BROWN UNIVERSITY

UNDERGRADUATE HONORS THESIS

Learning Feature Extraction for Transfer from Simulation to Reality

---

Author:
Josh ROY

Advisor and First Reader:
Dr. George KONIDARIS
Second Reader:
Dr. Stefanie TELLEX

A thesis submitted in fulfillment of the requirements for Honors in Computer Science

May 1, 2019

"You could claim that moving from pixelated perception, where the robot looks at sensor data, to understanding and predicting the environment is a Holy Grail of artificial intelligence"

Sebastian Thrun

BROWN UNIVERSITY

Abstract

George KONIDARIS
Department of Computer Science

Honors in Computer Science

Learning Feature Extraction for Transfer from Simulation to Reality

by Josh ROY

Deep reinforcement learning is able to solve complex visual and control tasks in simulation, but not in reality. The ability to transfer learned policies between simulation and reality will enable robots to better aid humans by learning task specific policies using only a few real-world interactions.

In this thesis, I focus on the task of transferring learned policies between tasks with different visual features. I present three methods. 1) A Markov assumption based autoencoder, 2) A generative adversarial network, and 3) A non-Markov autoencoder. I empirically compare each of these three methods and their results and compare them to the state of the art in transfer learning for visual reinforcement learning tasks.

Acknowledgements

Firstly, I would like to thank my project advisor, Professor George Konidaris. Thank you for your guidance in theoretical setup, problem formulation, and debugging of loss functions. I would like to thank my research advisor and reader, Professor Stefanie Tellex for the advice on research projects and academic interests over the past four years. I would also like to thank my academic advisor, Professor Tom Doeppner, for thoughtful answers to all my questions and advice on my academic career. Secondly, I would like to thank my mother and sister for supporting me through my college career, helping me through difficult decisions, and celebrating my accomplishments. Finally, I would like to thank my friends for listening to all of my thoughts on Computer Science, both related and unrelated to my thesis.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

In recent years, neural networks have shown drastic improvements in fields such as supervised learning, unsupervised learning, and reinforcement learning. Given their ability to draw associations from massive datasets, neural networks have surpassed most classical methods for tasks such as classification across many categories [9], semantic segmentation of images [10], and games such as Go or Chess [16]. While deep learning has yielded significant improvement in agents that solve specific tasks, some at a superhuman level, they do not act and react in the real world, as humans do. Two things separate these machines from people.

First, humans can tolerate slight differences in tasks without difficulty. For example, if a human who knows how to drive a car is asked to drive a new model of car that they have not seen before, they are not expected to re-learn to drive from scratch. Instead, they are expected to find the accelerator and steering wheel, and use their knowledge about cars to drive the new car. On the other hand, a deep reinforcement learning agent that learns to pick up a blue triangular prism using a robot arm will have to retrain almost from scratch in order to pick up another object, such as a red block. The ability to apply skills learned in one task to another related task is called transfer learning [17].

In transfer learning, an agent is typically trained on n related tasks, and its performance is tested on an unseen $n+1$th task. The first n tasks are referred to as the *source* tasks, and the $n+1$th task is referred to as the *target* task. The number n of source tasks varies depending on the specific method of transfer learning utilized. There are two primary branches of transfer learning: 0-shot transfer and k-shot transfer. A *shot* is defined as the number of samples from the target task that the agent is trained on when transfering. More specifically, a 0-shot transfer learning method would train its agent on the n source tasks, and the test on the target task, while a k-shot transfer learning method would train its agent on the source tasks, then train on k samples from target task, and finally test on the target task.

Second, humans have the ability to reason about what *would* happen in a task given some action. In other words, they have the ability to imagine performing a task. This gives them the ability to "train" on that task in their heads, resulting in intelligent, though not necessarily optimal, actions before even starting the task. This imagination is analogous to an agent training in simulation and transferring that training to reality. Such an approach would enable neural networks to obtain the massive amount of data and experience needed to train.

If photorealistic simulators with perfect physics engines existed, it would be easy to train a deep reinforcement learning agent in simulation and transfer seamlessly to reality. Unfortunately, such a realistic simulator does not exist at the current time. Furthermore, it seems that any future simulator will have problems. For example, a simulator that idealizes all objects as rigid bodies will not be able to capture the

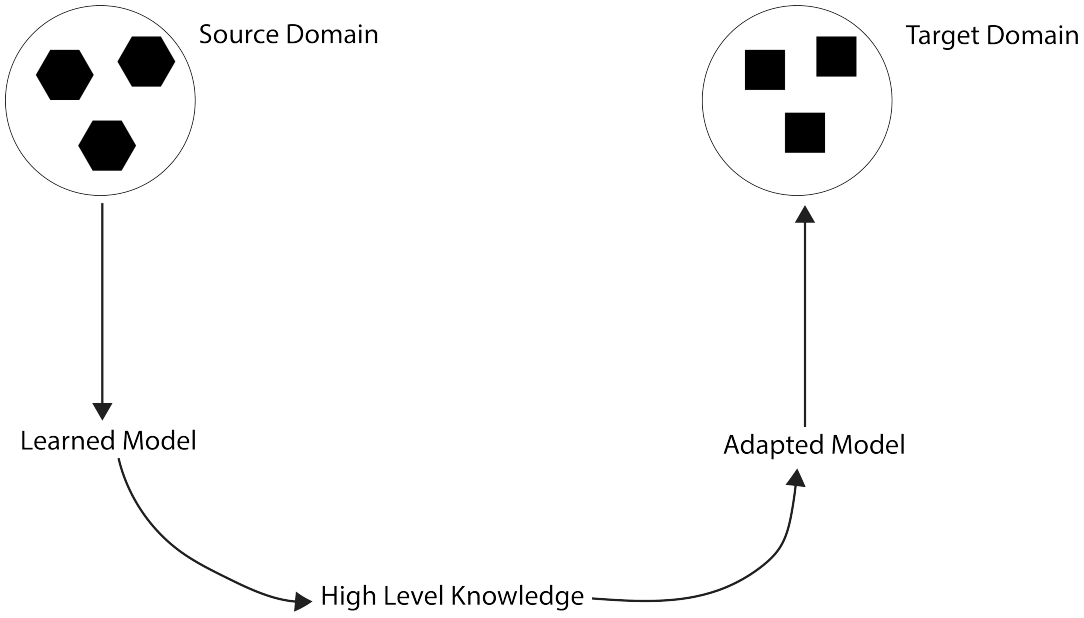


FIGURE 1.1: Transfer Learning

slight ability for a metal pipe to bend, which might be important for assembly line robots that build cars.

Given a task in reality and a corresponding task in simulation, we can model the simulation to reality transfer problem as a transfer learning problem. The task in simulation is the source task and the task in reality is the target task. Under this model, we can train a deep reinforcement learning agent on the source task, simulation, and transfer, either with 0 shots or k shots, to the target task, reality.

In this thesis, I focus on enabling deep reinforcement learning methods to transfer between related tasks, working towards transfer from simulation to reality.

Chapter 2

Background and Related Work

2.1 Background

2.1.1 MDPs

In the previous section, I discussed transfer between tasks. I will now define these terms more precisely.

Formally, a Markov Decision Process (MDP) is represented by a 5-tuple

$$(S, A, T, R, \gamma)$$

where S is the set of states, A is the set of possible actions, $T(s, a, s')$ is the probability of transitioning from a state to another state given an action, $R(s, a, s')$ is the reward for transitioning from a state to another state given an action, and γ is the discount factor.

An agent is an intelligent system that acts in a task. This agent could be a human or an artificial intelligence. The agent acts in an MDP by following the below steps.

1. Start in a state s_t
2. Pick an action a_t
3. Probabilistically transition to the next state s_{t+1} based on the transition function $T(s_{t+1}|s_t, a)$
4. Observe a reward r_t for taking action a_t at state s_t and transitioning to state s_{t+1} based on the reward function $R(s_t, a, s_{t+1})$.
5. If the task is completed or failed, do nothing. Otherwise, go to step 1.

An episode as one run of the MDP, starting from the starting state and ending once the task is completed or failed. This includes all states, actions, and observed rewards.

2.1.2 Deep Reinforcement Learning

To solve a task, the goal of the agent is to maximize the reward it gains throughout all of its actions on the task. There are two methods to solve these MDPs. Both of these methods generate a policy $\pi(a_t|s_t)$, which represents the probability that the agent should take action a_t from state s_t. To decide actions to take within the MDP, the agent randomly samples from the policy distribution.

1. **Model Based Learning:**

In model based reinforcement learning, the agent attempts to explicitly learn the transition and reward functions. It then maximizes the expected reward over the episode using these approximated functions.

2. **Model Free Learning:**

   In model free reinforcement learning, the agent does not ever learn an explicit model of the transition or reward functions of the task. Instead, it attempts to directly learn the policy $\pi(a_t|s_t)$.

Deep reinforcement learning is typically model free. The algorithm I use as comparison throughout my thesis is Deep Q-Network (DQN) [11]. This algorithm has successfully solved complex, visual tasks, such as Atari games, continuous control tasks, and classical reinforcement learning tasks. I briefly describe the algorithm below.

