Project

Learning - Based Medical Image Registration

by: Jayashri Masilamani

Abstract

Medical image registration is the alignment of two or more medical images. Information gained from the two images are usually used for efficient diagnostic and therapeutic purposes. Recent advancement in deep learning research and computational efficiency provides a way to design a recurrent neural network (RNN) as optimizer that can replace the hand-designed optimizers like gradient descent to perform the medical image registration task. Major problem to implement this approach is to train the RNN optimizer for the given input images. To overcome the training problem, State of the art reinforcement learning (RL) algorithm called Proximal Policy Optimization (PPO) algorithm is used with RNN variant called Long-short term memory (LSTM) network to design the optimizer. Then, various analytical experiments were carried out with the PPO-LSTM optimizer to understand its performance and learn its capability. Finally, the 2D-2D image registration task is implemented with the PPO-LSTM optimizer and the performances of the experiments are analysed.

Task of the Thesis in the Original:

Contents

List of Acronyms

CT	Computed Tomography
MR	Magnetic Resonance
SPECT	Single-Photon Emission Computed Tomography
PET	Photon Emission Computed Tomography
DOF	Degree of Freedom
PDF	Portable Document Format
NCC	Normalized Correlation Coefficient
LSTM	Long Short-Term Memory networks
RNN	Recurrent Neural Networks
GRU	Gated Recurrent Unit
RL	Reinforcement Learning
MDP	Markov Decision Process
POMDP	Partially observed Markov decision process
MLE	Maximum Likelihood Estimate
L2L	Learning to learn
SGD	Stochastic Gradient Descentt
CMEAS	Covariance Matrix Adaptation Evolution Strategy
TRPO	Trust Region Policy Optimization
PPO	Proximal policy optimization
AI	Artificial Intelligence

List of Figures

List of Tables

1 Introduction

The introduction should present the topic of the thesis to specify the purpose and importance of the work. Other possible contents of an introduction are described in section **??** on page **??**.

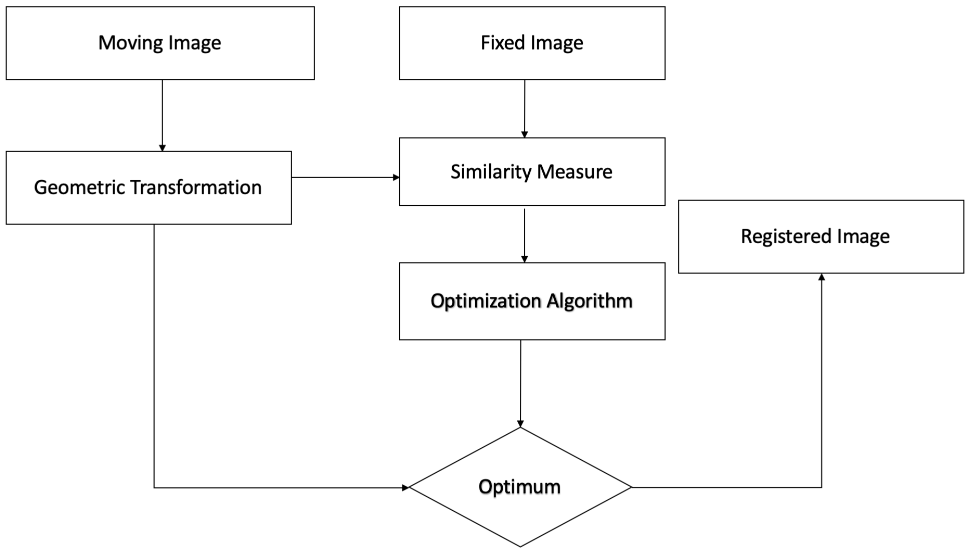


Figure 1.1: Block diagram represents the steps involved in image registration task

2 Background

2.1 Image Registration

2.1.1 Introduction

Image registration is the process of finding a geometrical transformation that aligns two images so that corresponding voxels/pixels can be superimposed on each other. Image registration is a crucial part for image analysis and accurate integration (or fusion) of the useful information from two or more images is very important. In the medical field, medical image registration applications occur throughout the clinical track of events; not only within clinical diagnostics settings, but prominently so in the area of planning, consummation, and evaluation of surgical and radio-therapeutic procedures like image-guided radiation therapy, image-guided radiation surgery, and image- guided minimally invasive treatments. Common tomographic modalities that acquire three-dimensional images are computed tomography (CT), magnetic resonance (MR) imaging, single-photon emission computed tomography (SPECT), and positron emission tomography (PET). These modalities contain contiguous set of two-dimensional slices provides a three-dimensional array of image intensity values. Typical two-dimensional images may be x-ray projections captured on film or as a digital radiograph. In all cases, primarily digital images stored as discrete arrays of intensity values. The two images are typically acquired from the same patient, in which case the problem is that of intra-patient registration, but inter-patient registration has application as well. Medical image registration techniques correlate the images acquired from different temporal and spatial sources and matches the anatomical points to allow a physician to obtain improved and detailed information. Variables like imaging principles and modalities, sampling time, and the physical state of a patient need to be considered when processing medical data because it impacts the image resolution.

2.1.2 Mathematical Background

Medical image registration techniques involve two input images: One image is defined as fixed image FIX(x) and another image is defined as moving image MOV(x). x represents the n-dimensional position in the image. A transform T(x) is a mathematical function which does the spatial mapping of points from the fixed image to points in the moving image. Thus, transform establishes the correspondence for each pixel in the fixed image to a position in the moving image. A similarity metric provides a measure of how well the fixed image matches the moving image. This measure forms a quantitative criterion to be optimized by an optimizer over the search space defined by the parameters of the transform. Combining the above part and the digital image technology, the registration procedure is formulated as an optimization problem in which a cost function C is minimized with respect to T. The optimizer adjusts the parameters of the transform in a way that minimizes the difference between the two images. The metric is a key component in the registration process. It uses information from the fixed and moving image to compute a similarity value. The derivative of this value tells us in which direction we should move the moving image for better alignment. The moving image is moved in small steps, and this process is repeated until a convergence criterion is met. The metric can use pixel intensities, point positions, pre-computed image features or anything we might want to optimize. We just have to define a metric for it. Sometimes a regularization term is added to the cost function to penalize unwanted transformations.

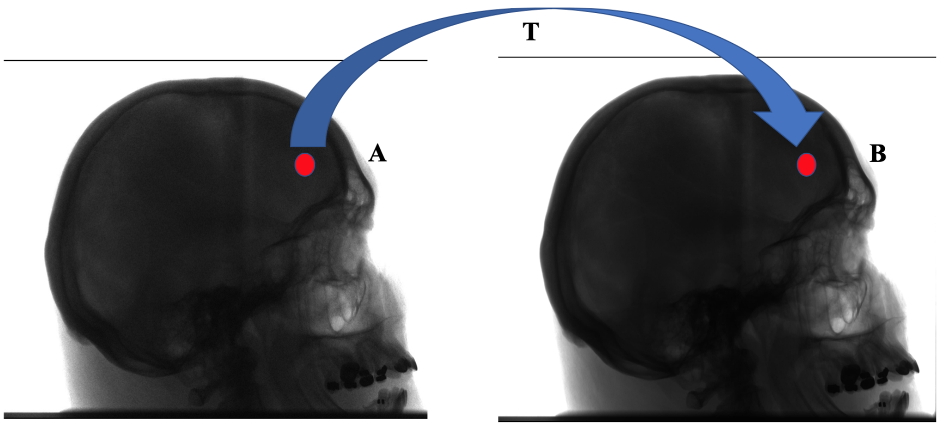


Figure 2.1: image-reg-diagram

2.1.3 Registration Components

In this section we introduce common terminology and some of the choices for different types of components.

Transforms

The choice of transform is the important factor for the success of image registration. The transform implies the desired type of transformation and constrain the solution space to that type of deformation. For example, in the case of registering the bones it would be sufficient only rigid transformations, but the cross-sectional study demands more flexible transformation models so that normal anatomical variability between patients will be taken in to consideration. The degree of freedom (DOF) of the transformation is determines to the number of parameters of the transform. DOF hugely varies from depending upon the registration method. For a simple registration like 3D translation, DOF is 3 and to anywhere between hundreds and millions of DOFs for b-spline deformation fields and non-parametric methods. It is often a good idea to start with simple transforms and propagate solutions through transforms of gradually increasing complexity.

2.1.4 Rigid Registration

In this thesis, we are going to use rigid registration for spine images. Rigid registration or Rigid transformations, or rigid mappings is one of the simplest of methods which belongs to the category of linear transformation models. Rigid-body transformations consist of only rotations and translations. When the registration problem is between the images of same patient with merely taken in different positions, then just translation and rotations are enough to describe the required registration. This is called rigid transformation. Rigid registration preserves the straightness of lines (and the planarity of surfaces) and all angles between straight lines.

