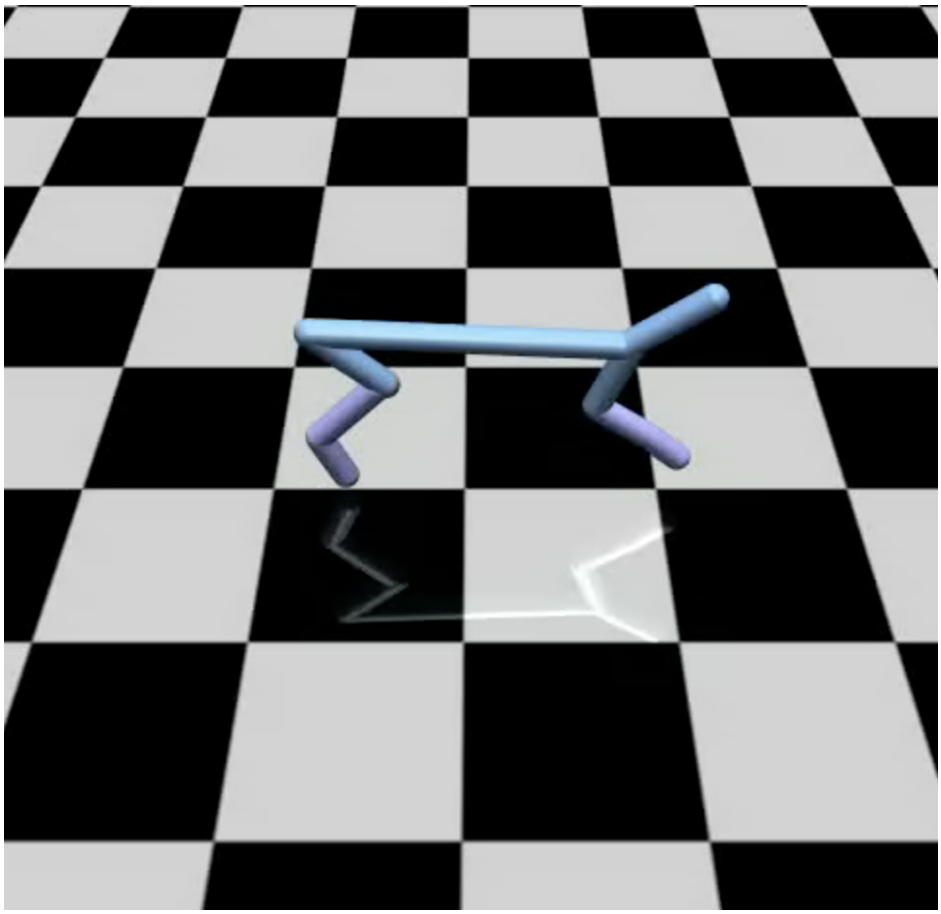


Figure 9.1. The MuJoCo Half Cheetah v1 Environment

State-Space name	joint	parameter:
- rootx	slider	position (m)
- rootz	slider	position (m)
- rooty	hinge	angle (rad)
- bthigh	hinge	angle (rad)
- bshin	hinge	angle (rad)
- bfoot	hinge	angle (rad)
- fthigh	hinge	angle (rad)
- fshin	hinge	angle (rad)
- ffoot	hinge	angle (rad)
- rootx	slider	velocity (m/s)
- rootz	slider	velocity (m/s)
- rooty	hinge	angular velocity (rad/s)
- bthigh	hinge	angular velocity (rad/s)
- bshin	hinge	angular velocity (rad/s)
- bfoot	hinge	angular velocity (rad/s)
- fthigh	hinge	angular velocity (rad/s)
- fshin	hinge	angular velocity (rad/s)
- ffoot	hinge	angular velocity (rad/s)

Figure 9.2. The Half-Cheetah State Space Description

Here sliders represent joints that can slide along an axis and hinges are joints that can rotate along an axis. The first two variables, $rootx$ and $rootz$ contain the absolute position of the cheetah along the x and z axis in metres. The cheetah is free to move along these two axes. $rooty$ contains the absolute orientation of the cheetah in radians and the cheetah is free to rotate along this axis.

The next six variables represent the orientation of each of the joints (Thigh, Shin and Foot) in the back and front legs in radians, measured along the y axis. The back thigh and shin can rotate $\sim$ 90 degrees while the foot can only rotate $\sim$ 66 degrees. The front thigh can rotate $\sim$ 100 degrees, shin $\sim$ 120 degrees and foot $\sim$ 60 degrees.

Finally the last 9 variables are the time derivatives of the first 9 variables. Using m/s for sliders and rad/s for hinges.

The action space of the HC env consists of a 6 dimensional vector, explained in figure 9.3

Actuator name	actuator	parameter
- bthigh	hinge	torque (N m)
- bshin	hinge	torque (N m)
- bfoot	hinge	torque (N m)
- fthigh	hinge	torque (N m)
- fshin	hinge	torque (N m)
- ffoot	hinge	torque (N m)

Figure 9.3. The Half Cheetah Action Space Description

The 6 values correspond to the control input on each of the six joints of the cheetah. The control input is limited between a range of [-1,1]. This is then converted into a torque.

The standard reward per time step is defined is defined as a function of the state and the action. The reward is defined as follows:

$$R_{\mathbf{s}}^{\mathbf{a}} = \mathbf{s}[9] - 0.1 * \|\mathbf{a}\|^2 \tag{9.1}$$

The first term is the velocity of the cheetah in the x direction and the second term is a control penalty. The second term ensures that actions which are strictly not necessary for running are minimized. It is common to ignore the x position, as it does not affect the policy and hence we have a 17 dimensional state space and a 6 dimensional action space.

9.2 The multi-task HERF benchmark

For evaluation on multiple tasks, I've designed a multi-task Human Engineered Reward Function (HERF) benchmark by adapting the reward formulation above. This benchmark uses a set of four basic tasks with a range of target goals to create a total of 25 tasks:

1. Running at speeds of -8, -6, -4, -2, 2, 4, 6 and 8 m/s
2. Flipping at speeds of -8, -6, -4, -2, 2, 4, 6 and 8 rad/s
3. Standing at angles of 90, 60, 120, -90, -60 degrees. (-120 has been removed as it is physically impossible to achieve)
4. Hopping at angles of 90, 60, -90, -60

For each of the running, flipping and standing tasks the reward function is defined as follows:

$$R_{\mathbf{s}}^{\mathbf{a}} = rewardscale * (-\text{abs}(x - target) + target - 0.1 * \|\mathbf{a}\|^2) \tag{9.2}$$

Here, x refers to the corresponding variable being measured, velocity or angle. Reward scale is set to 1 for all tasks except flipping. For flipping the reward scale is set to 5 as otherwise the agent fails to learn the flipping skill.

For the hopping tasks, the reward function is the same as the standing tasks but with an additional bonus term added for the absolute velocity in the z direction.

$$R_{\mathbf{s}}^{\mathbf{a}} = \text{abs}(s[1]) - \text{abs}(x - target) + target - 0.1 * \|\mathbf{a}\|^2 \tag{9.3}$$

9.3 Ant-v1

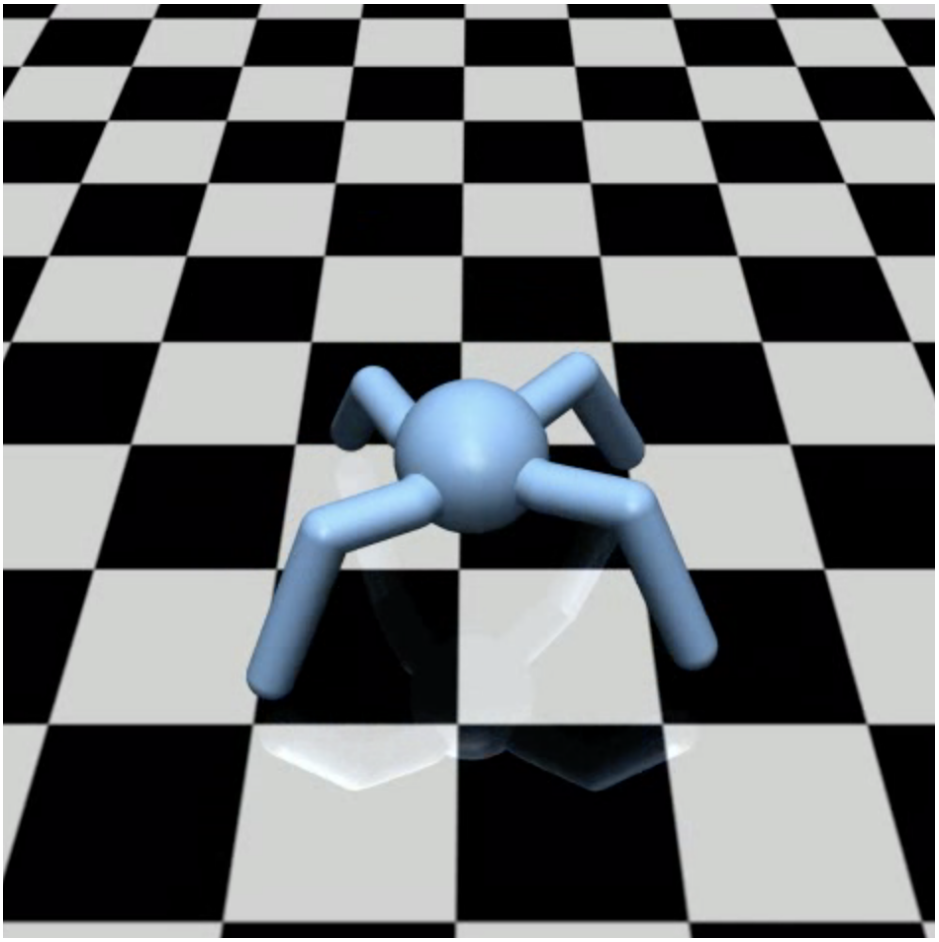


Figure 9.4. The MuJoCo Ant-v1 Environment

The ant-v1 is a simulation of a 3 dimensional ant-like robot with a spherical torso of 0.25m diameter with four legs. Each leg has a hip joint which is free to rotate about the z axis (ant reference frame) with a 60 degree range, and an ankle joint which is free to rotate with a 40 degree range. The ant is free to move in all three directions, with gravity in the negative z direction.

The ant state vector consists of 3 parts:

1. Position: The first 3 state parameters correspond to the x,y,z cooridinates of the torso. (The x and y coordinates are commonly ignored) The next 4 params correspond to the 4 components of the quaternion that defines the torso's absolute orientation. The next 8 params correspond to the joint angles in radians. Thus ignoring the x and y positions, we have 13 parameters for the position.

2. Velocity: The first 6 params correspond to the linear and angular velocities of the torso.The

next 8 params correspond to the angular velocities of the leg joints. So we have 14 params for the velocity.