A deep neural network (DNN) is a machine learning technique used to approximate linear and non-linear functions. It takes an input tensor and applies a series of linear transformations and non-linear functions, called layers and activation functions, resulting in an output tensor. DNNs learn by changing the learnable parameters, called weights, of the layers to minimize a loss function calculated based on the input and output. In deep reinforcement learning, DNNs are often used to learn the policy, reward, or transition functions.

Solving an MDP involves learning a policy that maximizes reward over the entire episode. $V_\pi(s)$ is the value function: the expected reward for an episode, starting at state s, following policy π. $Q_\pi(s, a)$ is the Q function. It is the expected reward for following policy π in an episode after taking action a at state s. the DQN uses deep neural networks to learn the Q function and a policy π.

DQN follows the below steps, using a prediction network Qp and a target network Qt, which both output a Q-value.

1. For each timestep in each episode:
   (a) Predict the optimal policy based on Qp.
   (b) Observe the next state and reward.
   (c) Calculate the Q value of the next state based on Qt
   (d) Train Qp based on the Q value calculated by Qt
   (e) Every n steps, set $Qt = Qp$

2.1.3 Variational Autoencoders

Variational Autoencoders (VAE) are generative models based on deep neural networks that learn a mapping from an image I to a mean vector z_m and a standard deviation vector z_{std}. They also learn a mapping from a vector sampled from z_m, z_{std} z to a reconstructed image $\hat{I}$ in image space $\mathcal{I}$ [1].

A VAE has two neural networks:

1. **Encoder:** This maps from the input image I to z_m, z_{std}.

2. **Decoder:** This maps from a vector z sampled from z_m, z_{std} to the reconstructed image $\mathcal{I}$.

The variational autoencoder then optimizes the networks in series, by optimizing the following loss function.

The loss function for a varational autoencoder is

$$\mathcal{L} = \mathbb{E}_{q(z|x)}[\log p_\theta(x|z)] - (D_{KL}(q_\phi(z|x)||(p(z) - \epsilon)$$

where x is the input image, z is the sampled predicted latent vector, q is the decoder network, p is the encoder network, θ is the weights of q, ϕ is the weights of q, and KL is the function that calculates KL divergence.

The first term is the reconstruction loss between the input image I and the reconstructed image $\mathcal{I}$. This term forces the encoder to learn a latent space that is useful for reconstruction and the decoder to learn a mapping that correctly produces a reconstruction. The second term is the KL divergence, which is the distance from the distribution with mean z_m and standard deviation z_{std} to a standard normal distribution. This forces the output of the encoder to be a distribution that can be sampled from easily.

A beta varational autoencoder is an autoencoder with a slight change to the loss function:

$$\mathcal{L} = \mathbb{E}_{q(z|x)}[\log p_\theta(x|z)] - \beta(D_{KL}(q_\phi(z|x)||(p(z) - \epsilon)$$

where β is a scalar hyperparameter greater than or equal to one [4]. When $\beta = 1$, the beta VAE is a standard VAE. When $\beta > 1$, the weight on the KL term is higher, which encourages the latent distribution parameterized by z_m, z_{std} to be more similar to a standard normal and more disentangled. Specifically, it enourages the distribution to have one dimension for each independent factor, such as color or shape, in the latent space [4].

2.1.4 Generative Adversarial Networks

Generative Adversarial Networks (GANs) are generative models based on deep neural networks that learn a mapping from a latent space $z \in Z$ to an image space $\hat{I} \in \mathcal{I}$ [3]. A GAN is able to learn such a mapping with noise as the input z and few examples of $\hat{I}$, due to their dueling architecture.

A GAN has the following two networks:

1. **Discriminator:** The discriminator takes as input an image I that could either be generated by the generator or sampled from the true image space $\mathcal{I}$. It outputs a label indicating whether it thinks I came from $\mathcal{I}$ or the generator. This is a supervised learning task and has a corresponding loss function, such as cross entropy loss.

2. **Generator:** The generator takes as input a vector z whose elements are randomly sampled from a normal distribution. It then learns to map this vector to an image I.

By training these two networks simultaneously, the generator correctly learns a mapping from input vector z to image I and the discriminator correctly learns to differentiate real and generated images.

2.1.5 Transfer Learning and Renderers

The focus of my work is on visual tasks. In such MDPs, the state is an image. For example, if the task is to utilize a robot arm to pick up a mug, the state would be the image of the mug and part of the robot captured by the robot's camera. Let us denote this image (observed state) as S^O. Each of these visual tasks also has a factored state,

denoted S^T. For example, S^T may be the position of the mug and the robot arm in the task described above. For these tasks, there exists a mapping from S^T to S^O. I denote this as a function $Ren : S^T \rightarrow S^O$ and refer to it as a *Renderer*.

Consider a family of tasks M that all have the same actions, rewards, transitions, and discount factors, but different states. The tasks in this family have the same true states S^T, but different observed states, S^O, since they have different renderers. For example, one renderer may be the Mujoco renderer, another may the Unity game engine, a third may be the Unreal Engine, and the last may be real life. Humans have the ability to transfer the knowledge they learned across this family of tasks without any difficulty, but Deep RL agents do not.

The goal of my thesis is to give Deep RL agents this ability to transfer learned knowledge across such a family of tasks.

2.2 Related Work

Domain randomization, formal modeling of transfer, and learning a mapping between source and target tasks are the main methods that have been used to transfer learning across a family of tasks M generated by different renderers.

Reinforcement learning agents that employ machine learning methods, such as neural networks, are generalizable to an extent. There is some amount of transfer possible between domains without any explicit effort. For example, a deep reinforcement learning agent that flies a drone without avoiding walls will not fly straight into a wall in reality if trained properly in simulation [14]. Unfortunately, due to the current quality of state of the art graphics renderers, directly transferring complex visual tasks from simulation to reality is not yet possible. As depicted in figure 2.1, even relatively simple simulated scenes differ from their real world counterparts. Lighting, textures, and shadows are particularly different.

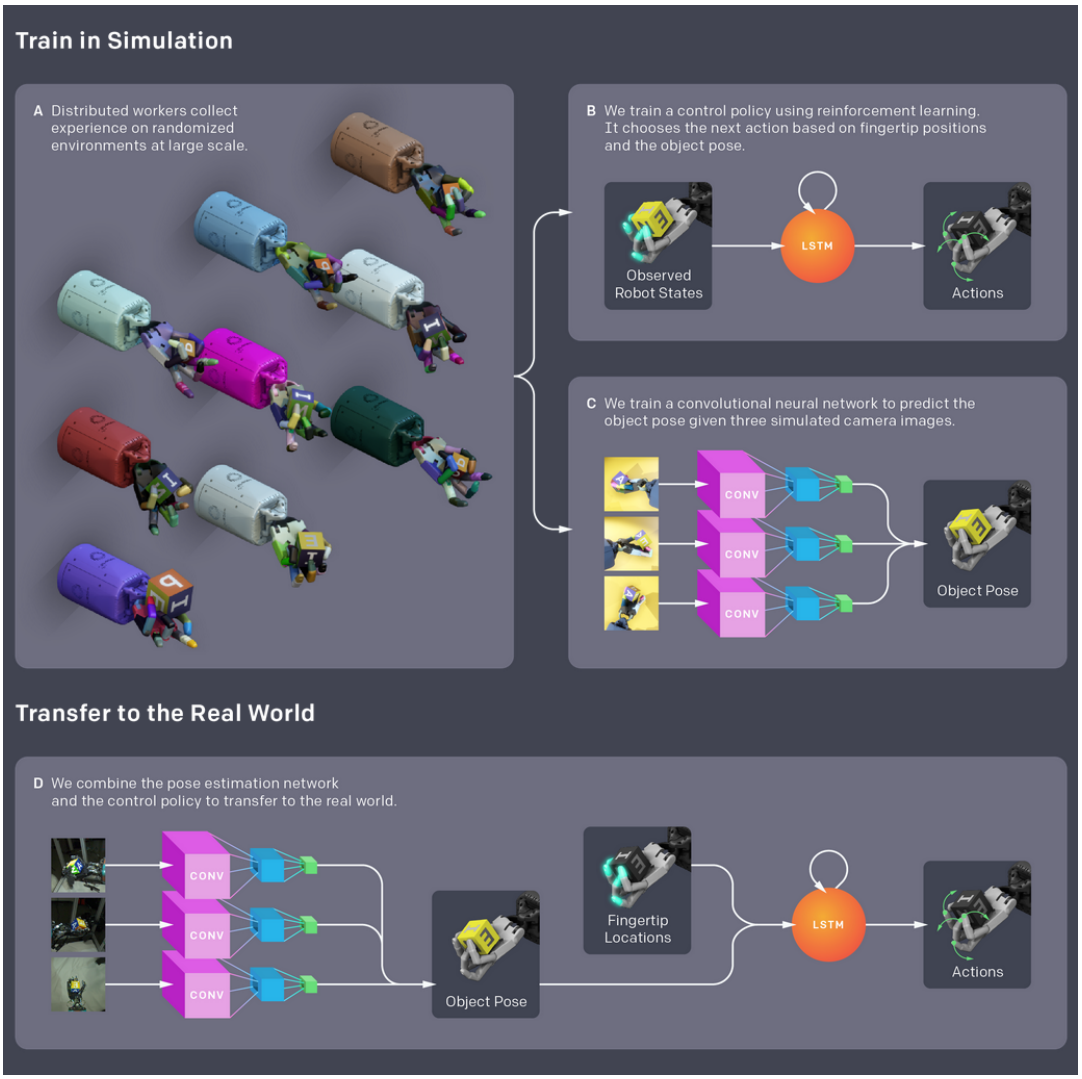
To overcome this reality gap, work has been done to increase the variation in the source domain [14, 13]. This allows the network to learn robustness to differences in renderers. Most popularly, this is done through domain randomization [14, 13]. Randomizing parameters such as gravity, motor strength, object weights, etc. enables the network to learn robustness to these physical parameters, accounting for differences between the transition dynamics of the source and target domains. Similarly, randomizing the colors and textures of the visual input to a network teaches robustness to visual input, disambiguating what the network should consider important in visual processing and what it should not [14].

