UNSUPERVISED FUTURE FRAME PREDICTION OF COMPLEX SYSTEMS

Justin Tienken-Harder
New College of Florida
Sarasota, Fl 34243
justin.tienken-har13@ncf.edu

Rosa Gradilla
New College of Florida
Sarasota, Fl 34243
rosa.gradilla19@ncf.edu

Nate Wagner
New College of Florida
Sarasota, Fl 34243
nathaniel.wagner19@ncf.edu

Amanda Bucklin
New College of Florida
Sarasota, Fl 34243
amanda.bucklin19@ncf.edu

Dominic Ventura
New College of Florida
Sarasota, Fl 34243
dominic.ventura19@ncf.edu

December 15, 2020

ABSTRACT

Given a video of a dynamical system such as a swinging pendulum, an oscillating spring-mass, or biological system, we are interested in learning the dynamics of the system without incorporating prior knowledge into the model. Ideally, we are interested in either extracting physical laws or predicting the changes in the system over time using deep neural networks. Once trained, our model must be able to predict the next state of the dynamical system given the current state or the past few states without incorporating prior knowledge. For instance given positions of a pendulum at time t=1 sec then what will be the position at times t=2 sec, t=3 sec, etc. We require that our neural network respects the underlying physics of our system such as conservation of energy.

1 Introduction

Image prediction is to generate new samples of potential future images, given one frame at time t = 0, that looks and evolves in a realistic manner. This study has a large amount of potential applications, such as predicting the future motion of a physical system both simple and complex. Our most promising solution uses a regularized autoencoder to first learn a low-dimensional representation (latent space) of the image or video. Autoencoders have shown significant promise in learning smooth representations of complex, high dimensional data such as images. However, a normal autoencoder is inadequate for image reconstruction. In order to obtain more realistic representations of the images, we used an adversarial autoencoder. In the adversarial autoencoder, a discriminator is trained to distinguish samples from a fixed prior distribution and the generator's encoding. The generator is an encoder network. The adversarial autoencoder setup utilizes the discriminator to force the encoder to embed our data into our prior distribution in a smooth and uniform way. Third, our solution uses a reinforcement learning algorithm to utilize a continuous policy network in order to obtain a distribution of the most likely next states from the current state and reward our network when it chooses a similar trajectory to the actual next frame.

2 Background and Related Work

2.1 Hamiltonian Paper Discussion

In recent years, there has been much research (where? who? what did they do?) given to developing models to learn the dynamics of a physical system from unlabeled data. One such simple physical system researched is the simple pendulum. The goal was to define a neural network that would be able to conserve energy-like quantities over time.

Originally, we were tasked to, using a neural network, learn the basic laws of the simple pendulum. First, we created a simple pendulum simulated dataset using the OpenAI Gym package. Using an Autoenoder (AE), we obtained the latent representation of the ideal simple pendulum and fed the representation to a Hamiltonian Neural Network (HNN). The HNN, which attempted to model the system's dynamics, output variables analogous to the position and velocity of the ideal simple pendulum.

2.1.1 The Theory

In essence, physics is all about predicting changes over time. With this in mind, the model should be able to predict the next state given the current one. Since the Hamiltonian Neural Network is based on Hamiltonian Mechanics, we will quickly review some concepts: In Hamiltonian Mechanics, we begin with a set of coordinates for a system. The coordinates in question, for a simple pendulum, are position denoted as q=(q1, q2, ..., qn) and momentum p=(p1,p2,...pn). Note that this gives us N coordinate pairs, this pair offers a complete description of the system and based on known physical laws, it allows us to determine future positions and momenta. We define a scalar function, H(q,p) called the Hamiltonian so that

$$\frac{dq}{dt} = \frac{\delta H}{\delta p} \tag{1}$$

$$\frac{dp}{dt} = \frac{-\delta H}{\delta q} \tag{2}$$

Letting **S** (the action) denote the time derivatives of the coordinates of the system:

$$(q_1, p_1) = (q_0, p_0) + \int_{t_0}^{t_1} S(q,p)dt \tag{3}$$

Moving coordinates in the direction of

$$S_H = (\frac{-\delta H}{\delta p}, -\frac{-\delta H}{\delta q}) \tag{4}$$

gives us the time evolution of the system. We can think of the action **S** as a vector field, in fact it is a symplectic gradient. Moving in the direction of the symplectic gradient keeps the output constant. If we recall the total energy in a system must be conserved:

$$E_{TOT} = H(q,p) \tag{5}$$

2.2 Hamiltonian Neural Networks

The HNN aims to learn a parametric function for the Hamiltonian, i.e., $H_\theta(p,q) \approx H(p,q)$. During the forward pass it consumes a set of coordinates and outputs a single scalar "energy-like" value. Then, before computing the loss, it takes a gradient of the output with respect to the input. We use this to compute and optimize an L2 loss:

$$L_{HNN} = \left|\left|\frac{\delta \mathcal{H}_\theta}{\delta p} - \frac{\delta q}{\delta t}\right|\right|_2 + \left|\left|\frac{\delta \mathcal{H}_\theta}{\delta q} - \frac{\delta p}{\delta t}\right|\right|_2 \tag{6}$$

In the case of an ideal pendulum, writing the gravitational constant as g and the length of the pendulum as l, the general Hamiltonian is:

$$H = 2mgl\big(1 - cos(q)\big) + \frac{l^2 p^2}{2m} \tag{7}$$

We set m =l =1 for simplicity. Then, sample initial coordinates with total energies in the range [1.3,2,3]. We constructed training and test sets of 25 trajectories each and added the same amount of noise.