3. External Contact Forces: The next 14 vectors correspond to the 3+3 force and torque components acting on the 14 bodies in the Ant Env. Thus we have 14x6 = 84 params for the external contact forces.

The action space corresponds to the control inputs applied on the two joints on each of the four legs, limited to a range of [-1,1]. This input is then translated into a torque.

The reward per time step is defined is defined as a function of the state and the action. The reward is defined as follows:

$$R_{\mathbf{s}}^{\mathbf{a}} = rewardscale * (\mathbf{s}[13] - 0.5 * \|\mathbf{a}\|^2 - 0.005 * \|\mathbf{s}[27:111]\|^2 + 1) \tag{9.4}$$

The first term is the velocity in the x direction, second term is the action penalty, the third term is an external contact force penalty and finally the last term is a survival bonus. In total we have a 111 dimensional state space and an 8 dimensional action space. The $rewardscale$ is set to 3, as done in DIAYN.

Chapter 10

Algorithms

10.1 DIAYN

Following is the DIAYN algorithm used. It is the same as the one from the official GitHub repository of DIAYN to ensure that all comparisons are fair. Further the same hyper-parameter settings are used as DIAYN in all cases. DIAYN trains for 10000 epochs on both HalfCheetah-v1 and Ant-v1 environments. However we train only for 1000 epochs as it takes around 5 days to run an experiment unto 10k epochs.

All networks, policy, value functions and discriminators are instantiated as a 2-deep, 300-wide ReLU network. The policy network outputs actions as a mixture of 4 diagonal gaussians in the action space. For this the network outputs the means and variances and the categorical weights of each mixture. The number of skills is fixed to 50 and the skills are represented by one-hot vectors that are appended to the state vector which is used as input to the policy and value functions.

Algorithm 1: DIAYN

Sample 1000 samples from the env using policy and add to replay buffer
while *epoch* $<$ *max epochs* **do**
  Sample skill $z \sim p(z)$ and initial state $s_0 \sim p_0(s)$
  for $t \leftarrow 1$ ***to*** *episode termination* **do**
    Sample action $a_t \sim \pi_\theta(a_t \mid s_t, z)$
    Step environment: $s_{t+1} = Env(s_t, a_t)$
    Add transition (s_t, z, a_t, s_{t+1}) to replay buffer
    Sample mini-batch d from replay buffer
    Compute $q_\phi(z' \mid s)$ on d with discriminator.
    Set skill reward $r = \log q_\phi(z' \mid s) - \log p(z')$ on d
    Update policy (θ) to maximize r with SAC.
    Update discriminator (ϕ) with SGD.
  end
end

10.2 Finetune

Following is the fine-tuning algorithm used in DIAYN. Before fine-tuning, the algorithm tries all 50 skills of the agent and finds the best skill that matches the task and fixes the agent to use that skill forever. Thus, if the DIAYN agent learns a running skill, it would get a high reward from the start. Again we use the same hyperparameter settings as DIAYN. This is used to test the generalization advantage for both HalfCheetah-v1 and Ant-v1 in all experiments.

Algorithm 2: Finetune

Initialize policy (θ) from pre-trained model
Get skill z that obtains the maximum return on the Env
Fix policy to use skill z
Sample 1000 samples from the Env using policy and add to replay buffer
while *epoch* $<$ *max epochs* **do**
  Sample initial state $s_0 \sim p_0(s)$
  for $t \leftarrow 1$ ***to*** *episode termination* **do**
    Sample action $a_t \sim \pi_\theta(a_t \mid s_t, z)$
    Step environment: $s_{t+1}, r_t = Env(s_t, a_t)$
    Add transition (s_t, a_t, r_t, s_{t+1}) to replay buffer
    Sample mini-batch d from replay buffer
    Update policy (θ) to maximize r with SAC on d.
  end
  validate policy on Env
end

10.3 Retrain

Following is the algorithm used to retrain the agents on the discriminator. The algorithm is the same as DIAYN but without training the discriminator. Only the policy is trained using the pre-trained discriminator.

Algorithm 3: Retrain

Sample 1000 samples from the env using policy and add to replay buffer
Load pre-trained DIAYN discriminator (ϕ)
while *epoch* $<$ *max epochs* **do**
  Sample skill $z \sim p(z)$ and initial state $s_0 \sim p_0(s)$
  for $t \leftarrow 1$ ***to*** *episode termination* **do**
    Sample action $a_t \sim \pi_\theta(a_t \mid s_t, z)$
    Step environment: $s_{t+1} = Env(s_t, a_t)$
    Add transition (s_t, z, a_t, s_{t+1}) to replay buffer
    Sample mini-batch d from replay buffer
    Compute $q_\phi(z' \mid s)$ on d with discriminator.
    Set skill reward $r = \log q_\phi(z' \mid s) - \log p(z')$ on d
    Update policy (θ) to maximize r with SAC.
  end
  validate policy for skill z using discriminator (ϕ)
end

10.4 Distill

Following is the distillation algorithm. We sample 1000 steps for each skill, totalling to 50k samples. Then we follow the same algorithm as DIAYN but only training the discriminator. The pre-trained DIAYN policy is kept fixed.

Algorithm 4: Distill

Initialize policy (θ) from pre-trained model
Sample 1000 samples from *each* skill using policy and add to dataset
while *epoch* < *max epochs* **do**
  Sample skill $z \sim p(z)$ and initial state $s_0 \sim p_0(s)$
  for $t \leftarrow 1$ ***to*** *episode termination* **do**
    Sample action $a_t \sim \pi_\theta(a_t \mid s_t, z)$
    Step environment: $s_{t+1} = Env(s_t, a_t)$
    Add transition (s_t, z) to dataset
    Sample mini-batch d from dataset
    Compute $q_\phi(z' \mid s)$ on d with discriminator.
    Update discriminator (ϕ) with SGD.
  end
  validate discriminator (ϕ) for skill z on policy
end

10.5 Curriculum training

Following is the curriculum training algorithm. We use a simple curriculum that consists of two stages. In the first stage, the agent is trained using the distilled discriminator for 500 epochs followed by the second stage with another 500 epochs of training on the original DIAYN discriminator. The switching point was chosen heuristically and further improvements may be obtained via line search.

Algorithm 5: Curriculum Training

Sample 1000 samples from the env using policy and add to replay buffer
Load distilled discriminator (ϕ_1)
Load pre-trained DIAYN discriminator (ϕ_2)
Set discriminator (ϕ) $\leftarrow$(ϕ_1)
while *epoch* $<$ *max epochs* **do**
  Sample skill $z \sim p(z)$ and initial state $s_0 \sim p_0(s)$
  if *epoch* $==$ $epoch_{switch}$ **then**
    Set discriminator (ϕ) $\leftarrow$(ϕ_2)
  end
  for $t \leftarrow 1$ ***to*** *episode termination* **do**
    Sample action $a_t \sim \pi_\theta(a_t \mid s_t, z)$
    Step environment: $s_{t+1} = Env(s_t, a_t)$
    Add transition (s_t, z, a_t, s_{t+1}) *to replay buffer*
    Sample mini-batch d from replay buffer
    Compute $q_\phi(z' \mid s)$ *on d with discriminator.*
    Set skill reward $r = \log q_\phi(z' \mid s) - \log p(z')$ *on d*
    Update policy (θ) *to maximize r with* SAC.
  end
  validate policy for skill z using discriminator (ϕ)
end

10.6 Multi-skill HERF

Following is the multi-task HERF benchmark evaluation algorithm. It is designed to be exactly the same as the fine-tuning algorithm but for training on 25 different tasks simultaneously. This was done to save time in benchmarking and to control for any task specific biases affecting the results by training on multiple different tasks. This was only used with the Half-Cheetah-v1 Environment. The validation was performed only once every 50 epochs as it was time consuming to validate on all 25 tasks every epoch.

One of the problems with multi-skill fine-tuning is that the DIAYN skills may not have a one to one match for all the tasks. It is possible that only one skill may produce the highest return on multiple tasks. Hence to reduce the impact of the skill matching on the algorithm results, we ensure that each HERF task gets matched with the highest scoring skill. The skills are cloned if necessary by swapping in the column of weights in the first layer that correspond to that skill's one-hot vector.

Algorithm 6: Multi-skill HERF Benchmark

Initialize policy (θ) from pre-trained model
Match best scoring policy skill with each HERF task
Sample 1000 samples from the env using policy and add to replay buffer
while *epoch* $<$ *max epochs* **do**
  Sample skill $z \sim p(z)$ and initial state $s_0 \sim p_0(s)$
  for $t \leftarrow 1$ ***to*** *episode termination* **do**
    Sample action $a_t \sim \pi_\theta(a_t \mid s_t, z)$
    Step environment: $s_{t+1}, r_t^z = Env(s_t, a_t)$
    Add transition $(s_t, z, a_t, r_t^z, s_{t+1})$ to replay buffer
    Sample mini-batch d from replay buffer
    Update policy (θ) to maximize r^z with SAC on d.
  end
  Validate policy on all HERF tasks every 50 epochs
end

Chapter 11

The Benefits of Retraining

In the context of unsupervised skill discovery, we can expect to see the same problems with distribution shifts and consequently the generalization gap. DIAYN has tasks defined by discriminators which are jointly trained along with the policy. Because of that, the task definition changes with every iteration leading to shifts in the optimal policy state distribution. Moreover, the policy already has to deal with a shifting data distribution as it learns the skills.

Figure 11.1 demonstrates the generalisation benefits of retraining in comparison with both the DIAYN trained model and the randomly initialized model on the HalfCheetah-v1 running task[1]. Figure 11.2 shows a comparison on the Ant-v1 environment[2].

From these figures, we can clearly see that the agent trained directly on the pre-trained DIAYN discriminator, generalises much better to the running tasks (i.e. produces much better returns upon fine-tuning on the running tasks) than the DIAYN agent. This is what we expected based on the distribution shifts argument. However, upon further inspection, there are a couple of other confounding factors that contribute to this result.

[1] Note that the plot shows 100k Steps. Here 1 Step = 10 Epochs.

[2] Some of the seeds produce bad results suggesting the DIAYN algorithm has some instabilities.