This is the same as training the agent on a subset of tasks $\mathcal{M} \subset M$ and testing on a different subset $\mathcal{U} \in M \neq \mathcal{M}$, as depicted in figure 2.2. While this method has shown success in transferring from simulation to real life, as well as to other tasks with different renderers, its sample complexity is far too high to be used in complex visual tasks. As shown in figure 2.2, training on one $m \in M$ takes about 3 years of simulation time to converge, while training on $\mathcal{M} \in M$ takes about 100 years. Additionally, each task $m \in M$ uses a different renderer. This forces the person training the agent to write a series of renderers, which adds additional overhead. If any of the renderers is incorrect, the agent will not correctly transfer to the target task. While control of a dexterous hand is a difficult task, the amount of training time and number of renderers necessary to transfer more complex tasks to real life makes domain randomization infeasible in practice.

Other work has reformulated the family of related tasks as an Hidden Parameter MDP (HiP-MDP). They model a distribution of tasks where variation in the

FIGURE 2.1: NVIDIA Lobby depicted in NVIDIA ISAAC Simulator (top) vs Real World (bottom). Image taken from NVIDIA's blog post introducing the ISAAC simulator [6]

.



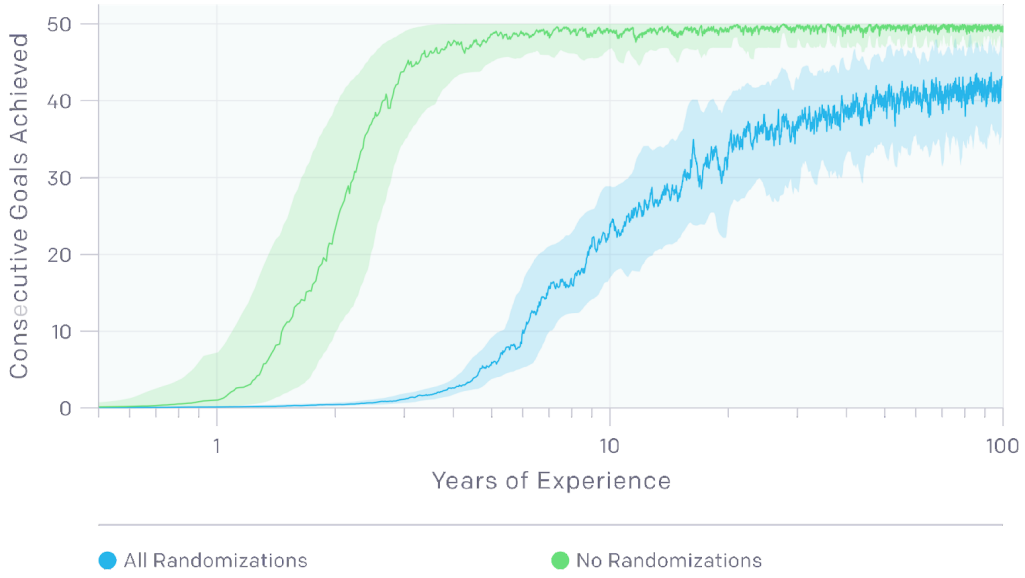


FIGURE 2.2: Domain Randomization for Dexterous Manipulation. OpenAI's Method is above and performance is below (higher is better) [13]. Both the image of the training process and the graph of goals achieved are taken fropm OpenAI's *Learning Dexterous In-Hand Manipulation* [13].

dynamics of the MDPs in the family are captured by a set of hidden parameters θ with probability P_θ [8]. Setting θ_b to some value will result in a task MDP that is analagous to $m \in M$. θ_b is not observed directly by the agent, as it is a set of parameters that specify the task. Transferring across tasks generated by varying θ is the same as transferring across tasks in the family. Note that the transition function T is now modeled as $T(s|a, s', \theta_b)$ where θ_b is a task specific latent parameter from θ [18]. This formulation allows for transfer across a related series of tasks whose transition functions are different. This may enable agents to transfer the knowledge learned from performing a task in simulation to performing the same task in reality. Extending this work may result in a similar hidden parameter that enables the agent to transfer between tasks with different observed states.

Additionally, other newer, neural network inspired methods may allow k-shot transfer. These methods, known as meta learning, are based around optimization of neural network weights for quick learning of a new task [12, 2]. Given a series of tasks F that can each be independently solved by a neural network, these approaches follow these steps.

Initialize some weights ϕ for the neural network
while *Neural Network has not yet converged*
do
  for *Each task* $f \in F$ **do**
    Train for some number of steps on f, storing updated weights in ϕ_f
    Update ϕ based on the gradient from ϕ_f to ϕ using a standard gradient descent optimizer
  end
end

Algorithm 1: General Meta Learning Algorithm [12].

After training to convergence, the network weights ϕ should be optimized to converge to any ϕ_f as quickly as possible given training in task F. Thus, it should also transfer to some held out task $g \notin F$, assuming the optimal network weights ϕ_g for that task are similar to some ϕ_f. This strategy yields transfer between all aspects of an MDP. Not only does it allow a neural network to transfer between tasks with different transition functions, but across MDPs with different states, actions, transitions, rewards, and discount factors.

While this method is very general and potentially very powerful, it is model-free and requires a large amount of training time. Additionally, this strategy will optimize ϕ to converge on ϕ_f as fast as possible when training on task f. However, there is no guarantee how many episodes that convergence will require. If the tasks in F are within the same family M, but their observed states are sufficiently different, meta-learning may have difficulty transferring.

Additional work has been done to attempt to transfer from simulation to reality by directly addressing the *reality gap* instead of modeling a general transfer between tasks. DARLA [5] focuses on 0-shot transfer using a three step method.

1. Learning to see. In this step, the agent learns latent representation for an environment that disentangles the raw image data from the representation using a modified β varational autoencoder [4]. Thus, looking at the same scene across domains should yield similar latent representations, enabling transfer between domains with slightly different visual characteristics.

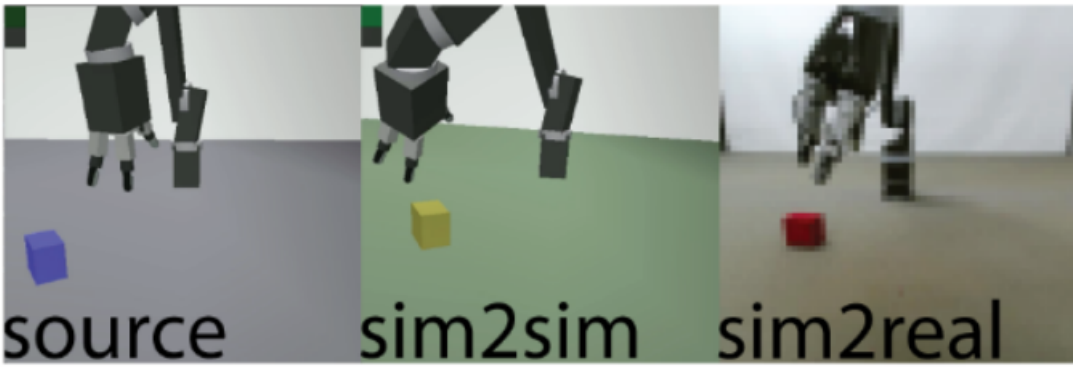


FIGURE 2.3: DARLA: Object Reaching [5]. This image comparison of simulators and reality is taken from DeepMind's *DARLA: Improving Zero-Shot Transfer in Reinforcement Learning* [5].

2. Learning to act. In this step, the agent learns to act based on the latent representation. The agent can use any standard reinforcement learning algorithm, such as PPO [15], actor critic, or other methods. Assuming the latent representation is mostly similar across domains, and the task itself remains the same, a policy learned on the latent representation should transfer across domains with no retraining.

3. Transfer. In this step, the transfer to a new domain is tested, with the same task.

The success of this method relies upon the sight network's ability to map similar states from both the target and source domains to the same state in the latent space. Given this mapping is correct, the action network will receive the same inputs, whether acting in simulation or reality. This method has shown to work for simple domains, such placing the end effector of a Jaco arm close to an object of interest [5]. However, this method's robustness has not yet been thoroughly measured.