2.1.5 Metrics

The similarity metric gives a value which tells about the degree of similarity between the moving and fixed image and is a key component in the registration process. The metric samples intensity values from the fixed and transformed moving image and evaluates the fitness value and derivatives This value will be given to the optimizer. Selecting an appropriate metric is highly dependent on the registration problem to be solved. For example, some metrics have a large capture range while others require initialization close to the optimal position. In addition, some metrics are only suitable for comparing images obtained from the same imaging modality, while others can handle inter-modality comparisons. There are no -¬-clear-cut rules as to how to choose a metric and it may require a trial-and-error process to find the best metric for a given problem. The Mean Squared Difference (SSD) metric computes the mean squared pixel-wise intensity differences between the fixed and moving images. The optimal value of the metric is zero. Poor matches are result in large values of the metric. The metric samples intensity values from the fixed and transformed moving image and evaluates the fitness value and derivatives, which are passed to the optimizer. This metric relies on the assumption that intensity representing the same homologous point must be the same in both images and only suited for two images with the same intensity distributions, i.e. for images from the same modality. Normalized Correlation Coefficient (NCC) computes pixel-wise cross-correlation normalized by the square root of the autocorrelation of the images. The metric is invariant to linear differences between intensity distributions and is therefore particularly well suited for intra-modal CT registration where intensity scales are always related by a linear transform even between scanners. This metric produces a cost function with sharp peaks and well-defined minima, but therefore has a relatively small capture radius.

2.1.6 Normalized Correlation Coefficient (NCC)

The SSD measure makes the implicit assumption that after registration the images differ only by Gaussian noise. The cross-correlation is based on the assumption that there is a linear relation between the intensities of the corresponding structures in both images. Thus, the larger the cross- correlation is, the better the registered image is.

$$CrossCorrelation = \frac{\sum_i (A(i) - \tilde{A})(B(i) - \tilde{B})}{\sqrt{\sum_i (A(i) - \tilde{A})^2 \sum_i (B(i) - \tilde{B})^2}} \tag{2.1}$$

2.1.7 Optimizers

The objective of optimization is to determine the values for a set of parameters for which some function of the parameters is minimized (or maximized). An iterative optimization algorithm aims at reducing the search time in optimization in order to increase the time sensitivity of the algorithm. One of the simplest cases involves determining the optimum parameters for a model in order to minimize the sum of squared differences between a model and a set of real-world data (χ^2).The usual approach is to make an initial parameter estimate and begin iteratively searching from there. At each iteration, the model is evaluated using the current parameter estimates, and (χ^2) computed. A judgement is then made about how the parameter estimates should be modified, before continuing on to the next iteration. The optimization is terminated when some convergence criterion is achieved (usually when (χ^2) stops decreasing). Common optimizers are Gradient Descent (GD), Robbins-Monroe (RM), Adaptive Stochastic Gradient Descent (ASGD), Conjugate Gradient (CG), Conjugate Gradient FRPR, Quasi-Newton LBFGS, Simultaneous Perturbation (SP), CMAEvolutionStrategy.

2.2 Deep Learning / Deep Neural Networks

2.2.1 Introduction

Deep Neural networks are a set of algorithms, modeled loosely after the human brain, which are very efficient to recognize pattern without the need of hand-made features. The patterns they recognize are commonly from images, sound, text or time series. The 'Deep' in Deep Neural networks refers a large number of layers between input and output. Each layer contains many units with weights and biases, with which the neural network improves its ability to approximate more complex functions.

2.2.2 Neural Network as Function approximation in Reinforcement Learning

In this thesis as described in previous chapter, we use the neural networks as function approximation in reinforcement learning by considering its use in estimating the value function and policy approximation. The approximate value and policy function is represented as a parameterized functional form with weight vector w $\in R_d$. As our goal is to learning to learn approach for training recurrent neural networks to perform black-box global optimization in various tasks. It would be important to understand the basics and nuances of the recurrent neural network. The recurrent network that is used in this thesis is called LSTM which stands for Long Short-Term Memory networks. A brief walkthrough of recurrent neural network and its variant called LSTM are as follows:

2.2.3 Recurrent Neural Networks (RNN)

Recurrent nets are a type of artificial neural network that are designed to recognize patterns in sequences of data such as time series, sensor data etc. The input of RNN is not just the current input but also it considers the inputs that they perceived previously in time. RNN are networks with loops in them, which allows past and current information to persist in each time step.

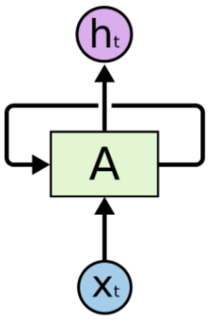


Figure 2.2: Recurrent Neural Networks Unit [cite]

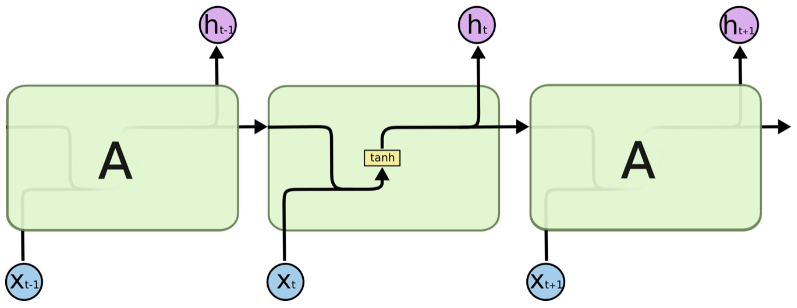


Figure 2.3: An unrolled standard RNN [cite]

RNN preserves sequential information in the hidden state so in each time step the information in hidden state influence the processing of new input. It is finding correlations between events separated by many moments, and these correlations are called “long-term dependencies”. So, we can picture as network that share weights over time.

The process of carrying memory forward by hidden state can be expressed mathematically:

$$h_t = \phi(Wx_t + Uh_{t-1}) \tag{2.2}$$

h_t represents the hidden state at time step t. As shown above, hidden state is the function of the input at the same time step x_t, multiplied by a weight matrix W added to the hidden state of the previous time step h_{t-1} which is multiplied by its own hidden - to-hidden-state matrix U, which is also called as transition matrix. The weight matrices act like a filter that estimate how much importance to allot to both the present input and the past hidden state. The error they produce will return back via backpropagation and be used to adjust their weights until minimum error that the network can reach. The squashing function ϕ used on the summation of the weight input and hidden state - either a logistic sigmoid function or tanh. The squashing function will efficiently condense the very large or small values given into a value in logistic space. Thus, makes the gradients values workable for backpropagation.

As this feedback loop occurs at every time step in the series, each hidden state contains traces not only of the previous hidden state, but also of all those that preceded h_{t-1} for as long as memory can persist.

2.2.4 The Problem of Long-Term Dependencies in RNN

Recurrent nets seek to establish connections between a final output and events many time steps it has seen before. but RNN finds it very difficult to know how much importance to accord to remote inputs. When the distance in time steps between the information acquired by network and the place where it's needed is small, RNN can efficiently correlate the information with the task. For example, the task where RNN has to look back some recent information. Then it can learn from the information and perform the task pretty easily. But when the gap between the relevant information and the point where it is needed is very large, RNNs become unable to learn to connect the information. In theory, RNNs are absolutely capable of handling such "long-term dependencies." Unfortunately, in practice, RNNs don't seem to be able to learn them. This is partially because the information flowing through neural nets passes through many stages of multiplication. This series of multiplication of weights in each step results in two main problems called Exploding gradients and Vanishing gradients. Exploding gradients can be solved relatively easily, because they can be truncated or squashed. Vanishing gradients can become too small for computers to work with or for networks to learn – a harder problem to solve. The above constraints of RNN can be successfully eliminated by LSTM networks.

2.2.5 LSTM Networks

LSTMs are proposed by the German researchers Sepp Hochreiter and Juergen Schmidhuber as a solution to the vanishing gradient problem. LSTMs also have this chain like structure, but the repeating module has a different structure. Instead of having a single neural network layer, there are four, interacting in a very special way called gated cells.

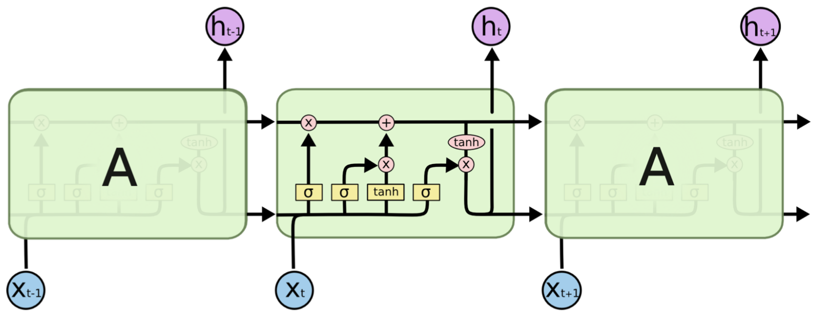


Figure 2.4: The unrolled LSTM with four interacting layers [cite]

2.2.6 The Core Idea Behind LSTMs

The key to LSTMs is the cell state, the horizontal line running through the top of the diagram. The cell state acts like a conveyor belt. It runs straight down the entire LSTM chain, with only some minor linear interactions. The LSTM does have the ability to remove or add information to the cell state, carefully regulated by structures called gates. These gates are analog, implemented with element-wise multiplication by sigmoid, which are all in the range of 0-1. Thus, makes it differentiable, and therefore suitable for back propagation. An LSTM has three of these gates, to protect and control the cell state.