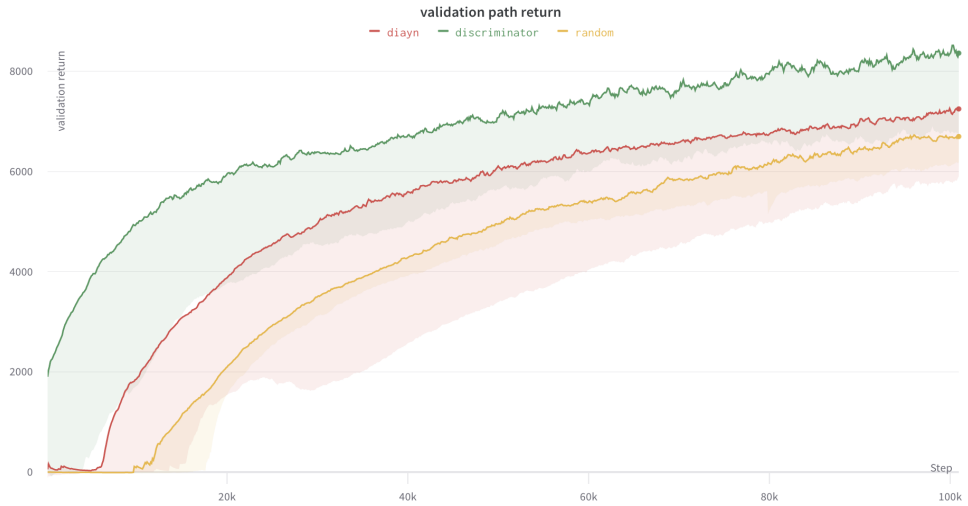


Figure 11.1. Here we can see a comparison of the validation path return over 1000 epochs on the HalfCheetah-v1 running task. The green curve corresponds to the agent trained directly on the DIAYN discriminators. The shaded regions represent the envelope (max, min) of the returns over a set of 5 seeds each. I compare the maximum as done in DIAYN. This is done since unsupervised pre-training is free and the best run can be selected at the start. [Personal communication from author]

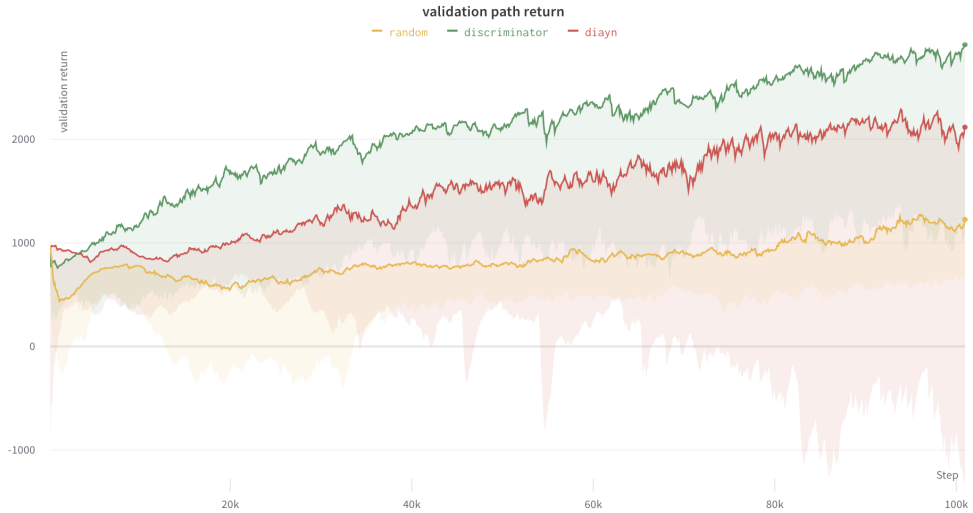


Figure 11.2. Here we can see a comparison of the validation path return over epochs on the Ant-V1 running task. In yellow is the return curve of the randomly initialized agent. In red is the DIAYN agent and in green is the agent trained directly on the discriminator obtained from DIAYN training. The shaded regions represent the envelope of the returns over a set of 5 seeds each. I compare the maximum as done in DIAYN

First, DIAYN, by default, uses an entropy scaling factor of 0.1 as the convergence is much slower for higher entropy factors. Figure 11.3 is a comparison with a higher entropy version of DIAYN with an entropy scaling of 1.0.

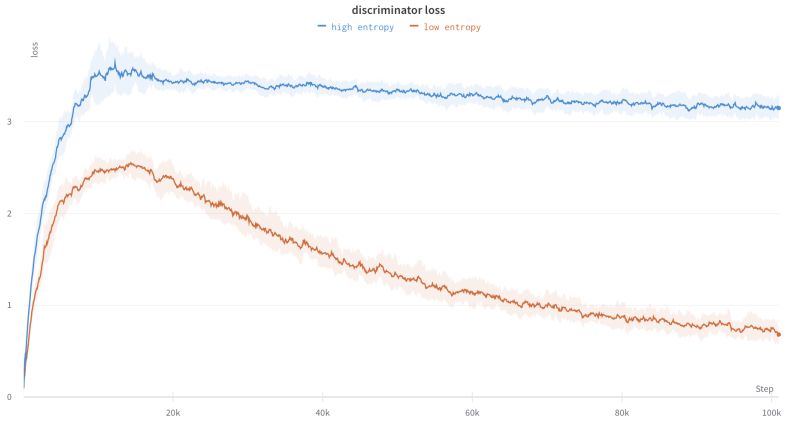


Figure 11.3. Comparison of discriminator losses when DIAYN is trained with an entropy scale of 1.0 (high entropy) vs 0.1 (low entropy). We see that the low entropy version learns far more discriminable skills based on the low discriminator loss. This is expected as higher entropy skills are likely to be less discriminable.

This is due to the fact that the discriminator reward, which is approximately within a range of [-4,4], is mostly subsumed by the entropy bonus. In the beginning, the agent learns to ignore the discriminability objective and simply maximises the entropy of its actions. This makes the skills very difficult to distinguish and thus the discriminator rewards become very low as seen in figure 11.4 and 11.5.

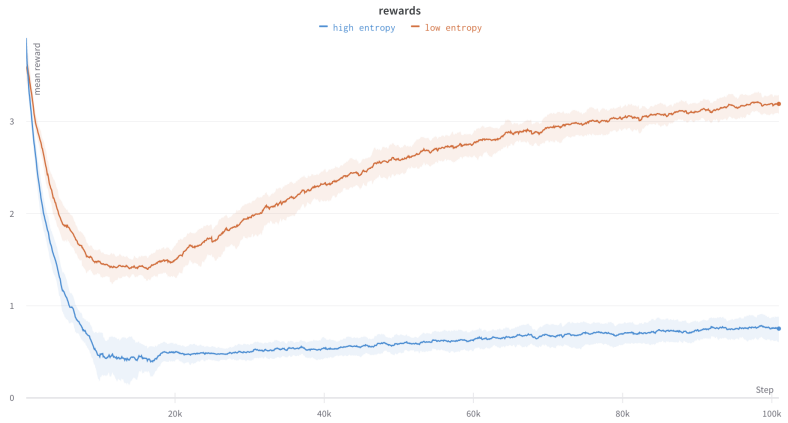


Figure 11.4. Comparison of the mean rewards of the mini-batches sampled from the Replay Buffer over time. We see that the high entropy version receives low rewards from the discriminator.

Once the entropy is maximised it starts slowly updating itself to become more discriminable. Hence 1.0 is a bad choice of the entropy scaling factor.

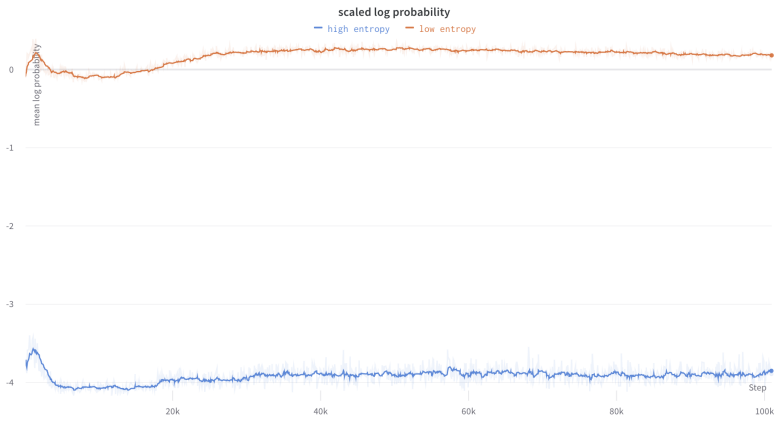


Figure 11.5. Blue curve represents DIAYN trained with a 1.0 entropy scale, and Orange represents 0.1. We can see that the higher entropy version relies heavily on the negative entropy penalty as a positive reward. While the low entropy version is maximising the discriminability at the cost of a positive entropy penalty.

When fine-tuning on downstream tasks such as the running task, the entropy factor of 0.1 is no longer good. This is because the running task reward is not restricted to a small range as before. Hence using a low entropy scaling factor hinders exploration drastically. This leads the agent to quickly maximize the task reward, only to get stuck in a local optimum. Further the training also becomes quite brittle due to a lack of data diversity in the Replay buffer. This makes the algorithm more prone to instabilities. [Haarnoja et al., 2018] This is not a problem with the DIAYN training, as the policy explicitly tries to learn a diverse set of skills. Thus, for fine-tuning, an entropy scaling factor of 1.0 is used in the DIAYN paper and the same choice is followed here as well.

Now, when retraining on the discriminator, I use an entropy factor of 1.0 as the discriminator is already trained and produces large negative rewards ~ -4; which easily competes with the entropy bonus. Further the trained policy would be a high entropy policy and hence would have a head start over the low entropy DIAYN policy.

We can see in figure 11.6 the exploration benefits of high entropy DIAYN over low entropy DIAYN. It provides a head start but as the low entropy DIAYN agent learns to increase its entropy, the benefit of having learned multiple discriminable skills seems to even out the results.

Figure 11.7 shows a comparison of the benefits of entropy and the retraining.

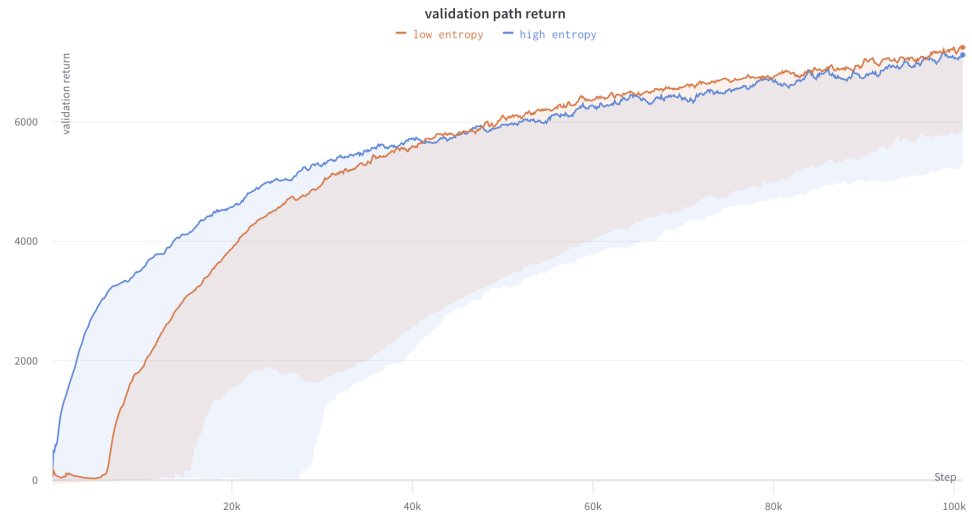


Figure 11.6. In blue is the high entropy DIAYN, in orange is the low entropy version. We can see that the high entropy version has a head start on the HalfCheetah-v1 fine-tuning task