Chapter 3

Research Methodology

As mentioned in the previous section, the goal of this thesis is to learn a feature extractor that allows a deep reinforcement learning algorithm to transfer a learned policy between tasks with different visual features. I implemented three different methods of learning the feature extractor. In this section, I describe the research methodology of each approach. Additionally, I describe the visual reinforcement learning task I created to evaluate each method.

3.1 Visual Cartpole Domain

To train and test different methods of transfer across a family M of tasks, I create a Visual Cartpole domain. I modify OpenAI Gym's Cartpole environment.

In this domain, an agent must learn to keep a pole attached to a cart on a track upright for as long as possible. The states in the original domain are the position and velocity of the cart and pole, and the states in the modified domain are image renderings of the cart and pole. The agent acts in both domains by applying a force of $+1$ or -1 to the cart, and the environment transitions based on a physics simulation of the cart and pole. The reward is $+1$ at every timestep that the pole has not fallen over, and the episode terminates when the pole falls over or the cart moves too far to the left or right.

I create a family M of visual cartpoles by varying the colors of the cart, pole, and track, effectively changing the renderer. The goal of this domain is to train an agent on a subset $\mathcal{M} \subset M$ and transfer to tasks that the agent has not been exposed to $\mathcal{U} = M \setminus \mathcal{M}$.

See figure 3.1 for the same state across different tasks in M and figure 3.2 for different states across one task in M.

Note that visual cartpole as described above is a Partially Observable Markov Decision Process (POMDP), in which there is a hidden state that is not directly observable. Specifically, it is not possible for an agent to determine the velocity of the cart and pole solely based on one image.

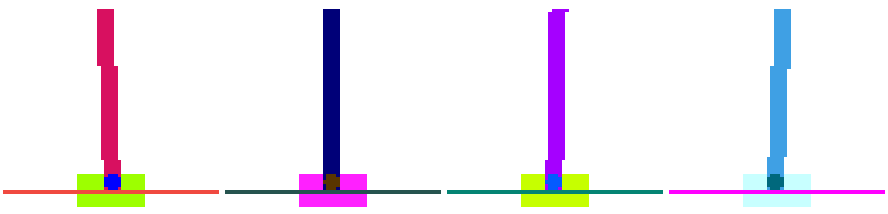


FIGURE 3.1: Fixed State across different tasks $t \in M$.

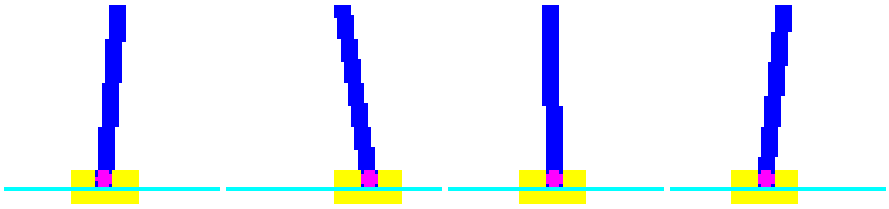


FIGURE 3.2: Varying state within a single task $t \in M$.

In order to simply the visual cartpole task to an MDP, I feed the velocity information directly into the DQNs used to solve the task later in this paper. This allows me to directly test the ability for my feature extractors to extract the position of the cart and pole and the performance of the DQNs utilizing them across visual domains.

To verify the visual cartpole environment and to establish a comparison baseline for the methods evaluated in this paper, I implement, train, and test a convolutional DQN to solve the visual cartpole problem.

The input to the network is the image of the visual cartpole environment and the velocity of the cart and pole. The output of the network is a policy distribution $\pi(a_t|s_t)$ where s_t is the input to the network. The image input is passed through several convolutional layers and flattened. The flattened features are then concatenated with the velocity input, and passed through several dense layers.

This convolutional DQN is able to achieve a maximum reward of 444 on the environment, an average reward of 282.1, and a minimum reward of 253.1. See figure 4.6 for a graph of reward vs. training time.

When the colors of the environment are changed, the convolutional DQN fails to transfer knowledge well, achieving a maximum reward of 129, a mean reward of 71.4, and a minimum reward of 39.6.

3.2 Beta Variational Autoencoder

The first approach I tried is heavily based off of DARLA [5]. There is one key difference. The latent space that is generated by DARLA's beta variational autoencoder is partially disentangled, due to the use of a beta vae.

Given that the observed states, renderings of the cart and pole, come from a visual reinforcement learning task in a family of related tasks M, there are properties that we use to further disentangle the data. In the latent space, there are two types factors that can vary. There are factors that correspond to elements of the S^T that vary within a single episode and across episodes, and factors that vary across episodes, but not within a single episode. I define these as state parameters S, and render parameters RP. For visual reference, see figure 3.1 for an example of varying render parameters and figure 3.2 for an example of varying state parameters.

To achieve this goal, I split the latent space into two sections: one representing the render parameters and the other representing the state. As shown in figure 3.3, I split the encoder into two different encoders, *RPencoder* and *Sencoder* to ease the training process. Additionally, this enables the *Sencoder* to be frozen and used as a feature extractor for the reinforcement learning algorithm without dependence on the *RPencoder*.

Forcing the variational autoencoder to learn to disentangle S from RP in its latent space requires a change to the loss term and the training procedure.

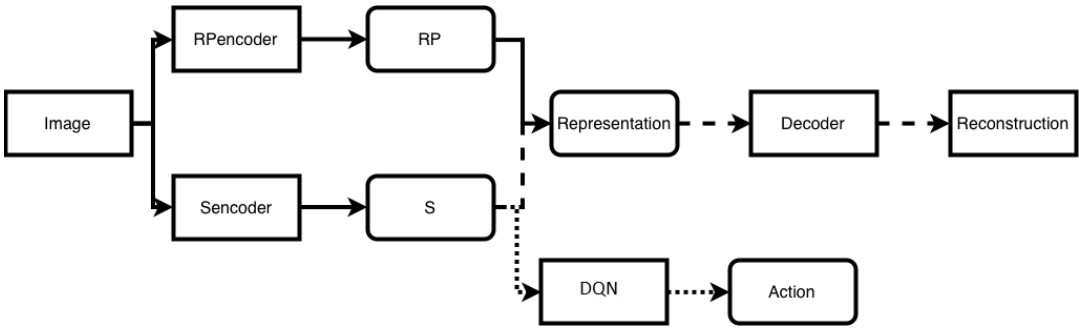


FIGURE 3.3: Beta Variational Autoencoder with RPencoder and Sencoder

3.2.1 Loss Function

For this two encoder architecture, I modify the loss term as follows.

$$\begin{aligned}\mathcal{L} =& \mathbb{E}_{q(z|x)}[\log p_\theta(x|z)] - \beta(D_{KL}(q_\phi(z|x)||(p(z) - \epsilon) \\ &+ 1_{\{x \in Sdataset\}} * L2(z_{0:RPdimension}) + 1_{\{x \in RPdataset\}} * L2(z_{RPdimension:end})\end{aligned}$$

where $L2$ is the $L2(x)$ L2 norm between the vectors in x. The indicator terms toggle the $L2$ losses based on whether or not the image x comes from the RP dataset or the S dataset, as described in the section below. The subscript of the z term denotes which section of the sampled latent vector z the $L2$ norm is applied over.

3.2.2 Training Procedure

Training this two-encoder beta variational autoencoder requires a different procedure than training the beta variational autoencoder used in DARLA. Specifically, in order to train the two different encoders, different training data sets were required. The first set, *RPdatset* contained fixed true state and varying render parameters (colors). The second set, *Sdataset* contained fixed render parameters (colors) and varying true states, as shown in figures 3.1 and 3.2, respectively.

When training the encoder, created batches of some data from each dataset, with an indicator variable. The network's loss function then used this indicator variable to decide if $1_{\{z \in Sdataset\}}$ or $1_{\{z \in RPdataset\}}$ and calculate the loss accordingly.

3.3 Stylegan

3.3.1 Motivation

The previous approach, with a two-encoder beta variational autoencoder network suffered from a lack of generalizability due to the inability of the decoder to learn a generative function over the entire S and RP space given the data it was provided. Additionally, a small number of samples from multiple places in the RP and S space is insufficient to train such an encoder.

Stylegan is a recently released GAN architecture with impressive disentanglement results [7]. There are a few factors that make Stylegan a desirable generative model for this transfer learning task.