In order to extract p and q values from an image, an Autoencoder was trained on a flattened representation of the image as input, with a 2 dimensional latent vector, each dimension representing position and momentum. The loss function for the HNN with an Autoencoder consisted of three components: the HNN loss mentioned before, a classic Autoencoder reconstruction loss: L2 and the third component was an extra loss for the Autoencoder based on our knowledge of the relationship between momentum and position:

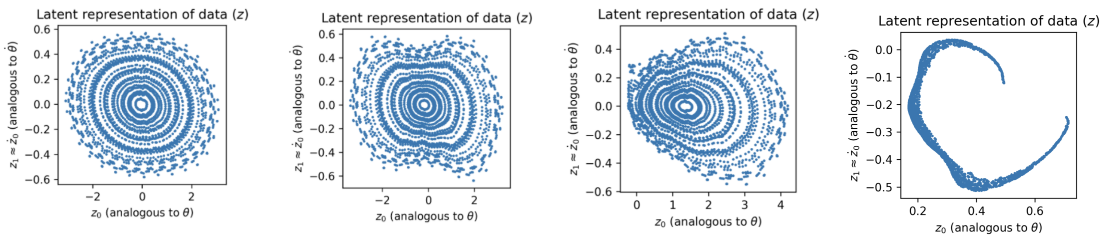
$$L_{CC} = ||p_t - (q_t - q_{t+1})||_2 \tag{8}$$

This encouraged the latent vector to resemble the properties of canonical coordinates (p,q).

2.3 Limitations

The Hamiltonian Neural Network required a couple of very specific features. The first one was the initialization of weights, the HNN requires a special initialization technique that requires orthogonalized weights. PyTorch developed this initialization method torch.nn.init.orthogonal() in accordance with the paper [8] *Exact Solutions to the Nonlinear Dynamics of Learning in Deep Linear Neural Networks*. In this paper, the authors show that random scaled Gaussian initializations can not achieve dynamical isometry despite their norm-preserving nature, while random orthogonal initialization can, thereby achieving depth independent learning times. The second specific requirement was to use residual connections in the autoencoder. This allows for more information to go deeper in the network, making it easier to generalize.

Although we were able to implement the HNN with the AE, we had constraints simply because the network modeled an ideal system. We were interested in different approaches which involved more complex physical dynamical systems and we wanted to come up with a solution that did not require any domain specific knowledge. Without knowing the degrees of freedom of the physical system in the video data, we have no good initial estimates for the size of our latent space, and we might be forced to just do a hyperparameter search. In general, slight modifications to the initialization of the autoencoder can have non-trivial impacts on the shape of the latent space representation which will have an impact on the HNN's performance in approximating the Hamiltonian of the system.



(a) (b) (c) (d)

Figure 1: How changes in auto-encoder initialization effected the shape of the latent space. (a) Includes weight-initialization, residual network, and latent space loss. (b) Removes weight-initialization. (c) Removes residual network for autoencoder only. (d) Removes latent representation loss described in [3].

3 Predicting Movement of Complex Physical Systems

Our goal was to design a model which allowed us to predict states of a complex physical system without the constraints of an ideal system, such as the simple pendulum. When it came to predicting subsequent frames within a video, we did not want uncomplex data which would produce overly simplified methods for generating new video frames directly from past frames. We wanted to work with data which would be challenging and develop techniques that would be transferable to other complex systems.

3.1 Data

For our data, we chose to model the complex physical system of the animals captured on the *CritterVision* Camera on YouTube. The data is from a live-streaming camera which shows local wildlife feeding from multiple seed troughs and water bowls in a backyard in low-country South Carolina. The physical systems most prominent in the data include, but are not limited to, raccoons, possums, deer, and squirrels.

We chose this video data because training data is unlimited and easily accessible due to continuous live-streaming, and constant camera position for consistency in our data evaluation.

3.2 Data Collection

The video specifications is 480p by 360p with a resolution of 30fps. We scraped sixteen hours of night time live-streamed video from YouTube and saved the video to .mp4 files.

Because of the limited space on the server, we saved every consecutive 10th frame from the video which was equivalent to 3 frames per second. We believed by capturing every 10th frame the appropriate movement of the physical systems would be reflected for the training data while still keeping our limited server capacity forefront.