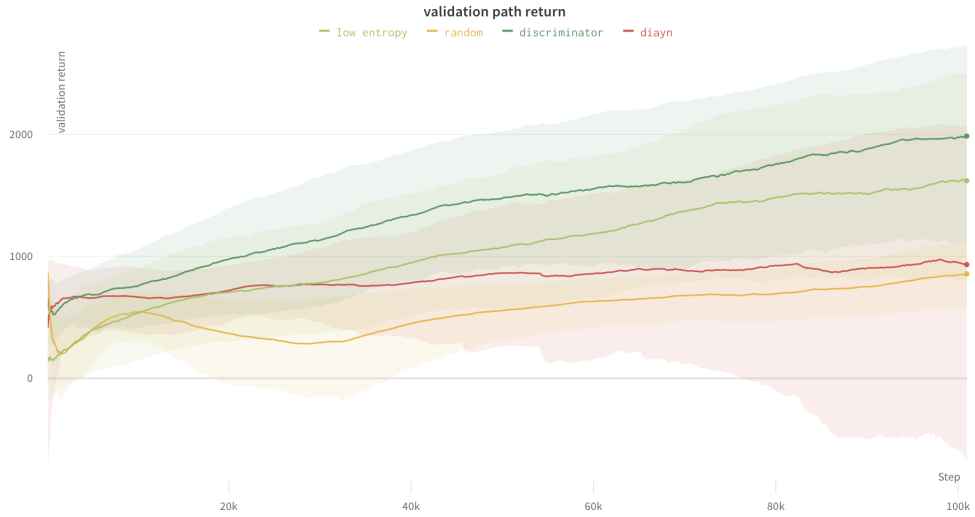


Figure 11.7. In dark green is the high entropy retrained agent, in light green is the low entropy retrained agent, in red is the DIAYN agent and yellow is the randomly initialized agent. The plot shows the fine-tuning results on the Ant-v1 running task. The low entropy retrained version still outperforms the DIAYN agent. This shows that the benefits are both due to retraining as well as the higher entropy coefficient.

Another factor that helps explain the good performance of the discriminator trained agent on the Half-Cheetah running task in particular, is a curious one. Upon observation of the learned skills, one finds that agents trained on the DIAYN discriminator learn running skills when even the original DIAYN agent doesn't learn them. Upon analysing the discriminator reward on this learned running skill we see the plot in figure 11.8

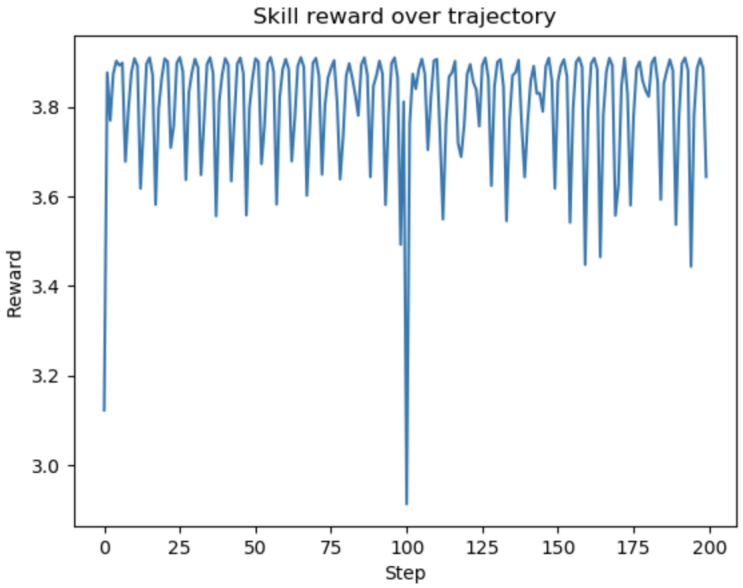


Figure 11.8. Reward-per-time-step from the DIAYN discriminator on the retrained agent over two episodes of 100 steps each. We see that the running skill learned by the agent shows oscillations suggesting it does not actually correspond to a maximum of the discriminator.

It seems the running skill learned by the agent doesn't correspond to a constant maximum of the discriminator which should be a reward-per-step of ~ 3.91. A naive way to understand what's happening is that the running skill can be achieved via oscillations around a fixed pose. Let's assume that one of the skills corresponds to such a pose. In the DIAYN setup, the discriminators are trained using the agent's state distribution so they would tend to reward any discriminable state achieved by the agent. In the retraining setup however, the agent needs to specifically achieve the state distribution, the discriminator corresponds to. Let's assume that the agent matches the pose perfectly apart for the hind leg being slightly off. It moves the hind leg a little to correct it and while doing so achieves the discriminator maximum, but consequently the front leg slips out of position. Now it tries to correct the front leg only to get the hind leg out of position again. This oscillating behaviour can produce a running skill. Thus we can see that in some of the runs, the retrained agent directly starts with a reward of ~ 2000 on the running task.

Thus to correct for such single-skill biases, a multi-skill benchmark has been designed to evaluate the benefits of retraining using the DIAYN discriminator. The multi-skill benchmark is explained in detail in the section on The multi-task HERF benchmark (section 9.2). Figure 11.9 shows the results on the multi-skill HERF benchmark.

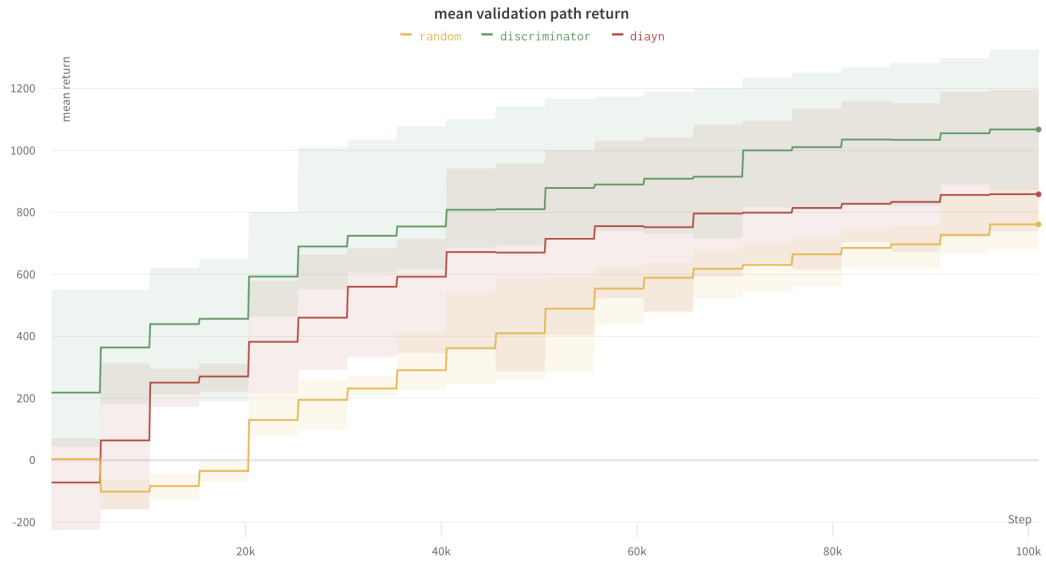


Figure 11.9. In green is the agent retrained on the discriminator, In red is the DIAYN agent and in yellow is the randomly initialized agent. The curves correspond to the average validation return on a set of 25 different tasks evaluated once every 50 epochs.

Thus we may conclude that there are now two major factors which help explain the performance of the retrained agent, the distribution shifts and the higher entropy training.

Chapter 12

The Benefits of Curricula

One of the benefits of DIAYN that we lose with retraining is that DIAYN learns some skills that the retrained version finds difficult to learn by itself. This is because although the multi-skill retraining provides the benefit of a curriculum, it is not however, as good as DIAYN. In DIAYN, not only does the multiple skill training provide a curriculum, but since the discriminators are jointly trained with the agent, the tasks are always of "intermediate difficulty". Thus making it much easier for the DIAYN agent to learn even very difficult skills like 360 degree back flips and front flips. Figure 12.1 shows a comparison of the average reward obtained by the agent for each of the 50 skills as it trains in the Half Cheetah environment.

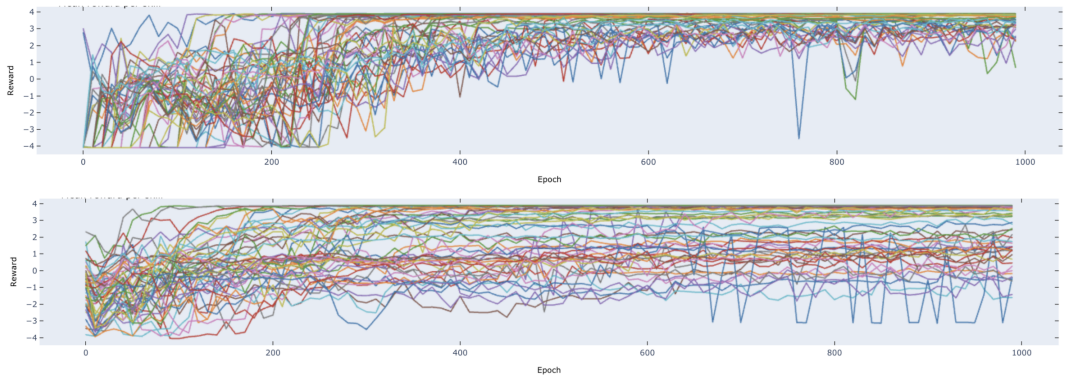


Figure 12.1. Above: Average validation reward-per-step on all 50 skills of DIAYN vs epochs. Different colours correspond to different skills. Below: Average validation reward-per-step of all 50 skills while retraining on the DIAYN discriminator. The DIAYN agent achieves much higher skill rewards and converges much faster due to the intermediate difficulty of the jointly trained discriminator.

This can also be considered as an issue of sparse reward. Since the discriminator is highly specific, the agent being retrained may never visit any rewarded state, or if it does, it may still get stuck in local optima. An important aspect to understand here is the meaning of reward sparsity. More precisely, a sparse reward scenario isn't necessarily one in which there is no reward. It can also be the case that there is positive reward but no matter which direction the agent explores, it finds the reward to be unchanging and low. Hence regions of sparse reward

may be considered to correspond to plateaus which lack significant reward gradient and hence providing little information about what the task is or how to improve on the task. Thus we can see that the discriminator trained agents receive a positive reward and yet, are unable to improve. Further this could also be a problem with local optima, where instead of the reward being unchanging, actually decreases in all directions that the agent explores.