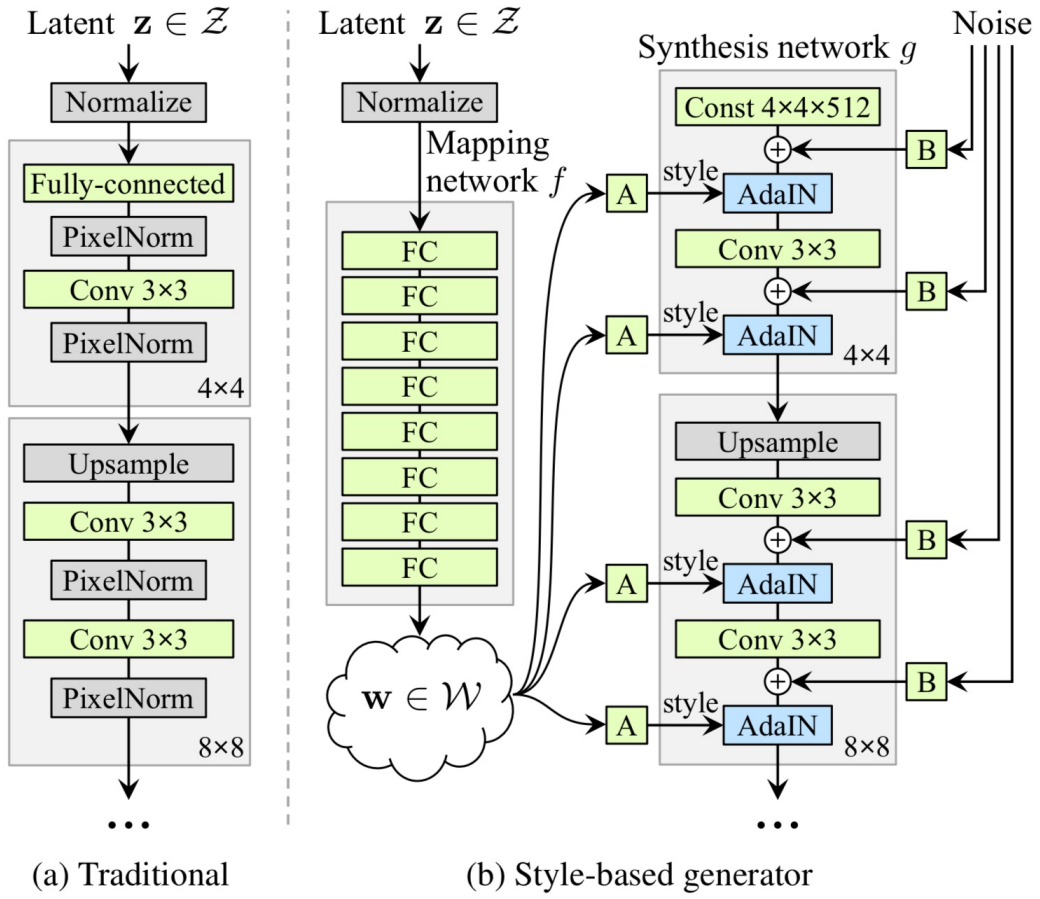


FIGURE 3.4: Stylegan Architechture. Image is from NVIDIA's *A Style-Based Generator Architecture for Generative Adversarial Networks* [7]

.

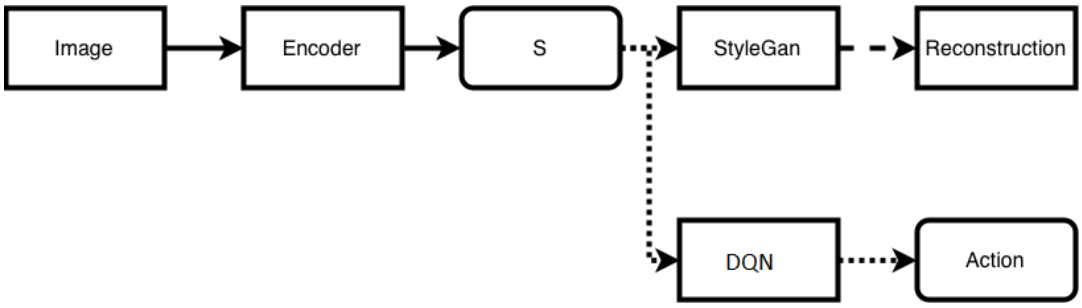


FIGURE 3.5: One Encoder Stylegan Architecture

As depicted in figure 3.4, Stylegan differs from a traditional GAN in one major way. Instead of mapping directly from the latent space Z to images, Stylegan first maps Z to W, an intermediate latent space. W contains the same information as Z, but is a disentangled space. Stylegan achieves this with a mapping network that takes as input $z \in Z$ and outputs $w \in W$. The generator, called a synthesis network, then uses a novel layer $AdaIN$ to apply the "style" learned by each dimension of w to the image at various points throughout the synthesis network [7]. For a visual comparison between a vanilla GAN and Stylegan, refer to figure 3.4.

3.3.2 Methods

In this experiment, I replaced the decoder of the beta variational autoencoder utilized in DARLA with the synthesis network learned in the Stylegan. Theoretically, this generator should learn different styles of the cartpole images with few training examples, and the encoder should be able to regress to the latent space W easily. This network architecture is depicted in figure 3.5.

In this architecture, the encoder takes an image I as input, outputs the corresponding $w \in W$, the disentangled space of Stylegan, and the synthesis network of Stylegan maps the w to a reconstructed image $\hat{I}$. When used as a feature extractor for the reinforcement learning task, the encoder is frozen and extracts a state S, which is the same $w \in W$ that would allow the synthesis network to reconstruct $\hat{I}$. As in DARLA, that state S is then passed as input to the deep reinforcement learning network, which in turn learns to solve the task.

Note that each dimension of W corresponds to a style that the generator utilizes. In the domain of visual cartpole, styles can be factors such as cart position, pole angle, or pole color.

I also tried a variant of this architecture that mirrors the two encoder approach in the previous section. This architecture is depicted in figure 3.6. Instead of one encoder that learns a mapping from I to w, there are two encoders, $RPencoder$ and $Sencoder$ that should learn the render parameters and the state of the image, respectively. As in the previous section, I train these on the same datasets, $RPdataset$ and $Sdataset$ and enforce the same $L2$ loss functions. This should force the output of the $RPencoder$ to stay fixed for fixed render parameters and the output of the $Sencoder$ to stay fixed for fixed state.

3.4 Training Procedure

I trained the one encoder network as described below.

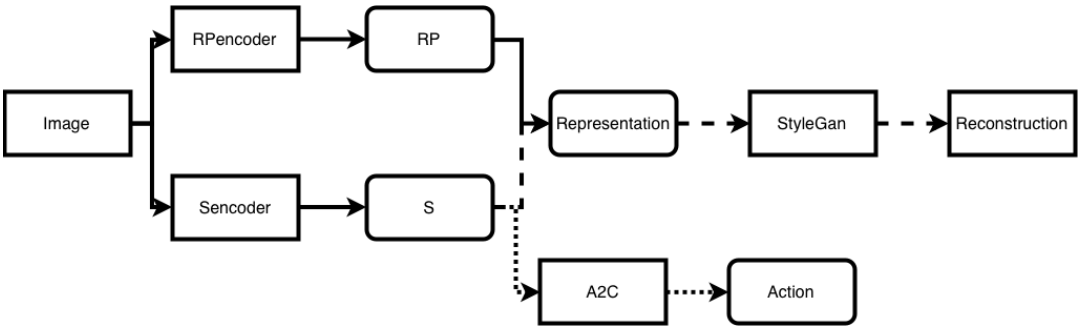


FIGURE 3.6: Two Encoder Stylegan Architecture

1. Generate training data sampled randomly by varying both RP and S.
2. Train the Stylegan as described in NVIDIA's paper on the dataset generated in step 1.
3. Extract and freeze the synthesis network.
4. Train the encoder by minimizing the reconstruction loss between the input image I and the reconstructed image $\hat{I}$ that is created by passing the output of the encoder to the frozen synthesis network.

This training process allowed the encoder to successfully learn a function from I to w.

I trained the two encoder network as described below.

1. Follow steps 1-3 from the one encoder training process.
2. Until the encoders converge:
   (a) Generate a batch of data sampled from $RPdataset$ and $Sdataset$
   (b) Train the encoders on the batch, as described in step 4 of the one encoder training process, but also enforce an $L2$ loss on the extracted RP across all data sampled from $Sdataset$ and an $L2$ loss on the extracted S across all data sampled from $RPdataset$.

3.5 Temporal Autoencoder

There are two main ways to improve the Stylegan DQN results. The first is to learn a separation between RP and S, and the second is to reduce the variance in the behavior of the agent.

In this method, I exploit the fact that data in transfer learning between RL tasks comes from episodes of a reinforcement learning task. I use this knowledge and a novel loss function to train a single encoder and decoder to extract both RP and S from a series of given images. I call this a temporal autoencoder.

The encoder of the temporal autoencoder takes as input an image or series of images corresponding to sequential frames from the reinforcement learning task, in this case, visual cartpole. It outputs a set of predicted render parameters RP and states S for each input frame.

The decoder of the autoencoder takes a series of RP and S, corresponding to render parameters and states extracted by the encoder from a sequential series of frames from the RL task. It then maps them to reconstructed images for each frame.