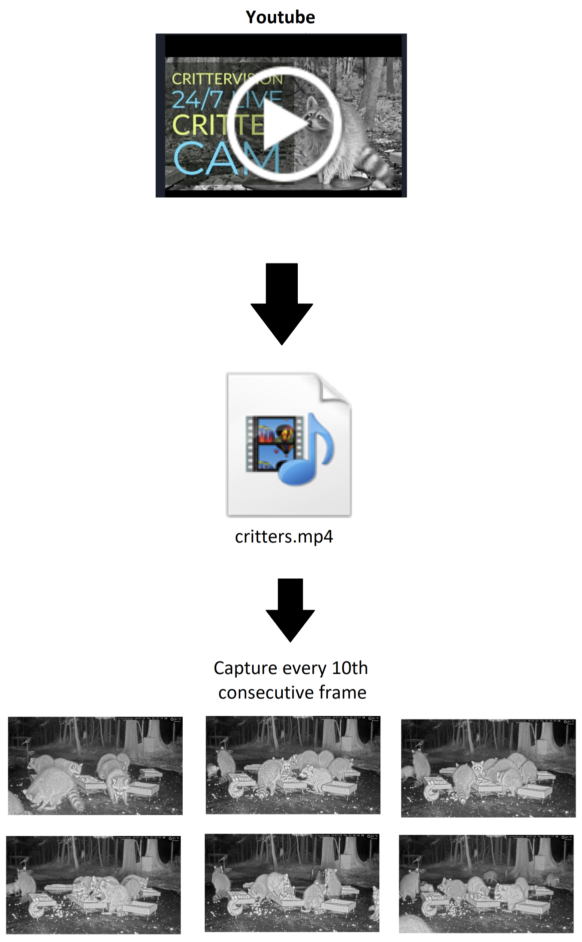

Youtube
CRITTERVISION 24/7 LIVE CRITTER CAM
critters.mp4
Capture every 10th consecutive frame

Figure 2

3.3 Autoencoder

Following data collection and cleaning, we need a latent representation of our data in some n-D dimensional distribution. We initially implemented a Variational Autoencoder (VAE) due to some of its desirable characteristics over a basic Autoencoder (AE). A big issue with the AE is that the latent space it transforms the inputs to and where their encoded vectors lie, is typically not continuous, or allows easy interpolation. Since we plan on making predictions within the latent space, we desire a smooth latent code, evenly distributed over a given prior. Also, gaps in the embedding of the latent space can result in unrealistic images when trying to interpolate between gaps. With the VAE, as opposed to a basic Autoencoder, we will try not to make guesses concerning the distribution that's being followed by latent vectors ($\vec{\mu}$ and $\vec{\sigma}$). We simply tell our network to learn the n-D dimensional independent Gaussian distributions.

Since our model learns some ($\vec{\mu}$ and $\vec{\sigma}$) to represent an image, our goal is to investigate the latent representation. We were anticipating the data closest to the mean vectors of our Gaussian distributions to be the most normal looking frames, such as no-animals static feed. Data in the one-standard deviation sphere could presumably be uncommon frames such as animals eating at the feeders.

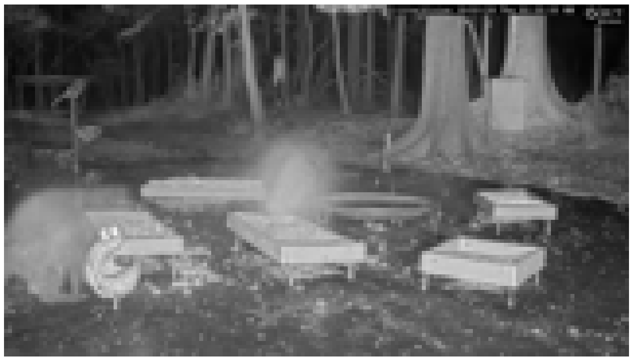


Figure 3: Reconstruction from variational autoencoder.

Finally, data in the two plus standard deviation sphere could likely include animal movement, multiple animals, or even persons topping off the feed. The original 640x360 images were resized down to 160x90 which gave us flattened 14400 dimensional vectors to be used as input to the VAE. Our model then consisted of fully connected layers with relu activation functions. We tried many different n-D dimensional latent space sizes, however, our reconstructed images were consistently inadequate. Figure 3 is an example of a reconstructing from the VAE. Even with increasing the number of latent space dimensions n-D (20, 40, 80, 100), we would still end up with blurry and distorted animals within the reconstructions. To try and enhance our reconstructed images, we also tried replacing the L2 loss on the decoder network with L1 loss. MSE losses on the decoder network can tend to result in blurrier images due to extreme predictions being heavily penalized. However, this still resulted in undesirable blurry reconstructions.

When the reconstructed image from an Autoencoder is blurry, it could be because the encoder output does not cover the entire n-D latent space (i.e. it has a lot of gaps in the output distribution). To fix this, we decided to implement an Adversarial Autoencoder (AAE). In Figure 4, the image (left) shows how a VAE can leave gaps in the output distribution of MNIST. While in the AAE image (right), the sharp transitions indicates that the coding space is filled and exhibits no holes or gaps. So with the AAE, we still force the encoder output to match a given prior distribution, however, this time we also incorporate a generator/discriminator process. We train the discriminator to tell apart real images from our data set with the fake ones generated by our generator. Our generator will initially output some random noise, then after training our discriminator to distinguish between this random noise and the real images, we connect the generator to the discriminator and back-propagate only through the generator with a constraint that the discriminator output should be one. This also encourages the generator to generate more realistic appearing images as training increases, and will increasingly confuse the discriminator, as it gets harder to distinguish between real and fake.