LSTMs helps to preserve the error that can be back-propagated through time and layers. By maintaining error of some acceptable range, they allow recurrent nets to continue to learn over many time steps (over 1000), thereby it can link causes and effects remotely. The gates act on the signals they receive, they block or pass on information based on its strength and import, which they filter with their own sets of weights. Those weights, like the weights that modulate input and hidden states, are adjusted via the recurrent networks learning process. That is, the cells learn when to allow data to enter, leave or be deleted through the iterative process of making guesses, back-propagating error, and adjusting weights via gradient descent.

2.2.7 LSTM Architecture

The first step in LSTM is the flow of input and hidden unit values to the forget gate layer which consist of sigmoid layer decide what information are going to be thrown away from the cell state. The input of forget gate are h_{t-1} and x_t and it outputs a number between 0 and 1 for each member in the cell state C_{t-1}, where 1 represents "completely keep this information" while a 0 represents "completely get rid of this."

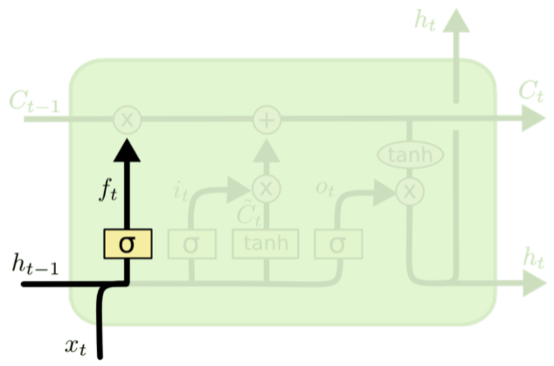


Figure 2.5: LSTM Unit with forget gate layer is highlighted

$$f_t = \sigma(W_f \cdot [h_{t-1}, x_t] + b_f) \tag{2.3}$$

The next step in the lstm cell is input gate layer which decides what new information are going to be stored in the cell state. The input gate layer has two parts. First, a sigmoid layer called the "input gate layer" decides which values we'll update. Next, a tanh layer creates a vector of new candidate values, C_t , that could be added to the state.

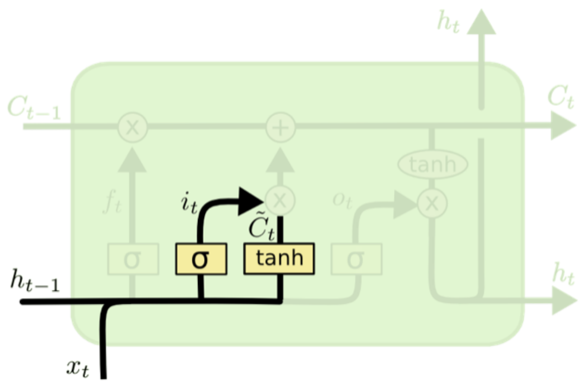


Figure 2.6: LSTM Unit with input gate and tanh layer are highlighted

$$\begin{aligned} i_t &= \sigma(W_i \cdot [h_{t-1}, x_t] + b_i) \\ \tilde{C}_t &= tanh(W_C \cdot [h_{t-1}, x_t] + b_C) \end{aligned} \tag{2.4}$$

In the next step, input gate layer and new candidate values are combined to create an update to the state. The new candidate values are created by the multiply the old state by ft Then add it with the multiplication of input gate and new candidate values ($i_t * C_t$).

$$C_t = f_t * C_{t-1} + i_t * \tilde{C}_t \tag{2.5}$$

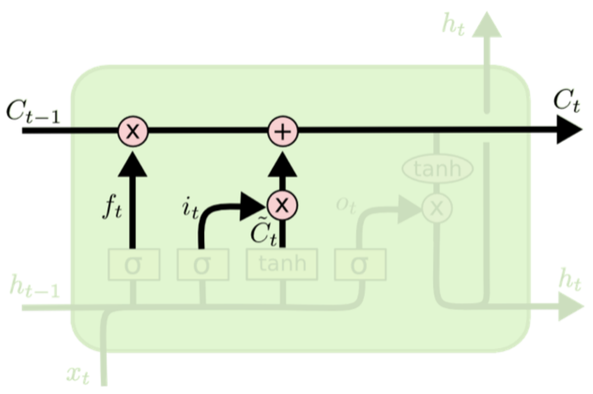


Figure 2.7: LSTM Unit with update gate layer is highlighted

Final step is the output gate, which decides what values are going to output. The output will be the filtered version of cell state. First, the hidden state value and input value is passed to sigmoid layer which decides what parts of the cell state are going to output. Then, the cell state is passed through tanh (whose values to be between minus 1 and 1) and multiply it by the output of the sigmoid gate, Thus, only the parts decided are passed to the output or next hidden state.

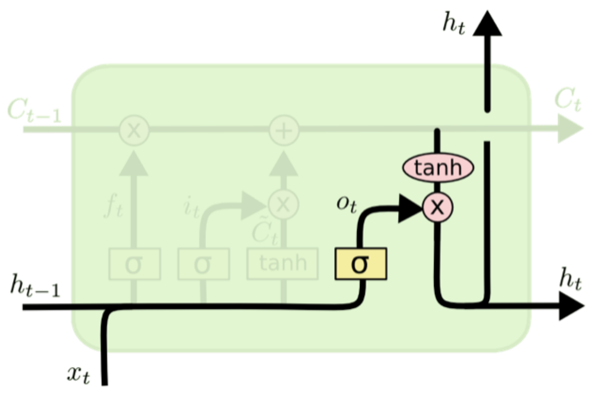


Figure 2.8: LSTM Unit with output gate layer is highlighted

$$O_t = \sigma(W_o \cdot [h_{t-1}, x_t] + b_o) \tag{2.6}$$

$$h_t = O_t * tanh(C_t) \tag{2.7}$$

2.2.8 Conclusion

These are only a few of the most notable LSTM variants. The most popular LSTM variant was introduced by Gers and Schmidhuber (2000), which has "peephole connections." These connections let the gate layers look at the cell state. Another variation is the coupled forget and input gates. Instead of separately deciding what to forget and what new information to be added, it makes the decisions together. The network only forget when its going to input something in its place. A slightly more dramatic variation on the LSTM is the Gated Recurrent Unit, or GRU, introduced by Cho, et al. (2014). It combines the forget and input gates into a single "update gate." It also merges the cell state and hidden state and makes some other changes. The resulting model is simpler and popular than the standard LSTM models. Attention network are more popular recently which is considered as next generation recurrent networks.

2.3 The Reinforcement Learning Problem

2.3.1 Introduction

Reinforcement Learning (RL) is a sub-class of machine learning domain. RL is a learning problem where the goal is to maximize a long-term reward. The RL system consists of an agent which interacts with the environment by taking an action and receives a reward. This transitions the agent into a new state. For each action, the environment feedbacks a new state and reward to the agent which can be seen in the following figure.

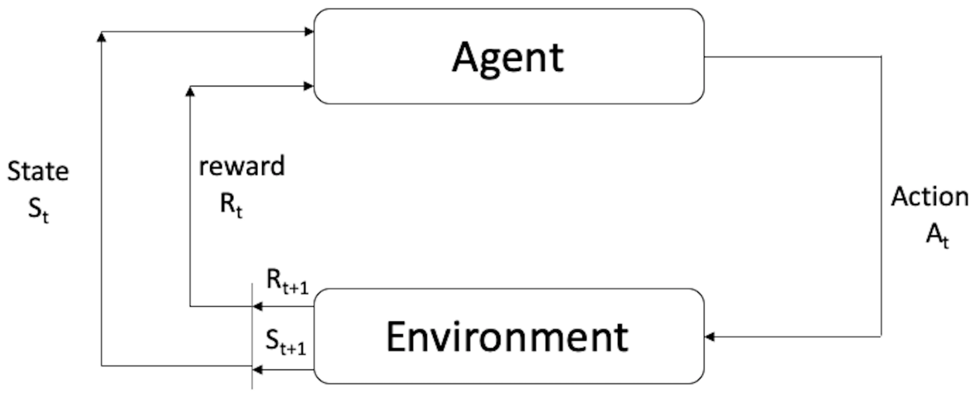


Figure 2.9: Agent - Environment interaction in a Markov decision process.

The main assumption of RL is The Reward Hypothesis which states that all goals and purposes of an agent can be explained by a single scalar called the reward.

The Reward Hypothesis: Maximization of the expected value of the cumulative sum of a received scalar signal (called reward).
Defining the right set of rewards for a given problem is called as ***reward shaping***. More formally, we look at the Markov Decision Process framework.