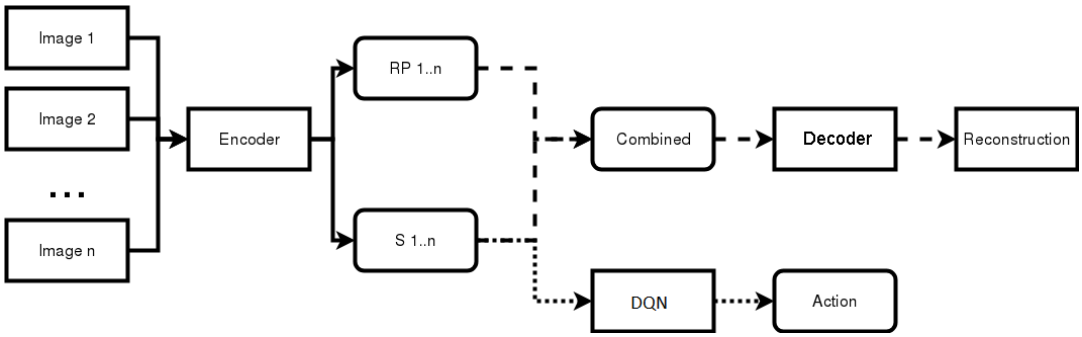


FIGURE 3.7: Temporal Variational Autoencoder Architecture

3.5.1 Loss Function

The loss function for the temporal autoencoder is as follows.

$$\begin{aligned}\mathcal{L} =& \mathbb{E}_{q(z|x)}[\log p_\theta(x|z)] - \beta(D_{KL}(q_\phi(z|x)||(p(z) - \epsilon) \\ &+ L2(z_{0:RPdimension}) - L2(z_{RPdimension:end}) + Entropy(z_{RPdimension:end})\end{aligned}$$

where $Entropy(x)$ is the entropy of the normalized x.

The first term of this function is reconstruction loss, and the second is KL loss, as are standard for a beta variational autoencoder. The third term encourages the render parameters $z_{0:RPdimension}$ to stay fixed over a given series of inputs. The fourth term encourages the extracted state $z_{RPdimension:end}$ to have high variance between subsequent states. Finally the fifth term encourages the states to have a low number of parameters that vary between subsequent frames.

Given that the input to this network is a sequence of images corresponding to one episode of visual cartpole, the true RP should be fixed for the entire episode. Thus, the $L2$ loss of subsequent extracted RP should be 0. The true S is the position of the cart and pole, which change between every frame. Thus, the $L2$ loss of subsequent extracted states should be rather high. However, the true S is a vector of size 2. Given that the extracted state from the temporal autoencoder has length 24, the network may learn to place both the RP and the S in the output vector that should contain only the extracted state. The entropy loss of the extracted state encourages the extracted state to contain as little information as possible, encouraging the extracted render parameters to contain render parameters. Combined with the fact that the extracted render parameters should not change between subsequent frames, and the extracted state should vary highly, this encourages the extracted state to only contain the state.

3.5.2 Training Procedure

Training this autoencoder is much more straightforward than the previous two approaches. I follow the below steps:

1. Create a dataset of images of episodes of visual cartpole based on a random policy
2. For each episode of training data:
   (a) Train the autoencoder on the episode of training data.

This training process is much simpler than the other two, requiring only a single dataset and a single network to optimize.

Chapter 4

Results

In this section, I list the training and test results for each approach. Additionally, I compare their results against the results of a baseline convolutional DQN as well as an implementation of the variational autoencoder-based approach described in DARLA [5].

4.1 Beta Variational Autoencoder

Despite significant hyperparameter turning, the two-encoder network was unable to generalize to unseen data. The network would learn to sufficiently reconstruct training data, yielding a low loss. However, when faced with test data, the network would output one of two reconstructions, depicted in figure 4.1:

1. An image of a cartpole with the same state as the input image, but incorrect colors.

2. An image of a cartpole with the correct render parameters (colors) as the input image, but an incorrect state. Specifically, the state was a starting state.

Upon closer inspection of the reconstructions, it becomes apparent that the network is learning a switch statement, reconstructing whichever option presents lower loss, but not actually learning the entire space. This behavior indicates that the autoencoder is simply memorizing the training examples instead of learning a generative function over the space. Despite changing the network architecture and various other hyperparameters, the network either continued learning the switch statement described above or failed to converge. This indicates an issue in the problem setup.

As shown in figure 4.2, this task gives the ovals as training data, and expects the decoder of the autoencoder to learn a generative function over the entire space. Based on these results, it seems the decoder of an autoencoder is not suited to learn this function.

It would be possible to train the decoder independently, in a normal beta variational autoencoder with data sampled from the entire RP and S space, freeze the weights, and then attach the two encoders and train them as described above.

I explored this approach briefly, but it seems that the amount of data required to train the decoder and then the two encoders separately is too high, since it requires a fairly large dataset where both the RP and the S vary in addition to the *RPdataset* and *Sdataset* as described above. Given the amount of data required to generate these three datasets, it seems almost as easy to train with domain randomization. Additionally, there are generative models that can learn the same function with much less data.

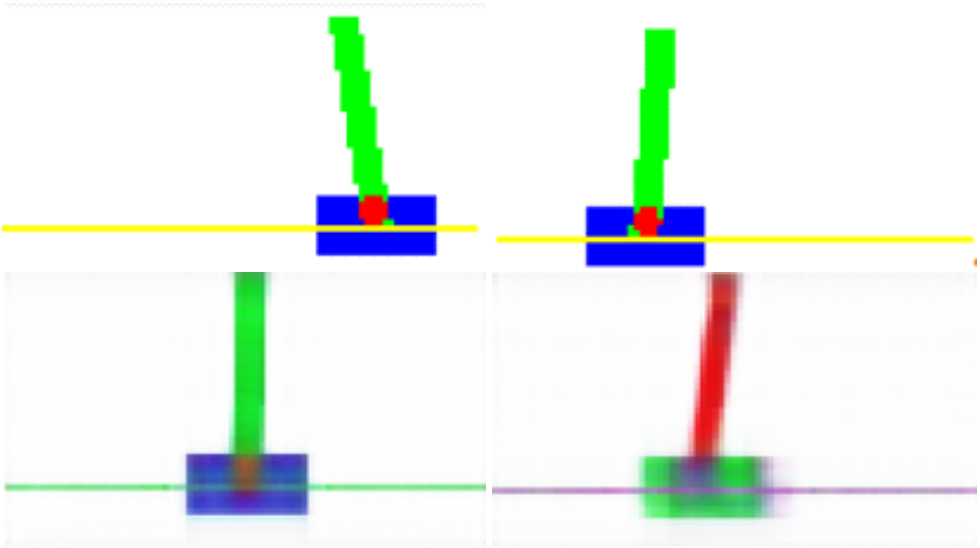


FIGURE 4.1: Two-Encoder Beta VAE Reconstructions. Top row contains input images and bottom row contains reconstructed images. Left utilizes correct RP and right utilizes correct S

4.2 Stylegan

Unfortunately, the two encoder network did not sufficiently learn to separate the RP from the S. Instead, it learned to properly regress from the input image I to the corresponding $w \in W$, but learned a similar piecewise function to the one described in the two encoder beta variational autoencoder, as shown in figure 4.3. The $L2$ loss terms forced the $RPencoder$ to learn both RP and S corresponding to images from $RPdataset$ and $Sencoder$ to learn both RP and S corresponding to images from $Sdataset$. After experimenting with various encoder architectures, weight terms on the different elements of the encoder loss function, and hyperparameters such as learning rate and learning rate decay, it seems that the two encoder is not able to separately learn RP and S.

This is surprising due to the fact that RP and S are relatively disentangled in Stylegan and they are represented as different styles. Since neither the two encoder beta variational autoencoder nor the two encoder Stylegan could seperately learn RP and S, it is likely that the combination of the two encoder architecture and the $L2$ loss function will not learn a disentanglement of RP and S, regardless of generator.

However, the one encoder network did successfully learn to map from an input image I to a state $S = w \in W$ that allows the synthesis network to sufficiently reconstruct the input image, as shown in figure 4.4. As such, since the W space of a Stylegan should contain disentangled styles, I trained a DQN on visual cartpole, utilizing the one encoder as a frozen feature extractor. The reward is plotted as a function of training time in figure 4.6, and a comparison to a baseline DQN as well as a DQN trained on a DARLA beta variational autoencoder is depicted in table 4.1. This table was generated by training each encoder and DQN and testing on both the source RP and a target RP over 10 trials.