Hence we need a way to regain this benefit of intermediate difficulty, while also having the benefits of lower distribution shifts and entropy. Thus we can consider designing a discriminator based curriculum. As we saw in the Neural Networks chapter, a lower width discriminator would have a larger area where it provides positive reward. Further, we would also like to ensure that there is a consistent reward gradient that leads the agent from a random initial state distribution to achieving the skill distribution. In general, since less wide networks also have less expressive power, we can expect that the reward function fit by the lesser width discriminator will also be less wiggly than the full width discriminator. Hence, I distill the 300-wide DIAYN discriminator for the HalfCheetah-v1 task into a 20[1]-wide discriminator. To verify whether the reward sparsity and consistency arguments hold, I sample a set of 1000 randomly initialised states, a set of 250 highly rewarded states for the DIAYN discriminator (5 states for each skill) and finally I sample 1000 pairs of points between the two sets. Then I plot the average of the reward curves along 100 points evenly spaced between each pair of points. Hence these correspond to the reward profiles along 1000 straight paths from random starting states to skill states. The outcome is depicted in figure 12.2

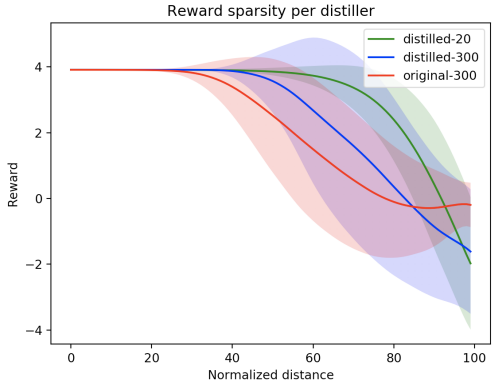


Figure 12.2. Curves showing average reward provided by each discriminator (original DIAYN, distilled 300-wide and distilled 20-wide) as one travels along a straight line joining a random start state (normalised distance = 100) to a random high scoring skill state (normalised distance = 0). The envelopes correspond to the standard deviation of the reward. We see that the original DIAYN discriminator has a region of sparse reward, a plateau near the beginning, making it difficult to use for training. The 20-wide discriminator has a very steep and consistent gradient from the very start, low standard deviation and the largest region of high reward. The 300-wide distilled discriminator is shown to compare the benefits of decreasing the width vs distilling. Distilling also has some benefit, but decreasing the width is clearly much better in terms of the variance, the gradient and the size of the region of high reward.

[1]Halving the width 4 times and rounding to nearest multiple of 10.

As we can see from figure 12.2, the distilled 20-wide discriminator has the largest highly rewarded region, steepest reward gradient and the least variance making it the easiest to maximise. To ensure that the benefit comes from the width and not just the distillation process, the DIAYN discriminator has also been distilled into a 300-wide discriminator and we see that although the distillation itself has some benefits, it is not good enough alone.

Thus training first using a less wide discriminator, we should also expect the agent to achieve a higher reward on the DIAYN discriminator and this is precisely what is observed! (12.3)

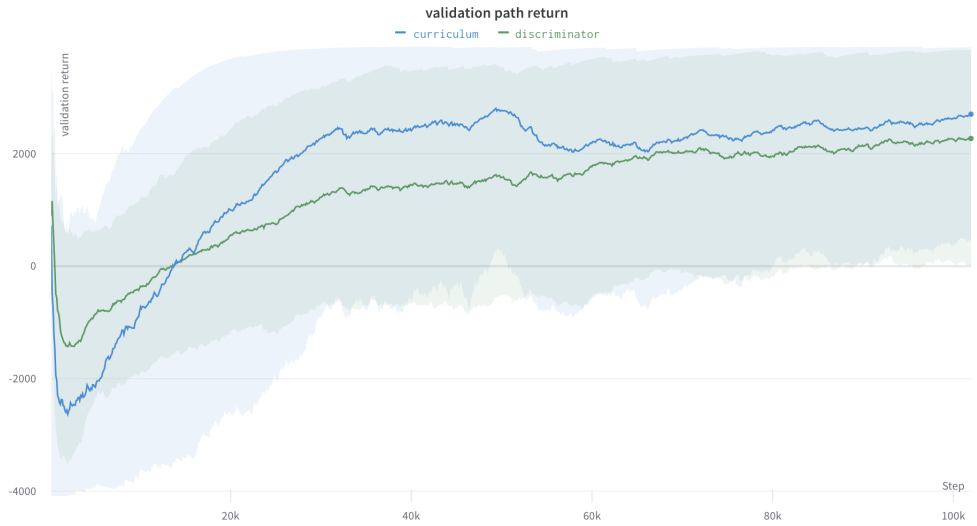


Figure 12.3. Validation path return over time. The blue curve represents the curriculum trained agent. It trains on the 20-wide discriminator for 500 epochs and on the 300-wide discriminator for the next 500. It is able to maximize the reward on the 20-wide discriminator much faster and also ends up achieving a much higher reward on the original DIAYN discriminator. The green curve represents the agent directly trained using the original discriminator and it achieves a lower reward in the end

Figure 12.4 shows a comparison of the returns achieved on the DIAYN discriminator after training. We see that on average, the curriculum trained agent achieves around 20% higher return than the agent trained directly on the DIAYN discriminator.

Further we can also see that it's not just the effect of the distilled discriminator alone but the combined result of the curriculum, from figure 12.5.

This is to be expected, as maximising the distilled discriminator is not the same as maximising the original task reward. The distilled discriminators are much easier to maximize and the agent achieves very high scores (Last column, figure 12.5) on the distilled discriminators. But this does not translate to actual task performance. At the cost of losing the task definition, we are able to maximize the reward function much more easily.

seed	DIAYN	discriminator	curriculum	% increase
1	3838	2838	3329	17
2	3755	1614	2071	28
3	3747	2133	2576	21
4	3847	2430	2821	16
5	3844	1414	1689	19
Average:	3806.2	2085.8	2497.2	20

Figure 12.4. Comparison of average validation path returns after training for all 5 seeds. The returns are computed using the DIAYN discriminator. The DIAYN agent scores the highest. The agent retrained using the curriculum scores on average 20% higher than the discriminator trained agent. This clearly shows that the curriculum is better than directly training on the discriminator as expected.

seed	distilled discriminator	curriculum	% increase	distilled score
1	2917	3329	14	3677
2	1364	2071	52	3048
3	2230	2576	16	3386
4	2115	2821	33	3549
5	1158	1689	46	2389
Average:	1957	2497.2	28	3210

Figure 12.5. Comparison of average validation path returns after training for all 5 seeds. The second column corresponds to the agent trained solely using the 20-wide distilled discriminator and we can see that the return is much lower than with the curriculum returns. The last column shows the return as measured by the distilled discriminator. This is higher than the return achieved by training on the original discriminator showing that the distilled discriminator is indeed much easier to maximize but the maximum does not correspond to a maximum of the original discriminator.

We see similar results when training on Ant-v1 (figure 12.6), although the difference is less pronounced. In general Ant-v1 requires training for at least 3000 epochs but the training has been curtailed around 1000 epochs due to compute time constraints. Further DIAYN seems to be less stable[2] on Ant-v1.

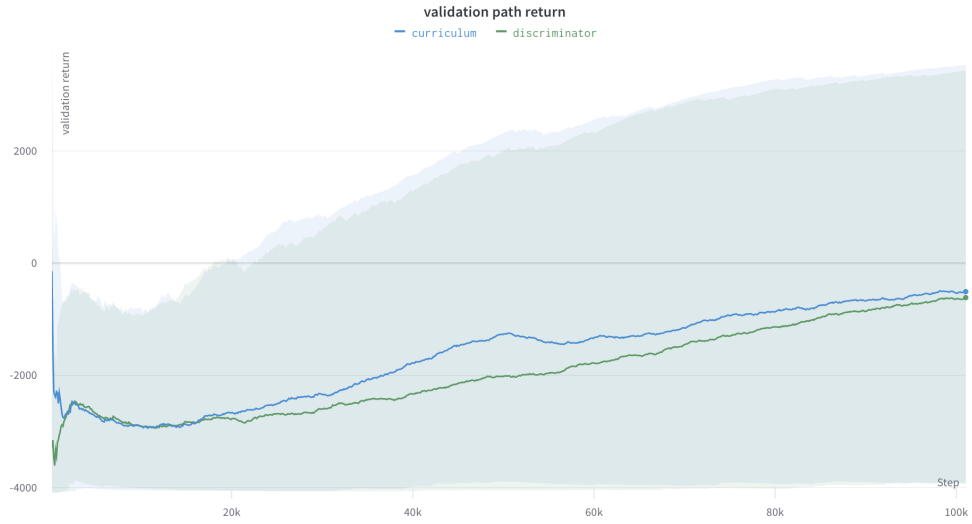


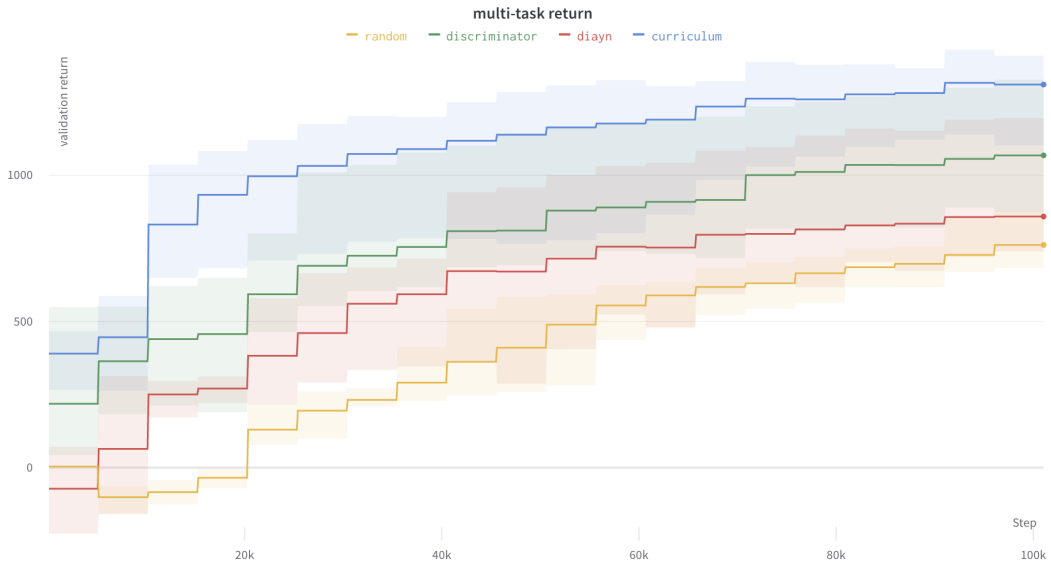
Figure 12.6. Validation path return over time. The blue curve represents the curriculum trained agent, which is able to maximize the return on the 20-wide discriminator faster and also ends up achieving a higher return on the original DIAYN discriminator. The green curve represents the agent directly trained using the discriminator and it achieves a lower return in the end

seed	DIAYN	discriminator	curriculum	% increase
1	3519	1071	1458	36
2	-3289	-2648	-2721	-3
3	3813	-873	-638	27
4	-426	-2303	-1885	18
5	3881	1688	1255	-26
Average:	1499.6	-613	-506.2	17

Figure 12.7. Comparison of average validation path returns after training on all 5 seeds. DIAYN training for Ant-v1 seems to be less stable than for HalfCheetah-v1 but there is an average improvement of 17% in returns using curriculum.