2.3.2 Markov Decision Process

A Markov Decision Process (MDP) is a mathematical framework that explains the interaction of an agent with a stochastic environment. This gives rise to a sequence of states, actions and rewards known as a ***trajectory*** and the objective is to maximize this set of rewards.

$$S_0, A_0, R_1, S_1, A_1, R_2 \tag{2.8}$$

Markov decision process (MDP) composes of following elements:

$$S, A, P, r, \rho_0, \gamma \tag{2.9}$$

MDP is defined by the following components:

- S: state space, a set of states of the environment.
- A: action space, a set of actions, which the agent selects at each time step.
- P (r, s' | s, a): a transition probability distribution. For each state s and action, a, P defines the probability distribution of environment which emits reward r and transition to state (s') accordingly.

An ***agent*** (e.g. RL Agent) observes the environment and takes actions. Rewards are given out, but they may be infrequent and delayed. Very often, the long-delayed rewards make it extremely hard to analyze the information and traceback what sequence of actions contributed to the rewards.

Policy: Agent's goal is to find a ***policy*** π, which maps states to actions. A policy is defined as the probability distribution of actions given a state.

$$\begin{gathered}\pi(A_t = a \mid S_t = s) \\ \forall A_t \in A(s), S_t \in S\end{gathered} \tag{2.10}$$

2.3.3 The episodic reinforcement learning problem

The tasks of this thesis are structured on episodic setting of reinforcement learning. An episode would consist of certain number of states, actions and rewards can explain agents experience with the environment. If the agent has access to all the states of environment, episodes of the reinforcement learning can be described in the following process. The first step of episode is sampling of initial state of the environment, s_0, from distribution $\mu(s_0)$. Then the agent selects an action a_t, sampled from the policy distribution. For the chosen action, environment gives out the next state and reward which depends on some distribution $P(s_{t+1}, r_t \mid s_t, a_t)$. An episode ends when it reaches the terminal state s_T. The episodic process can be explained by the following equations:

$$\begin{aligned} s_0 &\leftarrow \mu(s_0) \\ a_0 &\leftarrow \pi(a_0 \mid s_0) \end{aligned}$$

$s_1, r_0 \leftarrow P(s_1, r_0 \mid s_0, a_0)$
$a_1 \leftarrow \pi(a_1 \mid s_1)$
$s_2, r_1 \leftarrow P(s_2, r_1 \mid s_1, a_1)$
....
$a_{T-1} \leftarrow \pi(a_{T-1} \mid s_{T-1})$
$s_{T-1}, r_{T-1} \leftarrow P(s_T \mid s_{T-1}, a_{T-1})$

2.3.4 Partially observed problems

It is usual that agent has access only to an observation at each time step, which may give noisy and incomplete information about the state. The agent should consider information from many previous time steps, so the action picked by the agent depends on the past history i.e. past states and actions $h_t = (y_0, a_0, y_1, a_1, ..., y_{t-1}, a_{t-1}, y_t)$ and μ is the initial state distribution. The data-generating process is given by the following equations, and the figure below.

$s_0, y_0 \leftarrow \mu_0$
$a_0 \leftarrow \pi(a_0 \mid h_0)$
$s_1, y_1, r_0 \leftarrow P(s_1, y_1 \mid s_0, a_0)$
$a_1 \leftarrow \pi(a_1 \mid h_1)$
$s_2, r_1 \leftarrow P(s_2, y_2, r_1 \mid s_1, a_1)$
....
$a_{T-1} \leftarrow \pi(a_{T-1} \mid h_{T-1})$
$s_T, y_T \leftarrow P(s_T, y_T, r_{T-1} \mid s_{T-1}, a_{T-1})$

This process is called a Partially observed Markov decision process (POMDP). As we substitute Observation history h_t as the state of the system, we can consider partially-observed setting as the fully-observed setting. So POMDP can be expressed as an MDP (with infinite state space). When using function approximation, the partially observed setting is much closer with the fully-observed setting conceptually.

2.3.5 Policy Gradients

Policy gradients is one of the popular RL method. The objective of this method is to maximize the "expected" reward when following a policy π. The policy π_θ is defined by a set of parameters . The total reward for a given trajectory τ can be expressed as $r(\tau)$. The objective $J(\theta)$ is defined as the maximum 'expected' reward following a certain parametrized policy π_θ.

$$J(\theta) = E_\pi[r(\tau)] \tag{2.11}$$

All finite MDPs have at least one optimal policy (which can give the maximum reward) and among all the optimal policies at least one is stationary and deterministic.

To find the best objective J, we have to optimize the parameters to its best value. A standard approach to solve this maximization problem in Machine Learning Literature is to use Gradient Ascent. Because the parameter must improve the reward to the maximum value. In gradient ascent, we use gradient of objective function to keep stepping through the parameters by using following update rule.

$$\theta_{t-1} = \theta_t + \alpha \nabla J(\theta_t) \tag{2.12}$$

The gradient of the objective function $J(\theta)$ involves the expectation. Integrals usually make the computational setting complicated. It can be eliminated by expanding the expectation terms.

$$\begin{aligned} \nabla E_{\pi_\theta}[r(\tau)] &= \nabla \int \pi(\tau) r(\tau) d\tau \\ &= \int \nabla \pi(\tau) r(\tau) d\tau \\ &= \int \pi(\tau) \nabla log \pi(\tau) d\tau \end{aligned} \tag{2.13}$$

$$\nabla E_{\pi_\theta}[r(\tau)] = E_{\pi_\theta}[r(\tau) \nabla log \pi(\tau)] \tag{2.14}$$

The Policy Gradient Theorem: The derivative of the expected reward is the expectation of the product of the reward and gradient of the log of the policy .

$$\nabla E_{\pi_\theta}[r(\tau)] = E_{\pi_\theta}[r(\tau) \nabla log \pi(\tau)] \tag{2.15}$$

Next, we need to expand $\pi_\theta(\tau)$ the quantity to derive a practical formula. Using the chain rule of probabilities, we obtain

$$\begin{aligned} \pi_\theta[r(\tau)] = \mu(s_0)\pi(a_0 \mid s_0, \theta) P(s_1, r_0 \mid s_0, a_0) \pi(a_1 \mid s_1, \theta) \\ P(s_2, r_1 \mid s_1, a_1) \ldots\ldots \pi(a_{T-1} \mid s_{T-1}, \theta)), P(s_T, r_{T-1} \mid s_{T-1}, a_{T-1}) \end{aligned} \tag{2.16}$$

where μ is the initial state distribution.When we take the logarithm on both sides, the product turns into a sum, and when we differentiate with respect to π, the terms $P(s_T, r_{T-1} \mid s_{T-1}, a_{T-1})$ and $\mu(s_0)$ drops. Thus, We obtain

$$\nabla E_{\pi_\theta}[r(\tau)] = E_{\pi_\theta}[\sum_{t=0}^{T-1} \nabla log \pi(a_1 \mid s_1, \theta) r(\tau)] \tag{2.17}$$

This result is remarkable that we can compute the policy gradient without knowing anything about the system dynamics, which are encoded in transition probabilities P. As a result, all algorithms that use this result are known as “Model-Free Algorithms” because we don’t “model” the environment. The intuitive interpretation is that we collect

a trajectory, and then increase its log-probability proportionally to its goodness. That is, if the reward $r(\tau)$ is very high, we ought to move in the direction in parameter space that increases $log\pi$.
We can derive variants of this formula which is efficient by reducing variance of the reward. Some of the methods are described below:

2.3.6 REINFORCE (Baseline)

We can imagine the RL objective defined above as Likelihood Maximization (Maximum Likelihood Estimate). In an MLE setting, no matter how bad initial estimates are, in the limit of data, the model will converge to the true parameters. However, in a setting where the data samples are of high variance, stabilizing the model parameters can be extremely hard. In our case, any sample trajectory can cause a sub-optimal shift in the policy distribution. This problem is even worsened by the scale of rewards. Consequently, we instead try to optimize for the difference in rewards by introducing another variable called baseline **b**. To keep the gradient estimate unbiased, the baseline has to be independent of the policy parameters.

$$\nabla E_{\pi\theta}[r(\tau)] = E_{\pi_\theta}\Big[\Big(\sum(G_t - b)\nabla log\pi_\theta(a_t \mid s_t)\Big)\Big] \tag{2.18}$$

A close choice of baseline is the state-value function,

State Value: State Value is defined as the expected returns given a state following the policy π.

$$V(s) = E_{\pi_\theta}[G_t \mid S_t = s] \tag{2.19}$$

2.3.7 Actor Critic Methods

Another variant of policy gradient method is Actor Critic Methods.Gradient methods use a lot of samples to reach an optimal solution. Every time the policy is updated, we need to resample. Similar to other deep learning methods, it takes many iterations to compute the model. Actor critic model alleviates this problem to some extent.In this method, the actor models the policy π_θ and critic models the value function V. let us make approximate that as well using parameters ω to make $V^\omega(s)$. However, this time, we have to compute gradients of both actor and critic. The objective of critic is generally taken to be the Mean Squared Loss (or a less harsh Huber Loss) which needs to update the parameters ω of the critic and the parameters are usually updated using Stochastic Gradient Descent. In this method, we don't collect all samples until the end of an episode. This Temporal Difference technique also reduce variance.