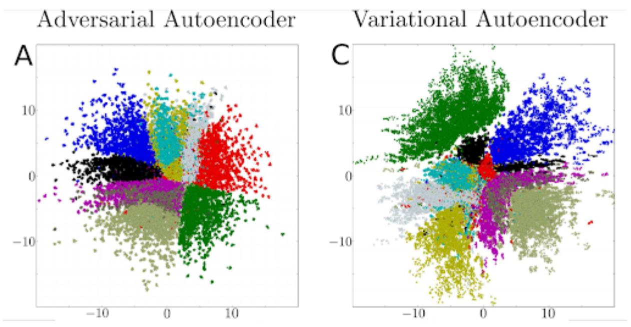


Figure 4: Left: Example latent space from adversarial autoencoder. Right: Example latent space from variational autoencoder. On the MNIST dataset sourced from [4]

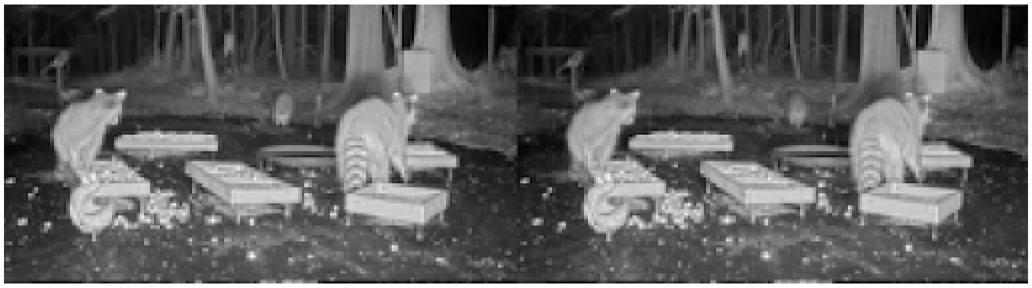


Figure 5: Left: 160x90 input image. Right: Adversarial autoencoder reconstruction from 20 dimensional latent space.

We started by training the AAE on approximately 30k images for 150 epochs. On the GPU this process took about 2 hours to complete. We started with a 40 dimensional latent space, which provided quite adequate results, however, we were able to cut this down to 20 dimensions while still maintaining clear reconstructions. The AAE was successful in outputting much more realistic and sharper reconstruction images. We include a sample test image reconstruction in Figure 5. The image on the left is the resized 160x90 input image, and on the right is the output of the decoder. Since we required the encoder output to match a Gaussian distribution, we expect the latent space variables to follow a sort of bell-shaped distribution. In Figure 6 we plot the distributions of each latent space variable. We ended up with mostly skewed left or skewed right distributions.

Following the training of the AAE, we wanted to see how we could interpolate between frames in the latent space. So we performed linear interpolation between two given encoded frames. In particular, we tested three different interpolations of images 15 frames apart, 70 frames apart and 200 frames apart. We then passed the interpolations through the decoder and stitched them together into MP4 video, which can be seen here: 15-frames-apart, 70-frames-apart, 200-frames-apart. One can notice some significant blur between the interpolations. This might suggest there are still some gaps or unfilled holes in the latent space.

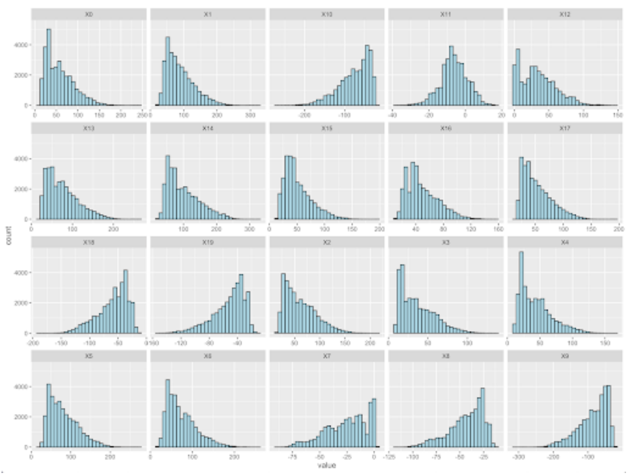


Figure 6: Latent variables distributions.

3.4 Reinforcement Learning

Now that we have a space of plausible video frames, we wish to find trajectories through this space, i.e., how can we get from the current frame to some potential frame some number of timesteps away. The entire video can be viewed as some very long walk through our latent space; moreover, this walk could have crossings – frames that are very similar but separated over a long period of time (e.g., a single raccoon who feeds in the exact same location every hour throughout the night with no other animals in frame). Because of these crossings, the next frame should be thought of as a distribution of plausible next states. Some of this unpredictability of a physical system can be due to the chaotic nature of the system being effected by unobservable subtle forces or simply the relatively unpredictable movements of animals. While individual next frame predictions are good, we're interested in next frame predictions that also lead to better long-term predictions. Maximizing long-term rewards is well suited for a Reinforcement Learning approach utilizing a policy network. Future work should investigate a supervised approach that predicts a distribution of next frames.

The underlying mathematical intuition of the frame prediction problem is sketched below following the idea that a chaotic dynamical system with imperfect information is similar to a random process:

Given some sequence of frames $(x_1, x_2, \ldots, x_n)$ where $x_i \in \mathbb{E}^D$ for some large D (for low quality images, this is as large as 360x480). We wish to find the next frame X_{n+1} which is modeled as a random variable with probability density function $f_{X_{n+1}} : \mathbb{E}^D \to \mathbb{R}$. For simplicity and without loss of generality, consider the case where the sequence of frames is of length 1 – then there is a (hand-waved, but probably) well defined statistical manifold constructed by the surjective function defined by:

$$\pi : E^D \to F$$

$$\pi(x_i) = f_{X_{i+1}}$$

Essentially, every point of in our high dimensional image space can be associated with a probability density function that describes the most probable next frames. There's (probably) a way to derive the probability distribution for the nth next frame from x_i which we expect would look much different than the single next frame distribution. We're interested in learning the mapping π and sampling from $f_{X_{i+1}}$. This formulation has a number of key issues that need to be addressed: Our data is in only a small region of this space; there's no restrictions on the type of distribution at each point; we must presume that points close together in this space result in small changes of the distribution (some notion of continuity); most the points in the space are white noise and shouldn't have a subsequent frame.