Unsurprisingly, the baseline DQN outperforms both the DARLA DQN and the Stylegan DQN on the source task. This is expected, since the baseline DQN is able to optimize its CNN-based feature detection for the source task. It only needs to learn to detect a cart and pole of a single color. Not only is this a strictly easier task than

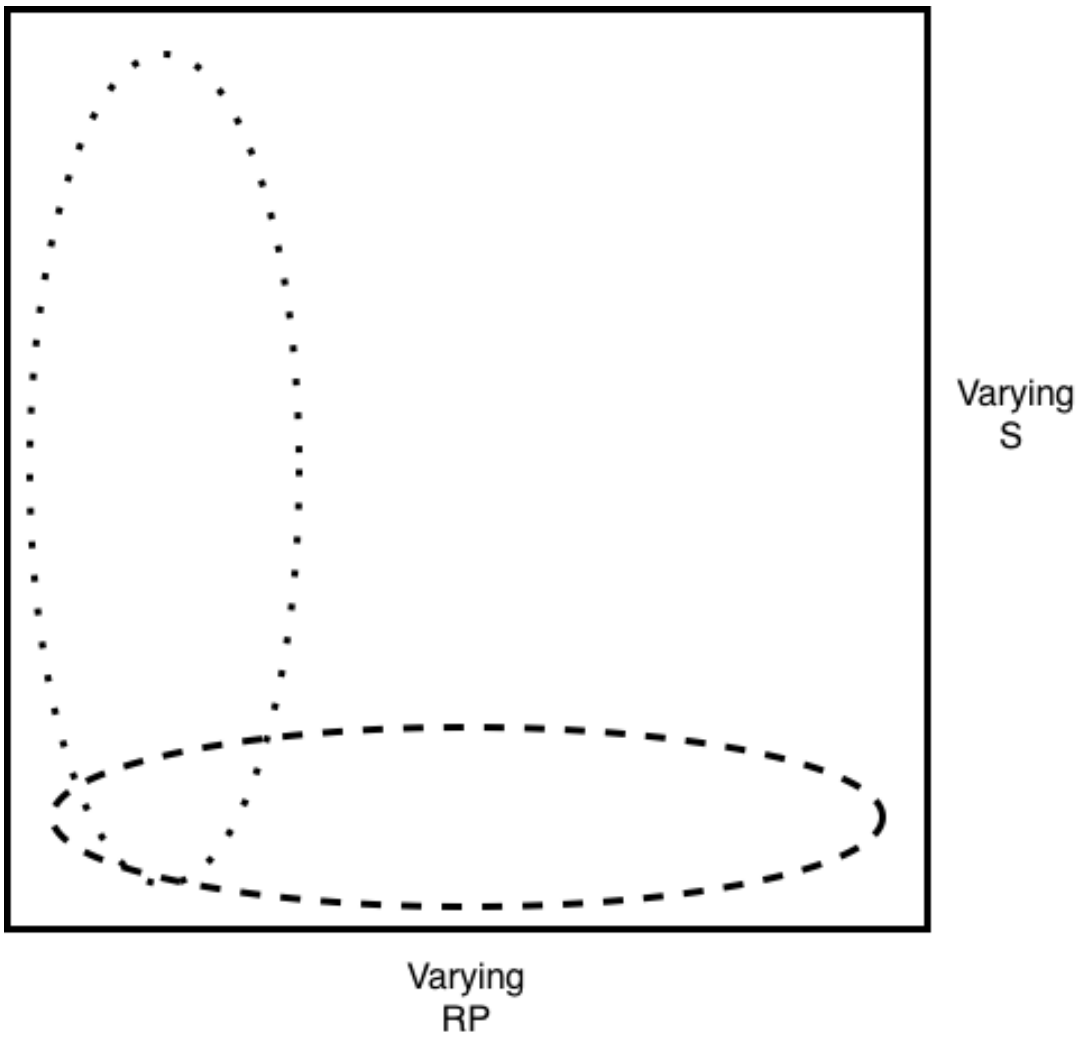


FIGURE 4.2: RPdataset and Sdataset compared to the entire RP and S space

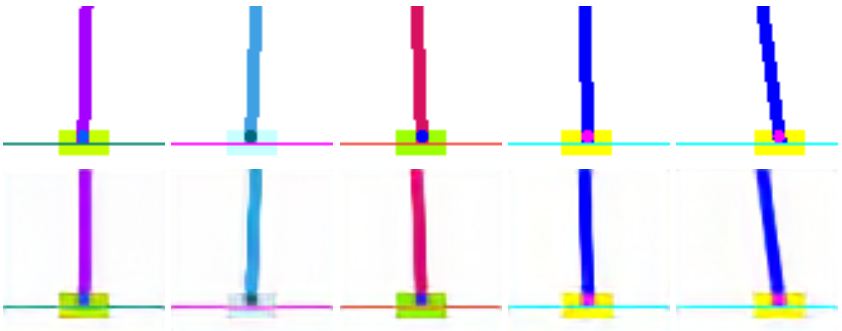


FIGURE 4.3: Two-Encoder Stylegan Reconstructions. Top row contains input images and bottom row contains reconstructed images.

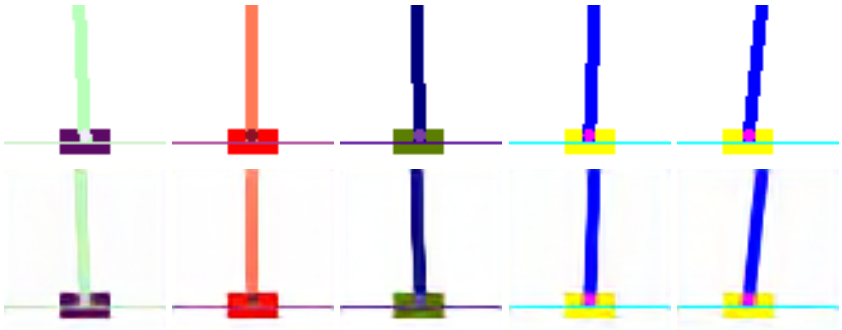


FIGURE 4.4: One-Encoder Stylegan Reconstructions. Top row contains input images and bottom row contains reconstructed images.

learning to detect a cart and pole of varying color, but the baseline DQN is also less modeled than the other two. Specifically, it does not need to learn factors that would aid in image reconstruction but not in actually solving the source task.

Also unsurprisingly, the baseline DQN performs poorly on the target task, balancing the cartpole with different colors (RP). It is able to somewhat generalize due to the simplicity of the task, but this would not scale on more complex visual tasks. The DARLA DQN comes in second and is able to transfer to the target task significantly better than the baseline DQN. This shows that it is learning some representation of the S that is agnostic to changes in RP, but given its drop in performance compared to the source task, it is still not ideal. The Stylegan DQN transfers to the target task the best, either maintaining or improving its performance. This seems to handily beat the other two methods.

The fact that the Stylegan's max performance on the target task is higher than its performance on the source task is surprising. While the Stylegan is able to extract features relatively well, its performance also varies more than the other two methods across test episodes. While it clearly extracts relevant features for transfer, its representation may be difficult for the DQN to fully understand, given the same training time. This is likely due to the fact that the Stylegan's $S = w \in W$ is a vector of length 512, instead of 16, 32, or 48 in the case of the DARLA DQN. This adds significant complexity to the DQN network. I expect that training the Stylegan DQN for longer would decrease variance in performance on visual cartpole. Additionally, changing the latent space size of the Stylegan may decrease complexity.

4.3 Temporal Autoencoder

The temporal autoencoder trains relatively easily, and the corresponding DQN performs surprisingly well on both the source and target tasks.

Surprisingly, the temporal autoencoder's reconstructions are not as accurate as other approaches. As shown in figure 4.5, it is correctly able to learn the state S of the cartpole, but the render parameters RP are not correct. The learned RP are within some bound of the true colors. Likely, this can be improved with additional training time, but does not seem to be necessary to solve the visual cartpole task.

Based on table 4.1, the temporal autoencoder DQN outperforms all the previous approaches both on the source and target tasks. Performing so well on the target task given the same training time as the other networks shows that the latent representation S learned for the state is significantly easier for the DQN to understand.

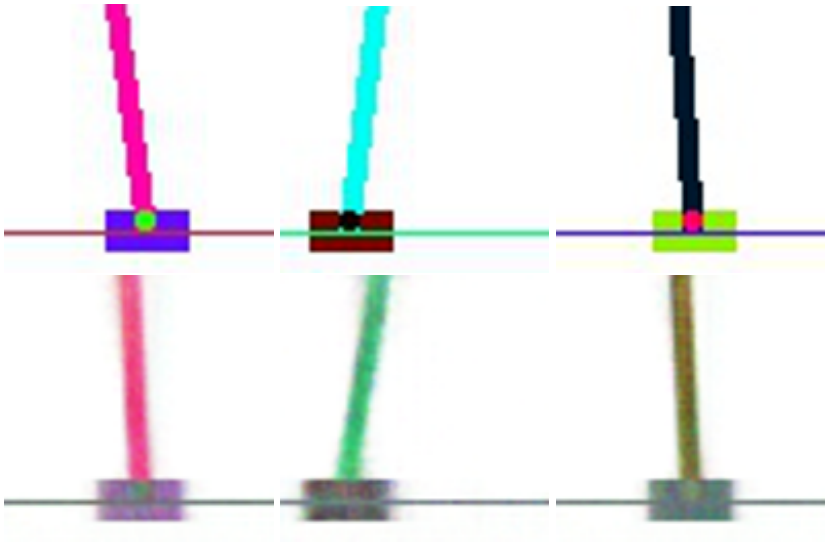


FIGURE 4.5: Temporal Autoencoder Reconstructions. Top row contains input images and bottom row contains reconstructed images.