2.3.8 Advantage Function

In deep learning, gradient descent works better when features are zero-centered. Intuitively, in RL, the absolute rewards may not be as important as how well an action does compare with the average action. That is the concept of the advantage function A.

An advantage function A is the difference of Action-value function Q and State value function V.

$$A^{\pi}(s_t, a_t) = Q^{\pi}(s_t, a_t) - V^{\pi}(s_t) \tag{2.20}$$

Action-value function Q(s,a) measures the expected discounted rewards of taking an action. For each state, if we can take k actions, there will be k Q-values. For optimal result, we take the action with the highest Q-value.

$$a^* = argmax_a Q^{\pi}(s_t, a_t) \tag{2.21}$$

Actor-Critic Policy Gradient

$$\nabla E_{\pi_\theta}[r(\tau)] = E_{\pi_\theta}\Big[A^{\pi}(s_t, a_t) \nabla log \pi_\theta(a_t \mid s_t)\Big] \tag{2.22}$$

Pseudocode

1. Agent takes action a $\sim \pi_\theta(a \mid s)$
2. Update Value function $V^{\pi}(s_t)$
3. Evaluate Advantage function $A^{\pi}(s, a)$
4. Calculate Objective $\nabla E_{\pi\theta}[r(\tau)] = E_{\pi\theta}[r(\tau) \nabla log \pi_\theta(a_t \mid s_t)]$
5. Update $\theta_{t-1} = \theta_t + \alpha \nabla J(\theta_t)$

3 Related Work

3.1 Learning to learn (L2L)

This paper explains an interesting concept that can replace normally used for neural network optimizers (e.g. Adam, RMSprop, SGD etc.) by a recurrent neural network. The common optimization algorithm like gradient descent is basically a sequence of updates (from the output layer of the neural net back to the input layer), in between which a state must be stored. Hence, we can design an optimizer with RNN networks. The learned algorithm is implemented by RNN variant called LSTM. Learning based algorithm tries to automate the optimizer by learning to exploit the underlying structure of the problem of interest.

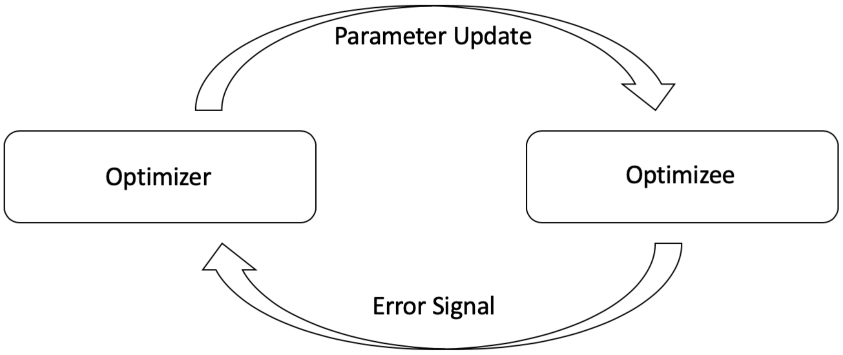


Figure 3.1: The optimizer (left) is provided with performance of the optimizee (right) and proposes updates to increase the optimizee's performance.

The machine learning task can be defined as the problem of optimizing an objective function $f(\theta)$ defined over some domain $\theta \in \ominus$. The optimizer has to find the minimizer θ^* i.e. $\theta^* = argmin_{\theta \in \ominus} f(\theta)$. The optimization algorithm that capable of finding this minimizer for a differential function are iterative and makes a sequence of updates according to the hand-designed update rules to reach the minimum.

$$\theta_{t+1} = \theta_t - \alpha_t \nabla f(\theta_t) \tag{3.1}$$

The idea proposed in this paper is to replace hand-designed update rules with learned update rule by utilizing LSTM capabilities. Let's call LSTM optimizer g which has its own set of parameters ϕ. Thus, we can modify update rule to the optimizee f of the form

$$\theta_{t+1} = \theta_t - g_t(\nabla f(\theta_t), \phi). \tag{3.2}$$

A high-level view of this process is shown in Figure 1. The recurrent neural network (RNN)models the update rule g by maintaining its own states and updates the weights as the function undergoes training iteratively. RNN-based learning algorithm performs well on a particular class of optimization problems because in contrast to analytically hand-designed optimizers, learning algorithms allows us to specify the class of problems through example problem instances. As the algorithm trains over problem instances, it generalizes well by gaining transfer knowledge between different problems. This ability is called transfer learning.

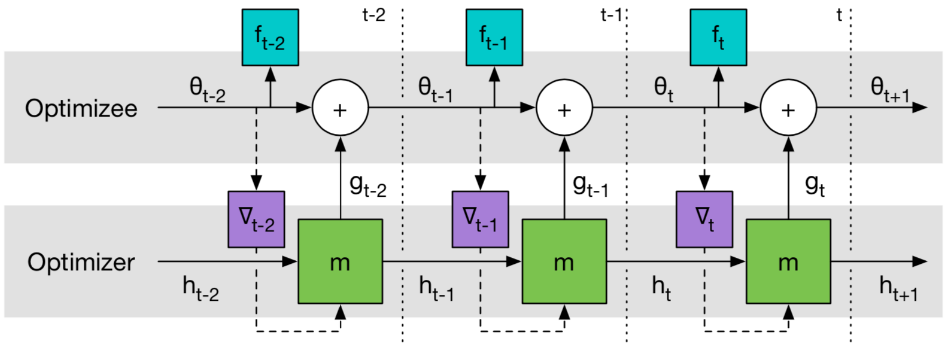


Figure 3.2: Computational graph used for computing the gradient of the optimizer [cite].

3.1.1 Loss Function

The optimizer is parametrized as ϕ and the final optimizee parameters can be written as $\theta^*(f, \phi)$ which the function of optimizer is parameters ϕ and the function in question. Given distribution of functions f the expected loss function can be written as

$$L_\phi = E_f[f(\theta^*(f, \phi))] \tag{3.3}$$

The update steps g_t gets the value from the output of a recurrent neural network m which is parameterized by ϕ and its hidden states are denoted by h_t. The objective function considers only the final parameter value to train the RNN optimizer. So, it is designed that the objective function takes the entire trajectory of optimization of horizon T.

$$L_\phi = E_f[\sum_{t=1}^{T} w_t f(\theta_2)] \tag{3.4}$$

where,

$\theta_{t+1} = \theta_t + g_t$

$$\begin{bmatrix} g_t \\ h_{t+1} \end{bmatrix} = m(\nabla_t, h_t, \phi)$$

$\nabla_t = \nabla_\theta f(\theta_t)$

The w_t are random weights for each timestep. It's better to set the last w_t to 1 and the rest to 0, we are optimizing for the best final result with our optimizee. This seems reasonable, but it makes it much harder to train. Instead we will use $w_t = 1$ for all t. In the above set of equations, f is the optimizee function, and θ_t is its parameters at time t. m is the optimizer function, ϕ is its parameters. ht is its state at time t. g_t is the update it outputs at time t. The plan is thus to use gradient descent on ϕ in order to minimize $L(\phi)$, which should give us an optimizer that is capable of optimizing f efficiently.

picture!!!!! The above diagram represents the RNN optimizer which takes and passes values to the optimizee. As the paper mention, it is important that the gradients in dashed lines in the figure below are not propagated during gradient descent.

3.2 Black box Optimization

This paper is related to the above L2L paper. The black box optimization is performed by training recurrent neural networks like LSTM as optimizer using learning to learn approach. The optimizer is trained on various smooth target functions and learned optimizers are allowed to learn the policies in reinforcement learning task and other black-box learning tasks. The problem is constructed as finding a global minimizer of an unknown (black-box) loss function f :

$$x^* = argmin_{x \in \chi} f(x), \tag{3.5}$$

where X is assumed as continuous search space. The black-box function f is not accessible by the optimizer but can be evaluated at a query point x in the domain. The evaluation gives out an output $y \in R$ such that $f(x) = [y|f(x)]$. That is, the function can be available to the optimizer only through the noisy point-wise observations y. A black-box optimization algorithm is designed as the following steps:

1. Given the current state of knowledge ht black box gives out a query point x_t.
2. Gets back the response y_t from the function of interest.
3. Update any internal states and produce h_{t+1}.

The above summarized steps can be designed using recurrent neural network (RNN) parameterized by θ such that

$$h_t, x_t = RNN_\theta(h_{t-1}, x_{t-1}, y_{t-1}), \tag{3.6}$$

$$y_t \sim p[y|x_t] \tag{3.7}$$

This can be seen as RNN updates its hidden state using data from the previous time step and then propose a new query point to the function that to be optimized. RNN can perform black-box optimization process iteratively with shared parameters. Figure 1 illustrates the computation of unrolled optimizer and cumulation of loss function.