[2]Seeds 2 and 4 become unstable with the value functions oscillating wildly: figure 12.7

In both cases the DIAYN agent scores much higher as the discriminator is trained based on their state distribution directly. What this means is the simply scoring highly on the discriminator is not an indicator of whether the agent will generalise well to downstream tasks. This suggests that the discriminators aren't necessarily the best rewards to maximize. Yet the curriculum trained agents show an improvement both on the discriminator rewards and in generalization as can be seen by the performance on the half cheetah multi-task HERF benchmark in figure 12.8.



Comparison of different initializations on the multi-task HERF benchmark.

Figure 12.8. The blue curve represents the curriculum trained agent. The red curve is the DIAYN agent. Green is the discriminator retrained agent and Yellow is the randomly initialized agent.

But since the different tasks of the multi-skill HERF benchmark have a different maximum reward that is achieved by the agent, it is important to verify that these results are corroborated by each task individually. Figures 12.9-12.14 verify this on one each of the six classes of tasks. The results corroborate for most of the tasks except for the flipping task where the random agent performs better than DIAYN. The reason for this is likely the fact that the flipping task requires the agent to learn to ignore the angular position[3] of the agent. This might be easy for the randomly trained agent but difficult for the DIAYN agent and its derivatives as they have explicitly learned to pay attention to it and must now unlearn it.

[3]This is not a problem for the running tasks as the x-position is not present in the state vector.

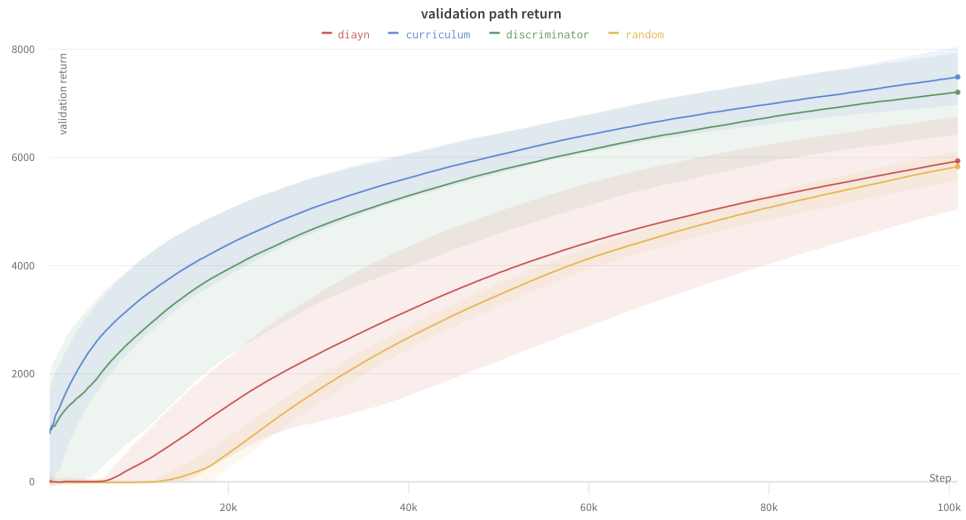


Figure 12.9. Comparison of returns on running forward task. All curves represent the mean of 5 seeds. The envelopes represent the max and min.

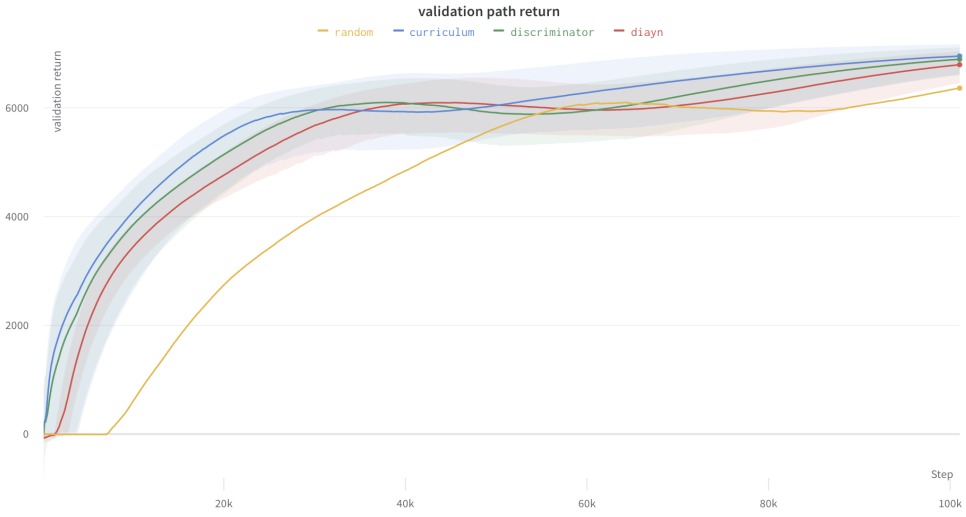


Figure 12.10. Comparison of returns on running backward (speed -8 m/s) task. The yellow curve just corresponds to 1 seed. (randomly initialised agents exhibit much lower variance). All other curves are 5 seed averages.

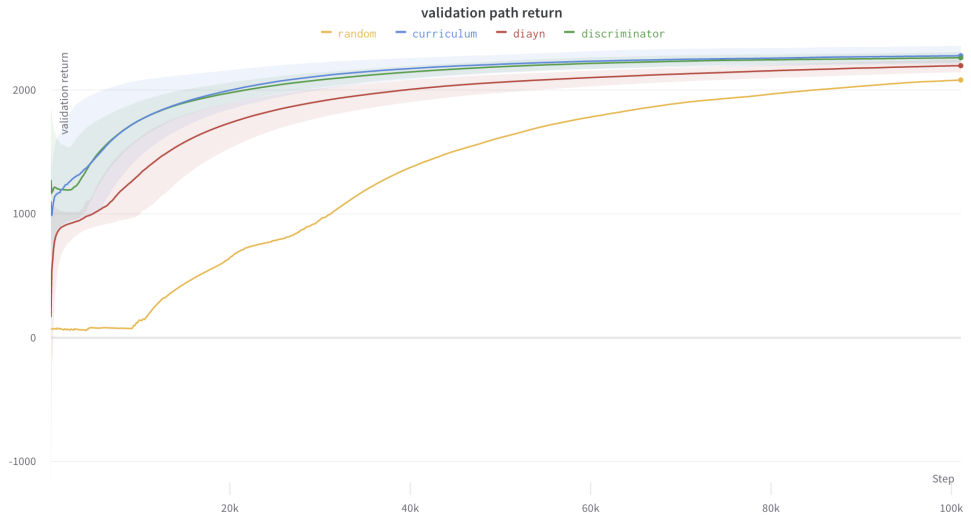


Figure 12.11. Comparison of returns on front foot hopping (angle 60 degrees) task. The yellow curve just corresponds to 1 seed. All other curves are 5 seed averages. Curriculum is only slightly better than the retrained agent.

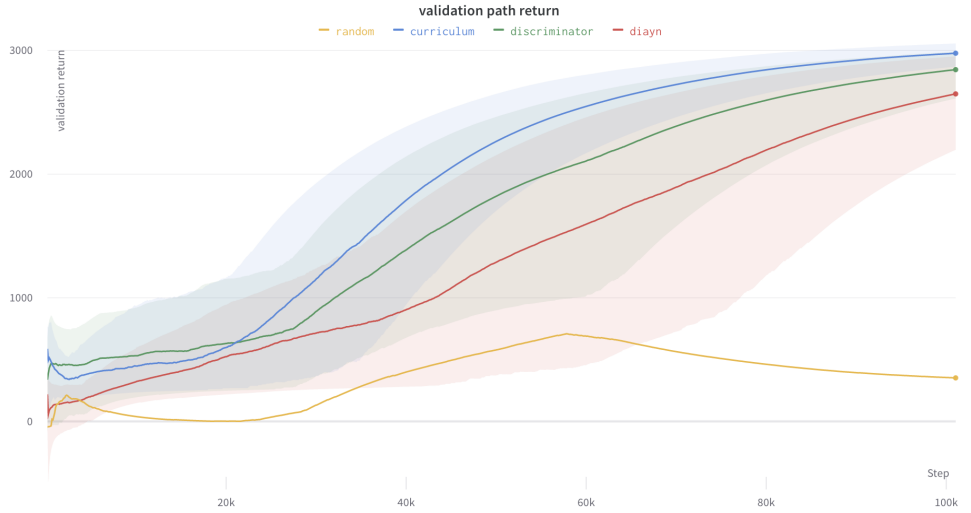


Figure 12.12. Comparison of returns on back foot hopping (angle 90 degrees) task. The yellow curve just corresponds to 1 seed. All other curves are 5 seed averages. Curriculum is significantly better than discriminator trained agent.

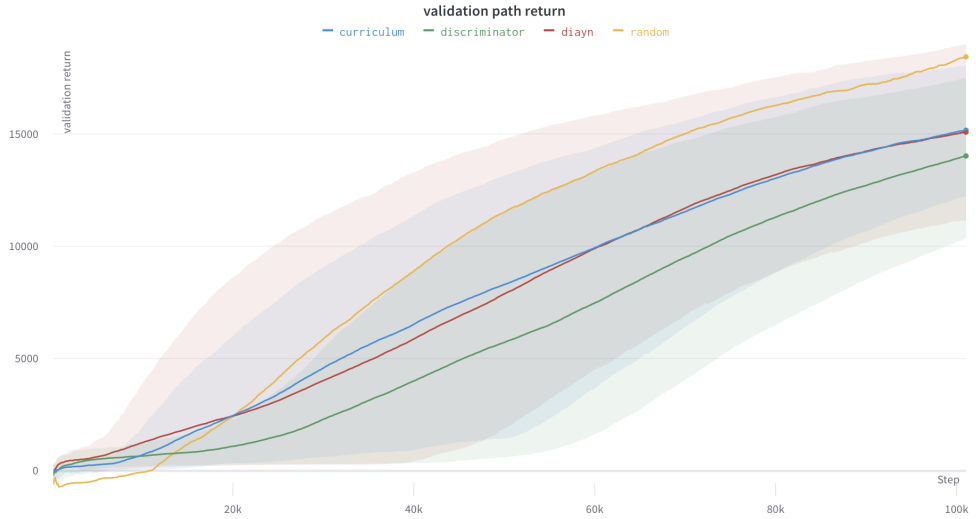


Figure 12.13. Comparison of returns on forward flipping (speed 8 rad/s) task. The yellow curve just corresponds to 1 seed. All other curves are 5 seed averages. Random agent is the best. Curriculum is as good as the DIAYN agent.