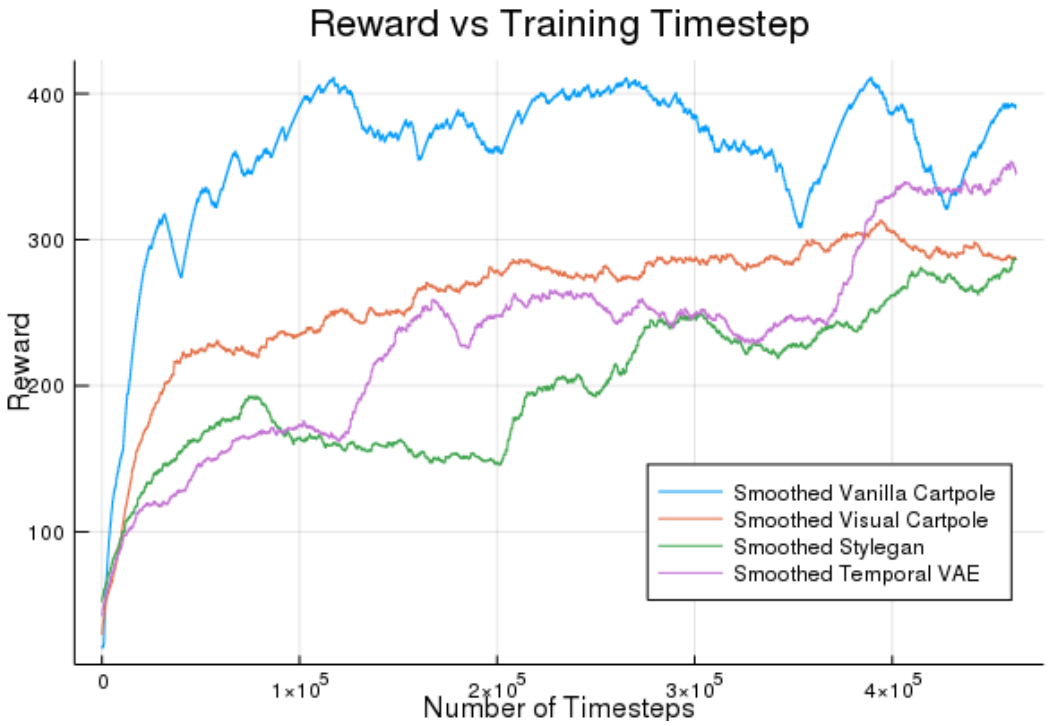


FIGURE 4.6: Episode Reward (Smoothed) vs Training Time across all approaches.

Metric	Visual DQN	DARLA DQN	Stylegan DQN	Temporal Autoencoder
Min Source Task Reward	239	201	180	249
Mean Source Task Reward	282.1	261	262.9	*403.5*
Max Source Task Reward	444	359	408	*500*
Min Target Task Reward	55	148	200	233
Mean Target Task Reward	71.4	189.4	256.5	*335.8*
Max Target Task Reward	129	248	489	*500*

TABLE 4.1: Minimum, Mean, and Maximum Reward when tested on the source and target tasks after training on the source task.

This suggests that the S learned contains all of the information the DQN requires to solve cartpole while having little irrelevant information that the DQN would need to learn to ignore.

The temporal autoencoder DQN's performance on the target task seems to be that its performance on the source task, which shows that it has not entirely disentangled the S and RP. If it had, the performance would be the same across tasks. However, these results show that it has learned to disentangle S and RP better than any of the other approaches, since its maximum reward is the same, and its mean reward decreases only marginally between tasks.

Chapter 5

Conclusion and Future Work

5.1 Future Work

I divide my future work into two section, simulation based work and robot based work.

5.1.1 Simulation Based Work

The approaches described in this paper transfer between visual cartpoles of different colors, however, the feature extractors must only learn to extract the position of the cart and the pole at any given step. Ground Truth velocity information of the cart and pole are directly passed into the DQN in order to simplify the DQN's task and enforce the Markov Property on states. Namely, the next state (image and velocity) of the environment depends only on the current state (image and velocity).

I intend to improve the temporal autoencoder so that it is able to extract the velocity of the cart and pole between subsequent images, in order to extract the entire ground truth state (position and velocity) from images. I will first try to utilize a fixed history of images to learn to predict velocity and will then implement a recurrent neural network to learn to predict velocity. The recurrent neural network should theoretically outperform the fixed history, since it will be able to remember information from images beyond the fixed history window.

This improvement will allow the DQN to solve an MDP without being given any ground truth state information, which is realistic to real world tasks. The improved extractor will allow the DQN to keep track of the velocity or acceleration of visual objects in addition to position.

5.1.2 Robot Based Work

This work has demonstrated that it is possible to transfer learned policies between visual domains in simulation, however, it has not been tested on transfer from a simulated task to a real world task. As such, my first robot based test is to create a simulator for a Markov, visual robotics task, such as grasping a cube, given a robot arm that takes positions as input and a camera that observes both the arm and cube. I will then train the visual feature extractor and the corresponding DQN and test transfer both between simulator parameters and to real life.

Given that my Stylegan and Temporal Autoencoder feature extractors outperform DARLA's feature extractor in visual cartpole, I am hopeful that they will outperform DARA's feature extractor in transfer between simulation and reality. If it does not, it will demonstrate that my feature extractors are not sufficient for complex visual tasks.

5.2 Conclusion

In this paper, I have presented three methods to extract features for visual reinforcement learning tasks that enable transfer between visual domains. I evaluate these three methods on a toy domain of visual cartpole, and compare against previous work as well as a baseline DQN.

This work brings transfer of deep reinforcement learning policies from simulation to reality closer to fruition. Visual differences between simulators and reality are one of the largest factors that contribute to the reality gap, and overcoming differences in rendering between simulations and the real world will enable complex visual policies learned in simulation to transfer to real world robots. My results show that learning to transfer between different visual domains, such as simulation and reality, is within reach, and that jumping the reality gap will soon be possible.

Transferring learned policies from simulation to reality will greatly improve autonomous robots' abilities to aid humans. Humans will not need to explicitly program robots to follow very rigid, specific instructions, such as working in assembly lines. Instead, these robots will be able to teach themselves, in simulation, to solve tasks that humans assign, immensely reducing the overhead of integrating autonomous robots into both industrial settings and daily life.

Bibliography

[1] Carl Doersch. "Tutorial on variational autoencoders". In: *arXiv preprint arXiv:1606.05908* (2016).

[2] Chelsea Finn, Pieter Abbeel, and Sergey Levine. "Model-Agnostic Meta-Learning for Fast Adaptation of Deep Networks". In: *CoRR* abs/1703.03400 (2017). arXiv: 1703.03400. URL: http://arxiv.org/abs/1703.03400.

[3] Ian Goodfellow et al. "Generative adversarial nets". In: *Advances in neural information processing systems*. 2014, pp. 2672–2680.

[4] Irina Higgins et al. "beta-vae: Learning basic visual concepts with a constrained variational framework". In: (2016).

[5] Irina Higgins et al. "DARLA: Improving Zero-Shot Transfer in Reinforcement Learning". In: *CoRR* abs/1707.08475 (2017). arXiv: 1707.08475. URL: http://arxiv.org/abs/1707.08475.

[6] *Introducing NVIDIA Isaac*. URL: https://www.nvidia.com/en-us/deep-learning-ai/industries/robotics/.

[7] Tero Karras, Samuli Laine, and Timo Aila. "A Style-Based Generator Architecture for Generative Adversarial Networks". In: *CoRR* abs/1812.04948 (2018). arXiv: 1812.04948. URL: http://arxiv.org/abs/1812.04948.

[8] George Konidaris and Finale Doshi-Velez. "Hidden parameter Markov decision processes: an emerging paradigm for modeling families of related tasks". In: *the AAAI Fall Symposium on Knowledge, Skill, and Behavior Transfer in Autonomous Robots*. 2014.

[9] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. "ImageNet Classification with Deep Convolutional Neural Networks". In: *Advances in Neural Information Processing Systems 25*. Ed. by F. Pereira et al. Curran Associates, Inc., 2012, pp. 1097–1105. URL: http://papers.nips.cc/paper/4824-imagenet-classification-with-deep-convolutional-neural-networks.pdf.

[10] Jonathan Long, Evan Shelhamer, and Trevor Darrell. "Fully Convolutional Networks for Semantic Segmentation". In: *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. 2015.

[11] Volodymyr Mnih et al. "Human-level control through deep reinforcement learning". In: *Nature* 518.7540 (2015), p. 529.

[12] Alex Nichol, Joshua Achiam, and John Schulman. "On First-Order Meta-Learning Algorithms". In: *CoRR* abs/1803.02999 (2018). arXiv: 1803.02999. URL: http://arxiv.org/abs/1803.02999.

[13] OpenAI et al. "Learning Dexterous In-Hand Manipulation". In: *ArXiv e-prints* (Aug. 2018). arXiv: 1808.00177.

[14] Fereshteh Sadeghi and Sergey Levine. "CAD2RL: Real single-image flight without a single real image". In: *arXiv preprint arXiv:1611.04201* (2016).

[15] John Schulman et al. “Proximal Policy Optimization Algorithms”. In: *CoRR* abs/1707.06347 (2017). arXiv: 1707.06347. URL: http://arxiv.org/abs/1707.06347.

[16] David Silver and Demis Hassabis. “AlphaGo: Mastering the ancient game of Go with Machine Learning”. In: *Research Blog* (2016).

[17] Matthew E Taylor and Peter Stone. “Transfer learning for reinforcement learning domains: A survey”. In: *Journal of Machine Learning Research* 10.Jul (2009), pp. 1633–1685.

[18] Jiayu Yao et al. “Direct Policy Transfer via Hidden Parameter Markov Decision Processes”. In: (2018).