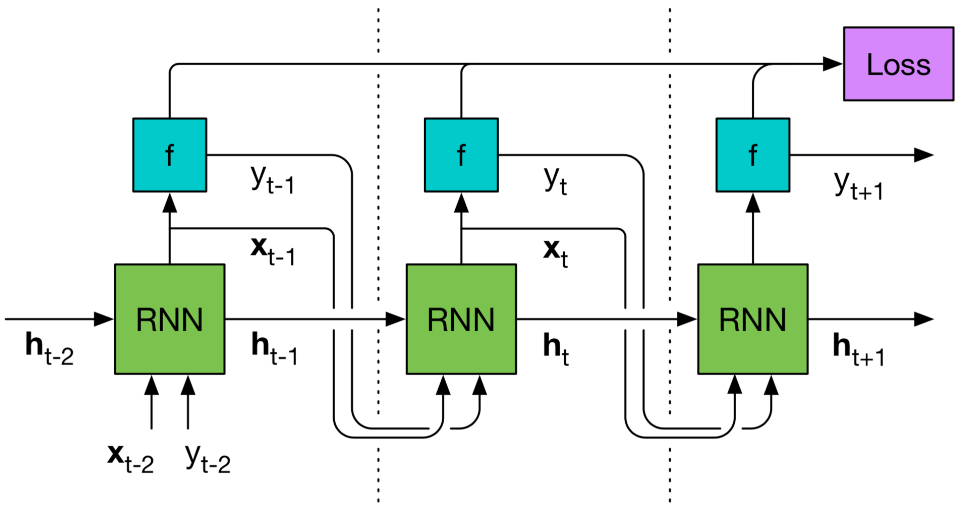


Figure 3.3: Computational graph of the learned black-box optimizer unrolled over multiple steps. The learning process will consist of differentiating the given loss with respect to the RNN parameters

3.2.1 Training of Optimizer

The optimizer can be trained simply by taking derivative of the loss function with respect to the RNN parameters θ and perform stochastic gradient descent (SGD). If the derivation of f with respect to θ is not available during the training time available, then it would be necessary to approximate these derivatives via an algorithm such as REINFORCE. The experiments that has performed in this paper have shown that the RNNs are massively faster than Bayesian optimization and RNN optimizers are recommended for applications involving a known horizon or where speed matters. The paper also points out RNN optimizers have shortcomings that Training for very long horizons is difficult.

3.3 Evolutionary Algorithm

CMEAS is the abbreviation of covariance matrix adaptation evolution strategy. The principle of evolution algorithm is loosely based on biological evolution. The algorithm provides the output a set of candidate solutions to evaluate a problem. An objective function evaluates the given set of candidates and return a fitness value. Based on the fitness results, the algorithm will then produce the next generation of candidate solutions that is more likely to produce even better results than the current generation. The iterative process will stop once the best-known solution is satisfactory for the user.

In the simple evolution strategies, firstly a set of solutions from a normal distribution with a mean μ and a fixed standard deviation μ will be sampled. The mean μ will be set to origin initially. After the evaluation of fitness value, the best solution in the population will be assigned as new mean value μ and next generation of solutions will be sampled around this new mean.

Major variation of CMEAS from other evolution strategy is that after the results of each generation in addition to adaptation of the mean μ and sigma σ parameters it also adaptively increases or decreases the search space for the next generation. At the end of each generation CMA-ES provides the parameters of a multi-variate normal distribution from which next generation candidates will sample solutions from.

CMA-ES makes use of covariance calculation formula in a way it can adapt well to an optimization problem which is describes as follows. Firstly, it considers only NBEST solutions in the current generation where NBEST is set to 25% of the best solutions. After calculating best solutions based on fitness value mean $\mu(g+1)$ of the next generation (g+1) is calculated as the average of only the best 25% of the solutions in current population (g). A set of equations below explains how to calculate the maximum likelihood estimate of a covariance matrix CC.

We first calculate the means of each of the x_i and y_i in our population:

$$\mu_x^{(g+1)} = \frac{1}{N_{best}} \sum_i^{N_{best}} x_i \tag{3.8}$$

$$\mu_y^{(g+1)} = \frac{1}{N_{best}} \sum_i^{N_{best}} y_i \tag{3.9}$$

Next, we use only the best 25% of the solutions to estimate the covariance matrix $C(g+1)$ of the next generation, but the clever hack here is that it uses the current generation's $\mu(g)$ rather than the updated $\mu(g+1)$ parameters that we had just calculated, in the calculation:

$$\sigma_x^{2,(g+1)} = \frac{1}{N_{best}} \sum_i^{N_{best}} (x_i - \mu_x^{(g)})^2, \tag{3.10}$$

$$\sigma_y^{2,(g+1)} = \frac{1}{N_{best}} \sum_i^{N_{best}} (y_i - \mu_y^{(g)})^2, \tag{3.11}$$

$$\sigma_{xy}^{2,(g+1)} = \frac{1}{N_{best}} \sum_i^{N_{best}} (x_i - \mu_x^{(g)})(y_i - \mu_y^{(g)}), \tag{3.12}$$

Armed with a set of μx, μy, σx, σxy, and σxy parameters for the next generation (g+1), now the next generation of candidate solutions can be sampled. Below is a set of figures to visually illustrate how it uses the results from the current generation (g) to construct the solutions in the next generation (g+1):

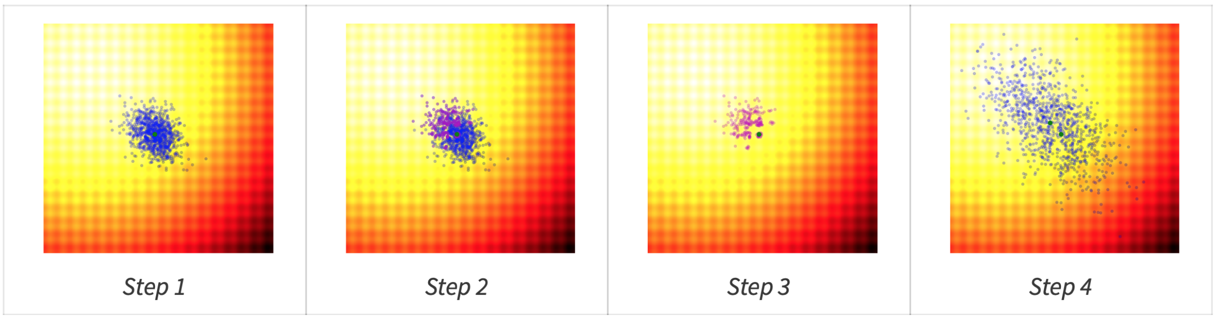


Figure 3.4: A set of figures to visually illustrate how it uses the results from the current generation (g) to construct the solutions in the next generation (g+1)

1. Fitness score is calculated for each candidate solution in current generation (g).
2. Purple samples indicate the best 25% of the population in generation (g).
3. The Covariance matrix C(g+1) of the next generation is calculated using only 25% of the the best solutions and the mean μ(g) of the current generation (the green dot).
4. Sample a new set of candidate solutions using the updated mean μ(g+1) and covariance matrix C(g+1)

Because CMA-ES can adapt both its mean and covariance matrix using information from the best solutions, it can make the search space wider or narrower when the best solutions are far away or closer respectively. The only real drawback is the performance efficiency, the increase in the number of model parameters increases the computational complexity of covariance calculation.

3.4 Experiments and Results

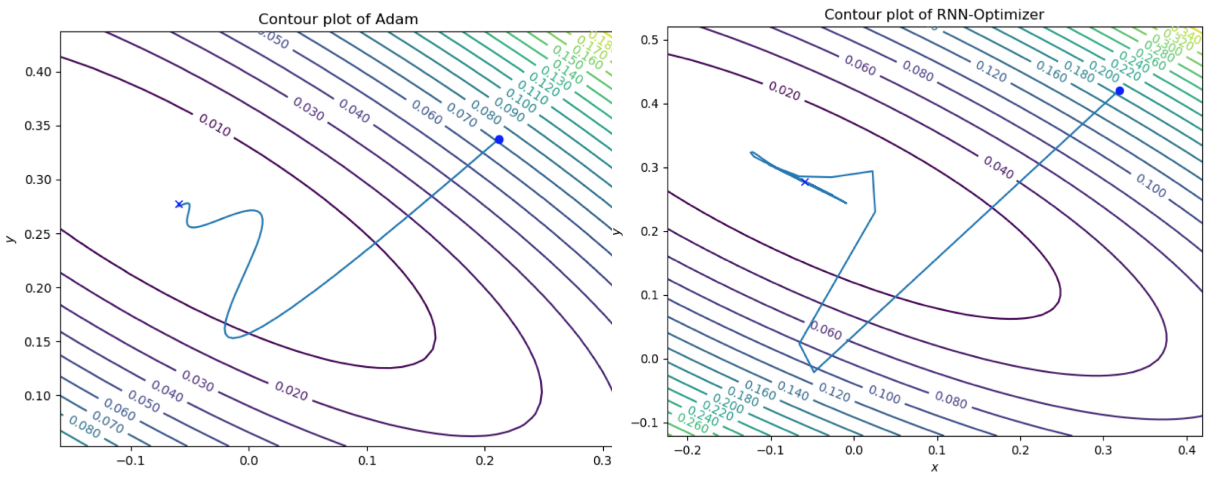


Figure 3.5: Contour plot of Adam [left] and L2L (RNN) [right] Optimizer.