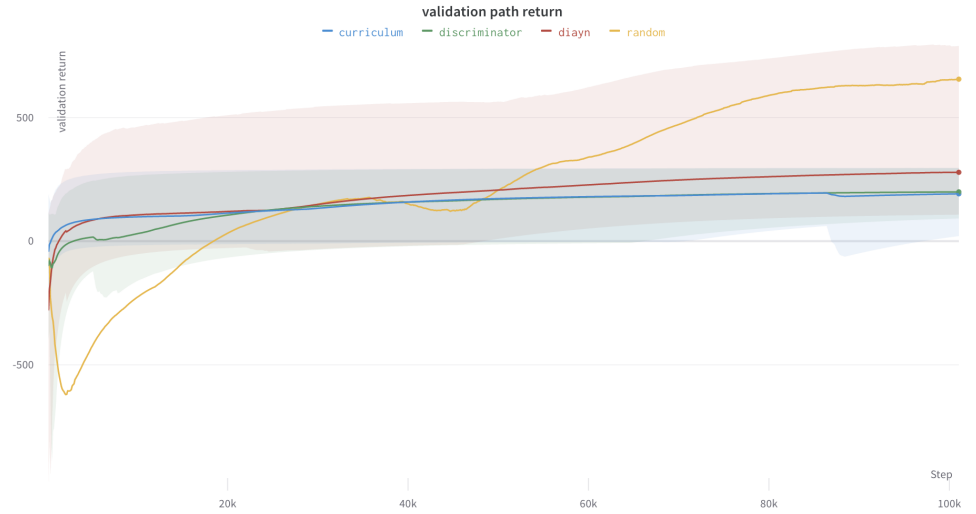


Figure 12.14. Comparison of returns on backward flipping (speed 8 rad/s) task. The yellow curve just corresponds to 1 seed. All other curves are 5 seed averages. Random agent is the best. Most of the seeds, except for random, do not learn the task at all as it is quite difficult.

Thus we see that apart from flipping, which seems to be the most difficult task, all others show the same trend with the best results obtained by the curriculum trained agent. Based on the differing advantage margins, we can conclude that *not every task may benefit from this curriculum, especially the ones where the random agent already scores as much or better than the DIAYN agent.*

Further we can also see the same trend on the Ant-v1 running task in figure 12.15.

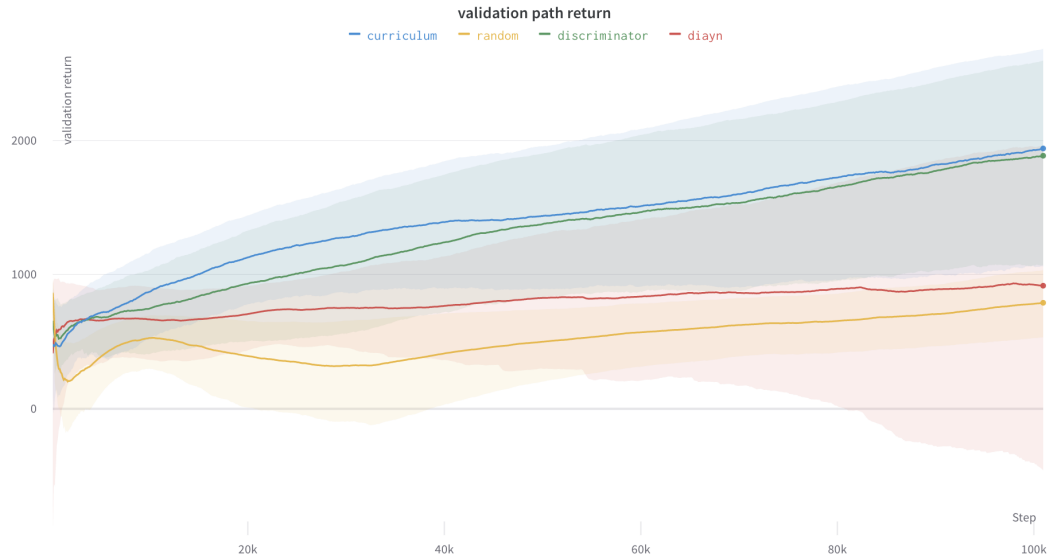


Figure 12.15. Finetuning results on the Ant-v1 running task. Curriculum performs the best although by a slight margin.

From this we can conclude that the training process used to maximize the rewards is important and not just the final return obtained on the discriminator. Since the DIAYN agent scores much higher on the discriminator but does not generalise well. Further, it seems, easing the training process is not the only advantage of the curriculum. The distilled discriminator helps teach skills that even the original discriminator fails to teach. Figure 12.16 shows the same running skill from the retraining chapter, but trained using the curriculum.

We see that the running skill learned by the curriculum trained agent obtains a constant maximum reward on the distilled discriminator but has oscillations on the original discriminator. Thus, we can infer that the distillation can teach skills that the original agent doesn't learn by itself.

Figure 12.16. Reward-per-time-step from the DIAYN discriminator and the distilled discriminator on the curriculum agent over two episodes of 100 steps each. We see that the running skill learned by the agent causes oscillations in the DIAYN discriminator but is a maximum of the distilled discriminator.

Chapter 13

Conclusion

So far we have seen that in the context of unsupervised skill discovery, retraining has advantages in the form of generalization. Further, we have also seen how to construct a curriculum using distilled discriminators to further enhance the generalization benefits. More specifically, We have seen that distilling reward functions into less wider networks produces reward functions that are easier to maximize and serve as a stepping stone for maximising the original reward function, which may be difficult to maximize[1]. Moreover, this is a simple way to automatically generate curricula.

DIAYN is one of the simplest unsupervised skill discovery algorithms and we have seen that it doesn't always work very well, like in the Ant-v1 environment. Potentially, this distillation method may produce better results when applied to other algorithms like Dynamics-Aware Unsupervised Discovery of Skills (DADS) [Sharma et al., 2020] or CARML[Jabri et al., 2019]which work well even in high dimensional state spaces. Further, I have only shown that the simplest two level curriculum works. But it remains to be seen whether this can be extended to multiple levels of distilled discriminators. Further, DIAYN uses a state discriminator and because it acts in the space of state embeddings, having an easy-to-maximize reward in the space of state embeddings is of limited utility as skills normally correspond to sequences of states and actions. This method may work much better with trajectory discriminators which work on trajectory embeddings, provided the trajectory latent space is restricted to feasible trajectories.

To understand the ease of maximisation of reward functions, the approach to visualising high dimensional reward functions in figure 12.2, by plotting reward along straight lines from a start state to a task state, is useful but limited. This is because there is no guarantee of the feasibility of the states that lie along the straight line due to physical limitations of the environment. However since the straight line is the shortest path, it serves as a heuristic. Ideally we would like to see the reward profile along the learning path of the agent so that at any point the agent experiences a consistent gradient without local optima, plateaus or saddle points. More work is needed to understand and quantify what makes a reward function ideal.

The curriculum here, serves two important purposes. It makes learning the task easier and also improves the generalization ability of the agent. I believe this is key in iterative frameworks like CARML [Jabri et al., 2019], where we want the process to continue until the agent achieves

[1] I have not performed hyper-parameter sweeps since the algorithms are computationally intensive and the results are already conclusive. Better results may be obtained with hyper-parameter optimisation.

human-level capabilities or beyond[2]. But in this case, the maximization step is a limiting[3] factor, as the agent is barely able to fit the objective once it becomes complex enough. Viewed in a dual way, the adapting task distribution does not serve as a proper curriculum for the agent. It remains to be seen whether the method I describe may be incorporated into this process.

This method relies on a very simple idea that when fitting complicated data, the less wide networks produce decision boundaries that cover a much larger area than necessary. As we have seen, in certain situations, this can be a feature, rather than a bug. However, it is possible to model our requirements of the reward function more rigorously. One might consider a surrogate objective of maximizing the entropy of the discriminator. This would ensure that the discriminator spreads it's distribution out among multiple classes. Thus, spreading out the skill specific reward curve over larger areas of the state space. However this does not guarantee that we will not have local optima or plateaus along the learning path. A more useful surrogate objective might be explicitly minimizing the Dirichlet energy or the Total Variation (TV) of the reward function. Dirichlet Energy is defined as the integral of the squared norm of the gradient of the function over the entire space. TV uses the absolute value of the squared norm. Both are measures of how variable the function is. This has popularly been used as a regularization term in variational image denoising[Rudin et al., 1992]. More recently it has also been used to study the expressiveness of node embeddings in graph networks.[Cai and Wang, 2020] This is a more mathematically precise formulation of what we want and may yield better results. But this also has limitations of precision, since we need to minimise the variability only along *feasible* trajectories. There are definitely better mathematical formulations that can achieve this and they are left for future work.

[2]Of course, the benchmark must support the demonstration of such capabilities.

[3]There are other limiting factors as well.

Bibliography

Marcin Andrychowicz, Filip Wolski, Alex Ray, Jonas Schneider, Rachel Fong, Peter Welinder, Bob McGrew, Josh Tobin, Pieter Abbeel, and Wojciech Zaremba. Hindsight Experience Replay. *arXiv:1707.01495 [cs]*, February 2018.

Jordan T. Ash and Ryan P. Adams. On the Difficulty of Warm-Starting Neural Network Training. *arXiv:1910.08475 [cs, stat]*, October 2019.

Chen Cai and Yusu Wang. A Note on Over-Smoothing for Graph Neural Networks. *arXiv:2006.13318 [cs, stat]*, June 2020.

Mathilde Caron, Piotr Bojanowski, Armand Joulin, and Matthijs Douze. Deep Clustering for Unsupervised Learning of Visual Features. *arXiv:1807.05520 [cs]*, March 2019.

G. Cybenko. Approximation by superpositions of a sigmoidal function. *Mathematics of Control, Signals and Systems*, 2(4):303–314, December 1989. ISSN 1435-568X. doi: 10.1007/BF02551274.

Jeffrey L. Elman. *Learning and Development in Neural Networks: The Importance of Starting Small*. 1993.

Benjamin Eysenbach and Sergey Levine. If MaxEnt RL is the Answer, What is the Question? *arXiv:1910.01913 [cs, stat]*, October 2019.

Benjamin Eysenbach, Abhishek Gupta, Julian Ibarz, and Sergey Levine. Diversity is All You Need: Learning Skills without a Reward Function. In *International Conference on Learning Representations*, September 2018.

Meng Fang, Tianyi Zhou, Yali Du, Lei Han, and Zhengyou Zhang. Curriculum-guided Hindsight Experience Replay. In H. Wallach, H. Larochelle, A. Beygelzimer, F. d\textquotesingle Alché-Buc, E. Fox, and R. Garnett, editors, *Advances in Neural Information Processing Systems 32*, pages 12623–12634. Curran Associates, Inc., 2019.

Chelsea Finn, Pieter Abbeel, and Sergey Levine. Model-Agnostic Meta-Learning for Fast Adaptation of Deep Networks. *arXiv:1703.03400 [cs]*, July 2017.

Carlos Florensa, David Held, Markus Wulfmeier, Michael Zhang, and Pieter Abbeel. Reverse Curriculum Generation for Reinforcement Learning. *arXiv:1707.05300 [cs]*, July 2018.