Because our data is actually in such a small portion of this space, we don't need to learn the mapping π across the whole domain, just restricted to (at least) the data we have. We accomplish this by learning an invertable mapping of our data into a smaller space (via an encoder/decoder network) $f_{\theta_1} \circ g_{\theta_2} : E^D \to E^d \to E^D$ where $f_{\theta_1}^{-1} \approx g_{\theta_2}$ and learning the very different $\pi' : E^d \to F'$ on this smaller space. It's important to notice that different prior-distributions (in the same dimensionality) for an autoencoder will result in differently "twisted" up walks [2]. Therefore, finding the interpretable directions in our latent space, e.g., "animals walks towards forest" direction, will likely be highly dependent on the starting point. We rely on the policy network to find these directions/step sizes.

3.4.1 Introduction Reinforcement Learning Approach

Reinforcement learning problems are frequently framed in terms of an Agent observing and acting in some Environment. This is largely due to a history with psychology and robotics; however, the common mathematical formulation is based upon Markov Decision Processes and their related structures (policies, value functions, etc). We will give a brief overview.

In reinforcement learning, an agent interacts with its environment, typically assumed to be a Markov decision process (MDP) $(\mathcal{S}, \mathcal{A}, p_M, r, \gamma)$, with state space $\mathcal{S}$, action space $\mathcal{A}$, and transition dynamics $p_M(s'|s, a)$. At each discrete time step, the agent receives a reward $r(s, a, s') \in \mathbb{R}$ for performing action a in state s and arriving at the state s'. Generally, we write $r_t = r(s_t, a, s_{t+1})$. The goal of the agent is to maximize the expectation of the sum of discounted rewards, known as the return $R_t = \sum_{i=t+1}^{\infty} \gamma^i r_i$ which decays future rewards by the discount factor $\gamma \in [0, 1)$. The agent selects actions with respect to a (deterministic) policy $\mu : \mathcal{S} \to \mathcal{A}$, or in our case, a stochastic polict $a \sim \pi(\cdot|s)$. Each policy π has a corresponding value function $V^\pi(s) = \mathbb{E}_\pi[R_t|s = s_t]$, the expected long-term return (ELTR) when following the policy π in state s. There's a related function, the action-value function for a policy $Q^\pi(s, a) = \mathbb{E}[R_t|s = s_t, a = a_t, \pi]$, the expected long term reward under the current policy given a current state and action (essentially, the starting state and first action is already chosen, and the remaining dynamics unfold under the current policy). The difference between $Q^\pi(s, a)$ and $V^\pi(s)$ is that the latter measures the expected long-term reward of a single state while the former measures the expected long-term reward of a state if it performed some action (and followed the policy thereafter). The mathematical existence and uniqueness of optimal policies for MDP's is what makes them so powerful, and the backbone of Reinforcement learning training algorithms.

There's a few important relationship between the on-policy action function and on-policy state value function that should follow from the definitions, and the (well proven elsewhere) Bellman equations which proves the intuition that the value of your starting point is the (immediate) reward you expect from being there plus the value of wherever you land next:

$$V^\pi(s_t) = \mathop{\mathbb{E}}_{a \sim \pi}[Q^\pi(s_t, a)] \tag{9}$$

$$Q^\pi(s_t, a_t) = \mathop{\mathbb{E}}_{s' \sim P_m}\left[r(s_t, a_t, s') + \gamma \mathop{\mathbb{E}}_{a' \sim \pi}[Q^\pi(s', a')]\right] \tag{10}$$

$$Q^\pi(s_t, a_t) = \mathop{\mathbb{E}}_{s' \sim P_m}\left[r(s_t, a_t, s') + \gamma V^\pi(s')\right] \tag{11}$$

Finally, we're frequently interested in finding how much better an action in our current state is than average under our current policy which is encapsulated by the advantage function. The advantage function is so important, I'll describe it another way – given a current state, action, and current policy, the advantage describes the difference between the ELTR of taking the chosen action in our current state (following the current policy thereafter) and the ELTR of the current state (under the current policy):

$$A^\pi(s, a) = Q^\pi(s, a) - V^\pi(s) \tag{12}$$

This concept is very important, as it allows us to quantify the long-term advantages or penalties that are incurred from taking an action in the current state, and it's worth showing an important identity as we actually try to approximate this advantage function with a neural network later. The main concern is that we might need to train two additional networks (in conjunction with the policy we eventually want to learn). But the helpful identity in equation (9) and (10) shows it can be done by just learning the value function under the current policy:

$$A^\pi(s_t, a_t) = \mathop{\mathbb{E}}_{s' \sim P_m}[r(s_t, a_t, s') + \gamma V^\pi(s')] - V^\pi(s_t) \tag{13}$$

$$A^\pi(s_t, a_t) = r(s_t, a_t, s') + \gamma V^\pi(s') - V^\pi(s_t) \tag{14}$$

Where the expectation is dropped under the "bellman backup" which is essentially an approximation to the true bellman identity due to the difficulty in calculating the expected single reward after the state transition (sampling from P_m), particularly in offline-learning cases where you can't directly explore the environment, and only have a history. Nonetheless, we make progress to describing the two training algorithms we utilized and their mathematical basis.