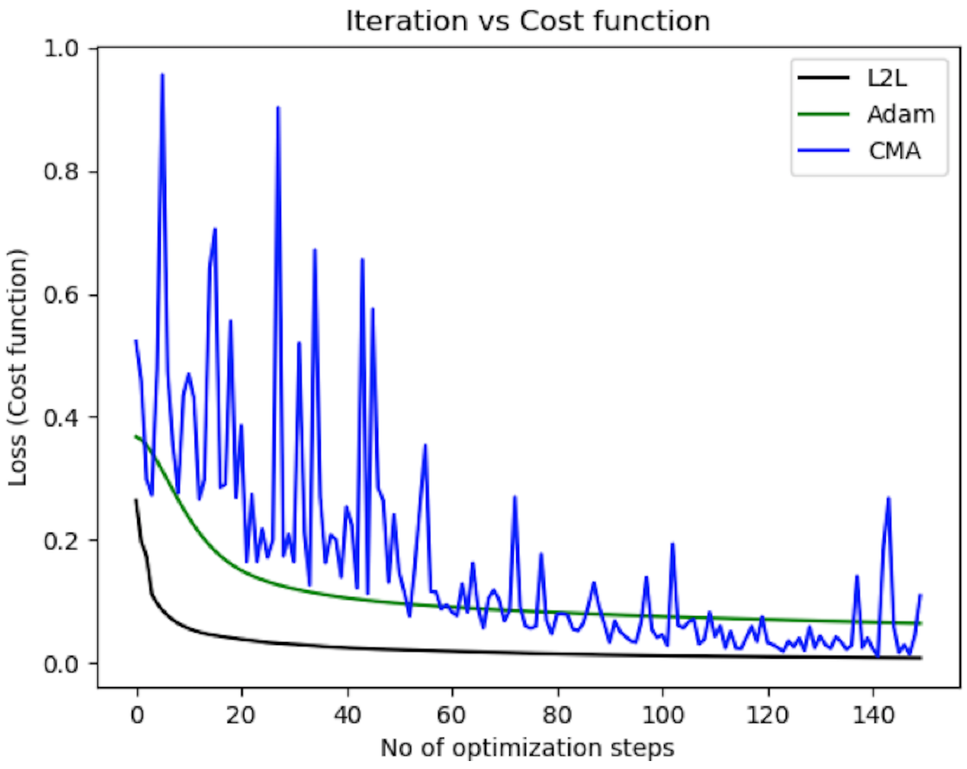


Figure 3.6: Iteration vs Cost Function of synthetic 2-dimensional quadratic functions.

4 Design

4.1 Proximal Policy Optimization Algorithms(PPO)

PPO is the latest member of the policy gradient methods in reinforcement learning, it interacts with the environment and optimizes “clipped surrogate” objective function using stochastic gradient ascent. Policy Gradient methods keeps the new and old policies in parameter space as close as possible. But even small change in parameter space can result a massive change in the algorithm performance. This tells us that even a single step change of parameter has the ability to collapse the entire policy performance. This makes the policy gradient methods dangerous to use large step sizes and it usually perform one gradient update per data sample these issues result in worse sample efficiency. Variants of policy gradients involves a second-order derivative matrix which makes it not scalable for large scale problems. The computational complexity is too high for real tasks. Intensive research is done to reduce the complexity by approximate the second-order method.

Trust Region Policy Optimization (TRPO) is one of the advanced and state of the art policy gradient algorithm. It provides better solution for many drawbacks of PG with its own disadvantages. TRPO tends to avoid the performance collapse and gives monotonically improving performance.

Disadvantage of TRPO are:

1. TRPO uses second order optimization which is relatively complicated.
2. It not compatible with architectures that include noise such as dropout or parameter sharing between the policy and value function, or with auxiliary tasks.

An algorithm was needed which provides monotonically improving performance i.e., each step in the training must be a constructive one. And also provide faster convergence, the step size has to be larger one and importantly computational complexity must be low.

PPO was proposed to overcome the drawbacks of TRPO, and to reduce the algorithm complexity. PPO has become the default reinforcement learning algorithm and go to algorithms at biggest research organization like OpenAI and to everyone because of its ease of use and good performance.

Proximal policy optimization (PPO) and Trust region policy optimization (TRPO) have some similarity.PPO and TRPO methods have the stability and reliability of trust-region methods but PPO is very simple to implement and have better empirical sample efficiency. PPO architecture is simple and yet provides reliable performance improvement with the use of first order optimization. Instead of imposing a hard constraint like TRPO, it formalizes the constraint as a penalty in the objective function and uses of first order optimization. The novel surrogate objective function with clipped probability ratios is used in PPO, which takes the pessimistic estimate of the function to avoid performance collapse and it calculates the surrogate objective function in multiple epochs of mini-batch updates to optimize the policy.

In paper[cite] PPO was compared to several previous algorithms from the literature. It performed performs better than the other algorithms on continuous control tasks like ATARI games. It was concluded that it performs significantly better in terms of sample complexity than A2C and similarly to ACER though it is much simpler.

4.1.1 Trust Region Methods

TRPO optimizes the surrogate objective function by applying a hard constraint on the size of the policy update. To eliminate the huge optimization step that ruins the training progress, TRPO formulates the region of maximum step size to explore and then locate the optimal point within this trust region.

$$\begin{gathered}\underset{\theta}{\text{maximum}}\ \hat{E}_t\left[\frac{\pi_\theta\left(a_t \mid s_t\right)}{\pi_{\theta old}\left(a_t \mid s_t\right)}\hat{A}_t\right] \\ \text{subject to } \hat{E}_t\left[\text{KL}\left[\pi_{\theta old}\left(\cdot \mid s_t\right), \pi_\theta\left(\cdot \mid s_t\right)\right]\right] \leq \delta\end{gathered} \tag{4.1}$$

Where δ is initial maximum step size which is the radius of the trust region and the objective is to find the optimal point within the radius δ. The trust region can be expanded or shrink in runtime to adjust to the curvature of the surface.Another variant of TRPO uses penalty to solve the unconstrained optimization problem for some coefficient β . It solves the function efficiently by using the conjugate gradient algorithm, after making a linear approximation to the objective and a quadratic approximation to the constraint.

$$\underset{\theta}{\text{maximum}}\ \hat{E}_t\left[\frac{\pi_\theta\left(a_t \mid s_t\right)}{\pi_{\theta old}\left(a_t \mid s_t\right)}\hat{A}_t - \beta\text{KL}\left[\pi_{\theta old}\left(\cdot \mid s_t\right), \pi_\theta\left(\cdot \mid s_t\right)\right]\right] \tag{4.2}$$

Mathematically, both KL penalized objective and the KL constrained objective are the same if we have unlimited computational resources. However, in practice, they are not.

TRPO that uses a hard constraint δ is much easier than a penalty. δ imposes a hard constraint to control the bad case scenarios in the policy space. δ restraints policy changes that can turn destructive. The results show it is hard to select a single fixed value β that perform better well across different problems or even within a single problem. As the characteristics change over the course of learning β need to be more adaptive. Therefore, trust region constraint is more popular. To optimize the penalized objective in Equation 5 with SGD further modifications are required.

4.1.2 Clipped Surrogate Objective

PPO strikes a balance between ease of implementation, sample complexity, and ease of tuning, trying to compute an update at each step that minimizes the cost function while ensuring the deviation from the previous policy is relatively small. Instead of imposing a hard constraint, PPO formalizes the constraint as a penalty in the objective function.This objective implements a way to do a Trust Region update which can be used with first-order optimizer like Stochastic Gradient Descent and simplifies the algorithm by removing the KL penalty and need to make adaptive updates.Even we may violate the constraint once a while, the damage is far less, and the computation is much simple. In its implementation, we maintain two policy networks.

π_θ is the current policy and $\pi_{\theta old}$ is the policy that we last used to collect samples.

With clipped objective, we compute a ratio between the new policy and the old policy: Let $r_t(\theta)$ denote the probability ratio. Algorithm has to maximize the surrogate objective.

$$r_t(\theta) = \left[\frac{\pi_\theta\Big(a_t \mid s_t\Big)}{\pi_{\theta old}\Big(a_t \mid s_t\Big)}\right] \tag{4.3}$$

Without a constraint, maximization of LCPI would lead to an excessively large policy update. hence, we now consider how to modify the objective, to penalize changes to the policy that move rt away from 1.

$$L^{CPI}\Big(\theta\Big) = \hat{E}_t\left[\frac{\pi_\theta\Big(a_t \mid s_t\Big)}{\pi_{\theta old}\Big(a_t \mid s_t\Big)}\hat{A}_t\right] = \hat{E}_t\left[r_t(\theta)\hat{A}_t\right] \tag{4.4}$$

But as we refine the current policy, the difference between the current and the old policy is getting larger. The variance of the estimation will increase, and we will make bad decision because of the inaccuracy. So, say for every 4 iterations, we synchronize the second network with the refined policy again.