Alex Graves, Marc G. Bellemare, Jacob Menick, Remi Munos, and Koray Kavukcuoglu. Automated Curriculum Learning for Neural Networks. *arXiv:1704.03003 [cs]*, April 2017.

Karol Gregor, Danilo Jimenez Rezende, and Daan Wierstra. Variational Intrinsic Control. *arXiv:1611.07507 [cs]*, November 2016.

Suriya Gunasekar, Blake Woodworth, Srinadh Bhojanapalli, Behnam Neyshabur, and Nathan Srebro. Implicit Regularization in Matrix Factorization. *arXiv:1705.09280 [cs, stat]*, May 2017.

Abhishek Gupta, Benjamin Eysenbach, Chelsea Finn, and Sergey Levine. Unsupervised Meta-Learning for Reinforcement Learning. *arXiv:1806.04640 [cs, stat]*, December 2019.

Tuomas Haarnoja, Aurick Zhou, Pieter Abbeel, and Sergey Levine. Soft Actor-Critic: Off-Policy Maximum Entropy Deep Reinforcement Learning with a Stochastic Actor. *arXiv:1801.01290 [cs, stat]*, August 2018.

Nicolas Heess, Greg Wayne, David Silver, Timothy Lillicrap, Yuval Tassa, and Tom Erez. Learning Continuous Control Policies by Stochastic Value Gradients. *arXiv:1510.09142 [cs]*, October 2015.

Kurt Hornik, Maxwell Stinchcombe, and Halbert White. Multilayer feedforward networks are universal approximators. *Neural Networks*, 2(5):359–366, January 1989. ISSN 0893-6080. doi: 10.1016/0893-6080(89)90020-8.

Nicholas V. Hud and David M. Fialho. RNA nucleosides built in one prebiotic pot. *Science*, 366 (6461):32–33, 2019.

Maximilian Igl, Gregory Farquhar, Jelena Luketina, Wendelin Boehmer, and Shimon Whiteson. The Impact of Non-stationarity on Generalisation in Deep Reinforcement Learning. *arXiv:2006.05826 [cs, stat]*, June 2020.

Allan Jabri, Kyle Hsu, Ben Eysenbach, Abhishek Gupta, Sergey Levine, and Chelsea Finn. Unsupervised Curricula for Visual Meta-Reinforcement Learning. *arXiv:1912.04226 [cs]*, December 2019.

Daniel Kahneman and Shane Frederick. Representativeness revisited: Attribute substitution in intuitive judgment. *Heuristics and biases: The psychology of intuitive judgment*, 49:49–81, January 2002. doi: 10.1017/CBO9780511808098.004.

Sergey Levine. Reinforcement Learning and Control as Probabilistic Inference: Tutorial and Review. *arXiv:1805.00909 [cs, stat]*, May 2018.

Yuanzhi Li, Tengyu Ma, and Hongyang Zhang. Algorithmic Regularization in Over-parameterized Matrix Sensing and Neural Networks with Quadratic Activations. *arXiv:1712.09203 [cs, math, stat]*, February 2019.

Long-Ji Lin. Self-improving reactive agents based on reinforcement learning, planning and teaching. *Machine Learning*, 8(3):293–321, May 1992. ISSN 1573-0565. doi: 10.1007/BF00992699.

Robert K. Merton. The Matthew Effect in Science: The reward and communication systems of science are considered. *Science*, 159(3810):56–63, January 1968. ISSN 0036-8075, 1095-9203. doi: 10.1126/science.159.3810.56.

Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A. Rusu, Joel Veness, Marc G. Bellemare, Alex Graves, Martin Riedmiller, Andreas K. Fidjeland, Georg Ostrovski, Stig Petersen, Charles Beattie, Amir Sadik, Ioannis Antonoglou, Helen King, Dharshan Kumaran, Daan Wierstra, Shane Legg, and Demis Hassabis. Human-level control through deep reinforcement learning. *Nature*, 518(7540):529–533, February 2015. ISSN 1476-4687. doi: 10.1038/nature14236.

Vaishnavh Nagarajan and J. Zico Kolter. Generalization in Deep Networks: The Role of Distance from Initialization. *arXiv:1901.01672 [cs, stat]*, January 2019.

OpenAI, Ilge Akkaya, Marcin Andrychowicz, Maciek Chociej, Mateusz Litwin, Bob McGrew, Arthur Petron, Alex Paino, Matthias Plappert, Glenn Powell, Raphael Ribas, Jonas Schneider, Nikolas Tezak, Jerry Tworek, Peter Welinder, Lilian Weng, Qiming Yuan, Wojciech Zaremba, and Lei Zhang. Solving Rubik's Cube with a Robot Hand. *arXiv:1910.07113 [cs, stat]*, October 2019.

Rémy Portelas, Cédric Colas, Lilian Weng, Katja Hofmann, and Pierre-Yves Oudeyer. Automatic Curriculum Learning For Deep RL: A Short Survey. *arXiv:2003.04664 [cs, stat]*, May 2020a.

Rémy Portelas, Katja Hofmann, and Pierre-Yves Oudeyer. Trying AGAIN instead of Trying Longer: Prior Learning for Automatic Curriculum Learning. *arXiv:2004.03168 [cs, stat]*, April 2020b.

Sebastien Racaniere, Andrew K. Lampinen, Adam Santoro, David P. Reichert, Vlad Firoiu, and Timothy P. Lillicrap. Automated curricula through setter-solver interactions. *arXiv:1909.12892 [cs, stat]*, January 2020.

Chris Reinke, Mayalen Etcheverry, and Pierre-Yves Oudeyer. Intrinsically Motivated Discovery of Diverse Patterns in Self-Organizing Systems. *arXiv:1908.06663 [cs, stat]*, February 2020.

Sebastian Risi and Julian Togelius. Increasing Generality in Machine Learning through Procedural Content Generation. *arXiv:1911.13071 [cs]*, March 2020.

Leonid I. Rudin, Stanley Osher, and Emad Fatemi. Nonlinear total variation based noise removal algorithms. *Physica D: Nonlinear Phenomena*, 60(1):259–268, November 1992. ISSN 0167-2789. doi: 10.1016/0167-2789(92)90242-F.

Christoph Salge, Cornelius Glackin, and Daniel Polani. Empowerment – an Introduction. *arXiv:1310.1863 [nlin]*, October 2013.

Tom Schaul, John Quan, Ioannis Antonoglou, and David Silver. Prioritized Experience Replay. *arXiv:1511.05952 [cs]*, February 2016.

J. Schmidhuber. Curious model-building control systems. In *[Proceedings] 1991 IEEE International Joint Conference on Neural Networks*, pages 1458–1463 vol.2, November 1991. doi: 10.1109/IJCNN.1991.170605.

Juergen Schmidhuber. Optimal Ordered Problem Solver. *arXiv:cs/0207097*, December 2002.

Juergen Schmidhuber. Driven by Compression Progress: A Simple Principle Explains Essential Aspects of Subjective Beauty, Novelty, Surprise, Interestingness, Attention, Curiosity, Creativity, Art, Science, Music, Jokes. *arXiv:0812.4360 [cs]*, April 2009.

Juergen Schmidhuber. Generative Adversarial Networks are Special Cases of Artificial Curiosity (1990) and also Closely Related to Predictability Minimization (1991). *arXiv:1906.04493 [cs]*, April 2020a.

Juergen Schmidhuber. Generative Adversarial Networks are Special Cases of Artificial Curiosity (1990) and also Closely Related to Predictability Minimization (1991). *arXiv:1906.04493 [cs]*, April 2020b.

Jürgen Schmidhuber. Learning Complex, Extended Sequences Using the Principle of History Compression. *Neural Computation*, 4(2):234–242, March 1992. ISSN 0899-7667. doi: 10.1162/neco.1992.4.2.234.

Jürgen Schmidhuber. POWERPLAY: Training an Increasingly General Problem Solver by Continually Searching for the Simplest Still Unsolvable Problem. *arXiv:1112.5309 [cs]*, November 2012.

Archit Sharma, Shixiang Gu, Sergey Levine, Vikash Kumar, and Karol Hausman. Dynamics-Aware Unsupervised Discovery of Skills. *arXiv:1907.01657 [cs, stat]*, February 2020.

Pranav Shyam, Wojciech Jaśkowski, and Faustino Gomez. Model-Based Active Exploration. *arXiv:1810.12162 [cs, math, stat]*, June 2019.

David Silver, Thomas Hubert, Julian Schrittwieser, Ioannis Antonoglou, Matthew Lai, Arthur Guez, Marc Lanctot, Laurent Sifre, Dharshan Kumaran, Thore Graepel, Timothy Lillicrap, Karen Simonyan, and Demis Hassabis. Mastering Chess and Shogi by Self-Play with a General Reinforcement Learning Algorithm. *arXiv:1712.01815 [cs]*, December 2017a.

David Silver, Julian Schrittwieser, Karen Simonyan, Ioannis Antonoglou, Aja Huang, Arthur Guez, Thomas Hubert, Lucas Baker, Matthew Lai, Adrian Bolton, Yutian Chen, Timothy Lillicrap, Fan Hui, Laurent Sifre, George van den Driessche, Thore Graepel, and Demis Hassabis. Mastering the game of Go without human knowledge. *Nature*, 550(7676):354–359, October 2017b. ISSN 1476-4687. doi: 10.1038/nature24270.

Karl Sims. Evolving 3D Morphology and Behavior by Competition. *Artificial Life*, 1(4):353–372, July 1994. ISSN 1064-5462. doi: 10.1162/artl.1994.1.4.353.

Rupesh Kumar Srivastava, Bas R. Steunebrink, and Jürgen Schmidhuber. First Experiments with PowerPlay. *arXiv:1210.8385 [cs]*, October 2012.

Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning: An Introduction*. A Bradford Book, Cambridge, MA, USA, 2018. ISBN 0-262-03924-9.

Emanuel Todorov, Tom Erez, and Yuval Tassa. MuJoCo: A physics engine for model-based control. In *2012 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 5026–5033, October 2012. doi: 10.1109/IROS.2012.6386109.

Nir Vulkan. *An Economist's Perspective on Probability Matching*. 1998.

Rui Wang, Joel Lehman, Jeff Clune, and Kenneth O. Stanley. Paired Open-Ended Trailblazer (POET): Endlessly Generating Increasingly Complex and Diverse Learning Environments and Their Solutions. *arXiv:1901.01753 [cs]*, February 2019.

Daphna Weinshall, Gad Cohen, and Dan Amir. Curriculum Learning by Transfer Learning: Theory and Experiments with Deep Networks. *arXiv:1802.03796 [cs]*, June 2018.

D.H. Wolpert and W.G. Macready. No free lunch theorems for optimization. *IEEE Transactions on Evolutionary Computation*, 1(1):67–82, April 1997. ISSN 1941-0026. doi: 10.1109/4235.585893.