3.4.2 Policy Gradient

Usually, we are interested in finding the optimal policy, $\pi^*(a|s)$, that maximizes the expected long-term reward $V^{\pi^*}(s)$ across all states. We do so by trying to approximate it with a neural network pi_θ and maximizing the objective function through gradient ascent, which for a finite trajectory is:

$$J(\theta) = \mathbb{E}[\sum_{i=0}^{T-1} \gamma^i r_i | \pi_\theta]$$

The weight updates are done by:

$$\theta_{i+1} \longleftarrow \theta_i + \frac{\delta}{\delta\theta} J(\theta)$$

And after some calculus, and thoughtful inclusion of the independence of each state in a MDP the following is a well understood result;

$$\nabla_\theta J(\theta) = \mathbb{E}[\sum_{i=0}^{T-1} \nabla_\theta log(\pi_\theta(a_i|s_i)) G_j] \tag{15}$$

Where $G_j = \sum_{j=i}^{T-1}$ is the "reward to go", and due to some nice properties of expectation, gradients, and logarithms, we can subtract any baseline function, $b : S \to \mathbb{R}$, from G_j and have an equivalent formulation;

$$\nabla_\theta J(\theta) = \mathbb{E}[\sum_{i=0}^{T-1} \nabla_\theta log(\pi_\theta(a_i|s_i)) G_j] = \mathbb{E}[\sum_{i=0}^{T-1} \nabla_\theta log(\pi_\theta(a_i|s_i))(G_j - b(s_i))] \tag{16}$$

3.4.3 REINFORCE

The main problem is actual computation of G_j (which is a constant, and best interpreted as a step size) due to our policy being stochastic. The "classic" way to do this was through the REINFORCE algorithm (our implementation is adapted from [5] and psuedo-code is described later) which estimates G_j through a monte carlo simulation by unfolding some number of trajectories from the same starting state. We can reduce the variance of the weight updates by subtracting a baseline average in our monte carlo simulation. However, it should be clear that the quality of the estimate decreases of G_j as j gets much larger than i because the ELTRs from these states can be strongly correlated to past decisions (and so this method generally has a large variance of the weight updates, even when subtracting out baselines). Moreover, doing this monte carlo rollout is computationally expensive and as a training method is shown to (generally) perform worse for on-policy continuous control problems (MDP's with continuous actions and/or states) [1].

3.4.4 A2C - Advantage Actor Critic

This leads to another approach where we use the on-policy Q-function to determine the step size which is actually equivalent to equation 16 due to some algebra and application of the law of total expectation. Then, utilizing the on-policy value function as our baseline we get these other equivalent formulations of the policy gradient. We're most interested in the advantage formulation;

$$\nabla_\theta J(\theta) = \mathbb{E}[\sum_{i=0}^{T-1} \nabla_\theta log(\pi_\theta(a_i|s_i))Q^\pi(s_i, a_i)] \tag{17}$$

$$= \mathbb{E}[\sum_{i=0}^{T-1} \nabla_\theta log(\pi_\theta(a_i|s_i))\big(Q^\pi(s_i, a_i) - V^\pi(s)\big)] \tag{18}$$

$$= \mathbb{E}[\sum_{i=0}^{T-1} \nabla_\theta log(\pi_\theta(a_i|s_i))A^\pi(s_i, a_i)] \tag{19}$$

$$= \mathbb{E}[\sum_{i=0}^{T-1} \nabla_\theta log(\pi_\theta(a_i|s_i))(r_i + \gamma V^\pi(s_{i+1}) - V^\pi(s_i)] \tag{20}$$

The question remains, how do we access the on-policy value function to train our policy/agent network? Quite simply, we approximate it with another neural network, called the critic, $V_\phi^{\pi_\theta}(s) \approx V^{\pi_\theta}(s)$, and there are quite a number of ways to construct loss functions. There is a great exposition covering other general strategies for advantage function estimation [9]. We'll write our algorithms for both REINFORCE and A2C in the implementations section.

3.5 Connecting Frame Prediction to Reinforcement Learning

As mentioned previously, we are interested in learning the mapping from points in our latent space representation to the "subsequent" frame's distribution. In this section we'll directly identify each aspect of the MDP with portions of our problem: The state space is formed by the vector space defined by the latent space learned by the autoencoder, in particular, $\mathcal{S} = \mathbb{E}^d$. Our action space is either the perturbation to the current latent-space vector $a \in \mathbb{E}^d$ or (equivalently) the actual next frame – the main difference is in the transition dynamics $P_M(s'|s,a)$ is the simple function (i.e. discrete probability distribution) $f(s,a) = s + a$ for perturbations for $f'(s,a) = a$ for next-frame. The most simple reward is $r(s_t, a_t, s_{t+1}) = -||s' - s_{t+1}||_2$ where s' is the next frame in some sequence of frames that we're training on. The discounted reward, γ was chosen to be 0.99.

A notable detail is that, generally, the next frame for some physical process is conditionally dependent on more than a single frame which violates the assumption that the optimal action in a state only depends on the current state. The way around this is to reframe the state space of our "improper" MDP to be the entire trajectory from a starting state as is described in [6]. Essentially, the state space is a whole trajectory from the starting state: $s'_t = (s_0, a_0, s_1, \ldots, s_{t-1}, a_{t-1}, s_t)$. However, the tradeoff is that now our problem is partially observable which has a bit less robust mathematical theory on convergence (unless your policy network could handle variable length inputs).

We have no way to actually start and explore from arbitrary points in our latent space because we only have a collection of walks from our video dataset (don't have data on arbitrary starting positions of animals); in this way, we're doing offline learning. In some sense, we're trying to closely imitate the policy inherent in the data (the aforementioned mapping from points in the latent space to the next-frame distribution) that we have and to generalize to points in the latent space that's never been seen before.

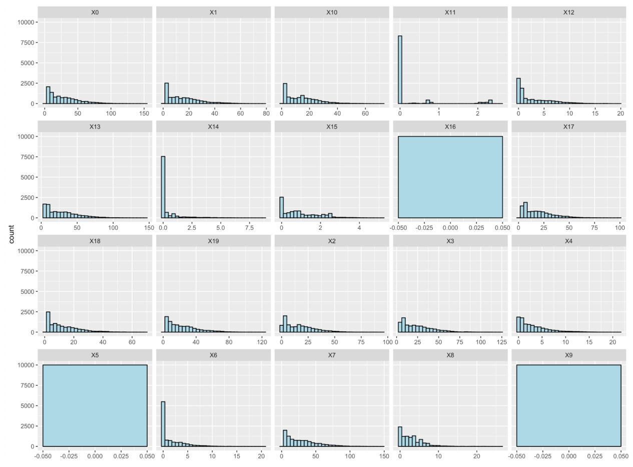


Figure 7: Latent Space distribution used in training of Actor/Critic

3.6 Implementation Details

After encoding all our video frames into a low-dimensional representation (dimension 20), we created 80/20 train/test split of dictionaries with frame number as the key and the latent space vector as the value. The testing frames/clips were from the last 20% of video. We then selected 30 subsequent frames (approximately 10 second clips due to every 10th frame sampling from videos) every 15th key to get a total of 2000 walks through our latent space loaded into an environment. In future work, we would investigate other strategies for selecting these walks. Here's the original latent space that got the best results (notice the scale of the x-axis, a much more dense embedding). Every variable was always positive which was due to the usage of a RELU activation on the output layer. Three of the values were always 0 in the dataset, and quite a few other variables are under-utilized (mostly 0).

The policy network returns a μ and σ vector to represent a 20 dimensional Gaussian distribution with a diagonal covariance matrix. The action is randomly sampled from this distribution. The policy network was a feed-forward residual network with three layers of $tanh$ activation functions, before being passed into μ and σ sides of the network. The μ-side was a residual network with another three layers with a final skip connection from the input layer. The σ side was a normal feed-forward network with leaky RELU activation functions – the last activation function is either exponentiation or an ELU activation .

Here is the psuedocode for the REINFORCE algorithm that we implemented, again adapted from [5]:

```
env = Environment(parameters)
for epoch in range(n_epochs):
    $\triangle\theta = 0$
    for trajectory in trajectories:
        run episode i=1, .., n
            $\{s^i_1, a^i_1, r^i_1, \ldots, s^i_{L_i}, a^i_{L_i}, r^i_{L_i}\} \sim \pi_\theta$
        compute returns: $v^i_t = \sum_{s=t}^{L_i} \gamma^{s-t} r^i_s$
        L = max([$L_1, \ldots, L_n$])
        for t in range(1, L):
            compute baseline: $b_t = \frac{1}{N} \sum_{i=1}^{N} v^i_t$
            for i in range(1,n):
                $\triangle\theta = \triangle\theta + \alpha \nabla_\theta \, log \, \pi_\theta(s^i_t, a^i_t) \, (v^i_t - b_t)$
        $\theta = \theta + \triangle\theta$
```

For the subsequent state when running an episode, you can either use the action as the subsequent state (i.e., the point we sample from the distribution which we think is the next frame), or the actual next state/frame provided from the environment. The former can be more difficult to train as the losses can get potentially very large (due to predicting a walk in the wrong overall direction), but results in more robust long-term oriented policies.

We tried tons of different (invertible) functions to shape/normalize our rewards due to experiencing many exploding gradient problems due to large losses. The REINFORCE training method was quite slow with a huge variance in the rewards throughout training. The more successful (and stable) training was through an implementation of advantage actor critic. We adopted some code from [10], but changed the updates for the policy network to. A decent variant worth testing is actually computing the long-term rewards before updating the critic network. The idea of updating the policy only at the end of the trajectory is to give the critic network an opportunity to adjust to the current policy (as it's supposed to be an on-policy value function).

```
env = Environment(parameters)
$\pi_\theta, V^{\pi_\theta}_\phi$ = actor, critic
for epoch in range(n_epochs):
    for trajectory in trajectories:
        done = False
        $s_i$ = env.make_start_state()
        $\triangle\theta = 0$
        while not done:
            $a_i = \pi_\theta(s_i)$
            $s_{i+1}, r_i, done = P_M(s_i, a_i)$ #env.state_and_reward
            if not done:
                A = $r_i + \gamma V^{\pi_\theta}_\phi(s_{i+1}) - V^{\pi_\theta}_\phi(s_i)$
            else:
                A = $r_i - V^{\pi_\theta}_\phi(s_i)$
            $\triangle\phi = \beta \nabla_\phi A^2$
            $\phi = \phi + \triangle\phi$
            $\triangle\theta = \triangle\theta - \nabla_\theta log \pi_\theta(s_i, a_i) * A$
            $s_i = s_{i+1}$
        $\theta = \theta + \alpha \triangle_\theta$
```

4 Results

We never got a divergence in rewards from training and test set evaluations which indicates that our policy networks were underfitted to the data. Nonetheless, we were able to take a single frame from the unseen test dataset, recursively feed it through our policy network, sample an action to build an entire sequence of future 10th frames. After performing a linear interpolation between the predicted frames, we decoded the vectors and constructed 15 second, 30fps videos of future animal predictions. One of the most interesting aspect was that the videos generated from an animal in the middle of feeding shows aperiodic cycles (differing lengths of time) of typical feeding activity.

You can view videos here with these (clickable) links: gaze of raccoons feeding; apossum walking around; deer and raccoon feeding.

There are some interesting limitations to using a latent-space embedding to predict the dynamics of the physical system. If you take a look at the Opossum walking around, there was not a lot of data of animals walking into or out of frames in the dataset (also indicated by the clarity of the opossum frames), and so predictions are restricted to the features in the latent space. You can further see that some of the fruits/nuts/seeds phase in and out of existence on the ground, clearly there is a preference for animal location changes as opposed to temporal consistency of other objects. This seems to indicate that this approach might not explicitly capture object permanence in exchange for capturing the things which tend to be much more variable between frames. Because of the frame skipping, not all locations of the deer legs were incorporated into the latent space, and so some linear interpolations don't have smooth transitions.

4.1 Discussion

While the results were good and showed a lot of promise, there were aspects of this project that could be improved or changed to make the results even better. For start, more data always helps. By having more video data to work with, we could get a more densely filled latent space, which should fill in more of the unknown gaps and would cut down on the number of unrealistic images when interpolating. Attempting to implement some of the techniques for VAE's in [7] could help avoid some of the issues we had with embedding our data into our prior distribution caused by some common issues with adversarial training. Moreover, that training method could have solved some of the bluriness in the reconstructions. More data is useful for more trajectories to train the policy network on, as well. These trajectories are more training data for learning the potential paths these animals take. Finally, with more data, we would be able to find frames close together in the latent space that diverge in behavior. This allows for more variation in our predictions.

Future work could be through different approaches that we didn't have time to try. We could try replacing the reconstruction error in the autoencoder with intersection over union loss of an object tracker or the MSE of the final feature layer of a pre-trained image classifier (ImageNet, ResNet, etc). This would allow us to explicitly track where objects are in the reconstructed images. Because the policy network provides an explicit probability distribution for the next frame, it's worth investigating if the trained policy network can be used to detect anomalies in how some complex system unfolds relative to past data. Finally, it's worth trying other continuous control training algorithms, such as Proximal Policy Optimization to reduce the chances of certain weight updates being too large and causing a huge setback in maximizing long-term rewards.

5 Conclusion

In the end, we were successful in being able to predict where and how the animals may move in a realistic manner. Reinforcement Learning offers an interesting approach to modeling complex systems without worrying about the constraints we often see and have to worry about in other simpler systems (i.e. simple pendulums, spring systems, etc...). This approach appears to be less sensitive to latent space embedding shape or dimensionality than prior work. These models learn to make realistic next-frame predictions, and by maximizing long-term rewards, produce realistic and stochastic long-time-frame oriented predictions which makes them applicable in a wide variety of problems.

References

[1] Marcin Andrychowicz et al. *What Matters In On-Policy Reinforcement Learning? A Large-Scale Empirical Study*. 2020. arXiv: `2006.05990 [cs.LG]`.

[2] Marissa C. Connor, Gregory H. Canal, and Christopher J. Rozell. *Variational Autoencoder with Learned Latent Structure*. 2020. arXiv: `2006.10597 [stat.ML]`.

[3] Sam Greydanus, Misko Dzamba, and Jason Yosinski. "Hamiltonian Neural Networks". In: *CoRR* abs/1906.01563 (2019). arXiv: `1906.01563`. URL: `http://arxiv.org/abs/1906.01563`.

[4] Alireza Makhzani et al. *Adversarial Autoencoders*. 2016. arXiv: 1511.05644 [cs.LG].

[5] Hongzi Mao et al. “Resource Management with Deep Reinforcement Learning”. In: *Proceedings of the 15th ACM Workshop on Hot Topics in Networks*. HotNets ’16. New York, NY, USA: Association for Computing Machinery, 2016, pp. 50–56. DOI: 10.1145/3005745.3005750.

[6] Volodymyr Mnih et al. “Human-level control through deep reinforcement learning”. In: *nature* 518.7540 (2015), pp. 529–533.

[7] Danilo Jimenez Rezende and Fabio Viola. “Taming vaes”. In: *arXiv preprint arXiv:1810.00597* (2018).

[8] Andrew M. Saxe, James L. McClelland, and Surya Ganguli. *Exact solutions to the nonlinear dynamics of learning in deep linear neural networks*. 2014. arXiv: 1312.6120 [cs.NE].

[9] John Schulman et al. *High-Dimensional Continuous Control Using Generalized Advantage Estimation*. 2018. arXiv: 1506.02438 [cs.LG].

[10] Phil Tabor. *Actor Critic Continuous*. https://github.com/philtabor/Youtube-Code-Repository/tree/master/ReinforcementLearning/PolicyGradient/actor_critic. 2018.