This ratio measures how difference between two policies. We construct a new objective function to clip the estimated advantage function if the new policy is far away from the old policy. Our new objective function becomes:

$$L^{CLIP}(\theta) = \hat{E}_t\left[min\Big(r_t(\theta)\hat{A}_t, clip\Big(r_t(\theta), 1-\epsilon, 1+\epsilon\Big)\hat{A}_t\Big)\right] \tag{4.5}$$

In the above equation, the L^{CPI} and clipped L^{CLIP} compared and the minimum value which is the lower bound of unclipped objective was taken. This implies the change in probability ratio is not considered when it makes improvement in the objective and include it when it makes the objective function worse.

Let's look at the dynamics of the equation for different advantage values.

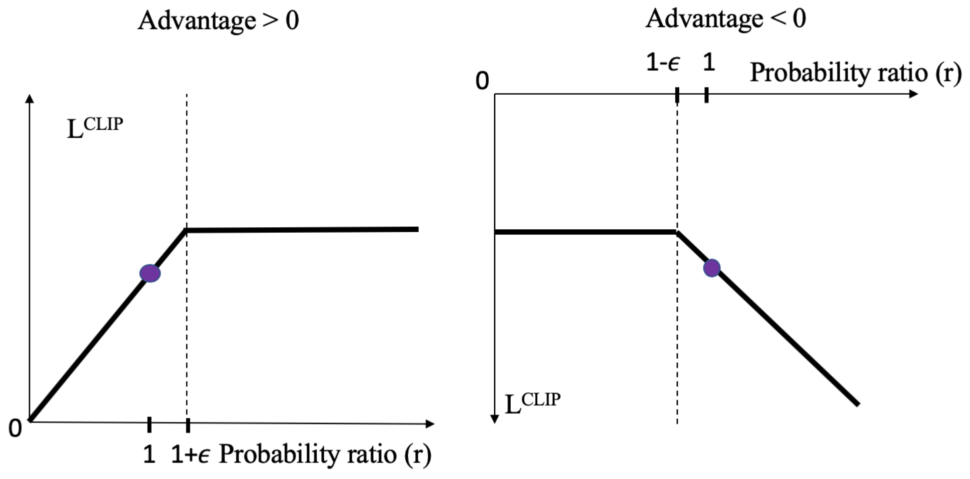


Figure 4.1: Plot showing a single a single timestep of the surrogate function L^{CLIP} as a function of the probability ratio r, for positive advantages(left) and negative advantages(right). The red circle on each plot represents the starting point for the optimization, i.e., r =1

Figure plots a single term i.e., a single t in L^{CLIP} note that the probability ratio r is clipped at $(1-\epsilon)$ or $(1+\epsilon)$, depending on whether the advantage is positive or negative. ϵ is the hyper parameter which corresponds to how far the new policy can move away from the old policy at the same time improving the objective.

Advantage is Positive: In this case the objective function is reduced to:

$$L^{CLIP}(\theta) = min\left(\frac{\pi_\theta(a_t \mid s_t)}{\pi_{\theta old}(a_t \mid s_t)}, (1+\epsilon)\right)\hat{A}_t \tag{4.6}$$

When the advantage is Positive, the objective will increase only if pi(theta) increases which implies the action is more likely. The minimum operator in the equation limits the increment range of objective to when the policy value goes above the ,

Advantage is Negative: In this case the objective function is reduced to:

$$L^{CLIP}(\theta) = max\left(\frac{\pi_\theta(a_t \mid s_t)}{\pi_{\theta old}(a_t \mid s_t)}, (1-\epsilon)\right)\hat{A}_t \tag{4.7}$$

When the advantage is Negative, the objective will increase only if pi(theta) increases which implies the action is less likely. The maximum operator in the equation limits the increment range of objective to when the policy value goes away from the , If the probability ratio between the new policy and the old policy falls outside the range , the advantage function will be clipped. epsilon is set to 0.2 for the experiments in the PPO paper.

4.1.3 Algorithm

The above-mentioned surrogate loss can be implemented with typical policy gradient implementation but with some modification. The surrogate loss simply can be constructed the loss L^{CLIP} and multiple steps of stochastic gradient ascent can be performed on it. The PPO architecture shares parameters between the policy and value function so the loss function consist of the policy surrogate and a value function error term.Usually, entropy bonus is also added with the objective to provide sufficient exploration. After incorporating the above discussed changes, the following objective function is obtained.

$$L_t^{CLIP+VF+S}\Big(\theta\Big) = \hat{E}_t\Big[L_t^{CLIP}\Big(\theta\Big) - c_1 L_t^{VF}\Big(\theta\Big) + c_2 S\left[\pi_\theta\right]\Big(s_t\Big)\Big] \tag{4.8}$$

where c_1 , c_2 are the coefficients, S is entropy bonus and L_t^{VF} is a squared-error loss.

The learned value function is also used for calculating advantage-function, the estimation obtained using the value function highly reduce the variance of the advantage function. Generalized advantage estimation Sch+15a , and the finite-horizon estimators in Mni+16 are the currently popular advantage estimation techniques. As the tasks in this thesis uses recurrent neural networks, the policy gradient estimation used in Mni+16 which are well suited for recurrent neural networks are used in this thesis. This estimation runs the policy for T timesteps and importantly length of T is much less than the episode length and uses the collected samples for an update. And a special type of advantage function is needed for this style which does not look beyond timestep T. An advantage estimator used Mni+16 can be suitable for this condition which is given below:

$$\hat{A}_t = -Vs_t + r_t + \gamma r_{t+1} + ... + \gamma^{T-t+1} r_{T-1} + \gamma^{T-t} V(s_T) \tag{4.9}$$

where t species the time index in [0, T], within a given length-T trajectory segment Generalizing this choice, we can use a truncated version of generalized advantage estimation, which reduces to Equation when lambda = 1

$$\begin{aligned}\hat{A}_t = \delta_t + (\gamma\lambda)\delta_{t+1} + \gamma r_{t+1} + ... + ... + (\gamma\lambda)^{T-t+1}\delta_T + 1 \\ where\ \delta_t = r_t + \gamma V(s_{t+1}) - Vs_t\end{aligned} \tag{4.10}$$

4.1.4 Conclusion

Proximal policy optimization, a family of policy optimization methods is explained in this chapter. It uses multiple epochs of stochastic gradient ascent to perform each policy update. PPO is very simple and easy to implement and still preserves the comparable stability and reliability as trust-region methods.The joint architecture of the policy and value function can be implemented with only few lines of code change to a vanilla policy gradient implementation and most importantly PPO has better overall performance.

5 Implementation

5.1 PPO (pseudocode)

Algorithm 1 PPO-Clip Algorithm

```
1: Input : Initialize policy parameter $\theta_0$, Initialize value function parameters $\phi_0$
2: for k=0,1,2.... do
3:     Collect set of trajectories dk by running policy $\pi_k$ in the environment.
4:     Compute reward $\tilde{R}_t$
5:     Compute Advantage function $\tilde{A}_t$ (using Generalized Advantage Estimate) based
       on the current value function $V_{\phi_k}$.
6:     Update the policy by maximizing the PPO-Clip objective:
7:
       $$\theta = \underset{\theta}{\operatorname{argmax}} \frac{1}{|D_k|\,T} \sum_{r \in D_K} \sum_{t=0}^{T} \left[ min\Big( r_t(\theta) A^{\pi_{\theta_k}}, clip\Big(r_t(\theta), 1-\epsilon, 1+\epsilon\Big) A^{\pi_{\theta_k}} \Big) \right]$$
        where,
8:
       $$r_t(\theta) = \left[ \frac{\pi_\theta\Big(a_t \mid s_t\Big)}{\pi_{\theta k}\Big(a_t \mid s_t\Big)} \right]$$
       typically via stochastic gradient ascent with Adam.
9:     Fit value function by regression on mean-squared error:
10:
       $$\phi_{k+1} = \underset{\phi}{\operatorname{argmax}} \frac{1}{|D_k|\,T} \sum_{r \in D_K} \sum_{t=0}^{T} \Big( V_\phi\Big(s_t\Big) - \tilde{R}_t \Big)^2,$$
        typically via stochastic gradient descent algorithm.
```

5.2 OpenAI Gym

Gym is an OpenAI toolkit. It helps to create a better benchmark for reinforcement learning. The gym library is a collection of test problems — environments — that you can use to work out your reinforcement learning algorithms. The main aim of this tool is to increase reproducibility in the field of AI and provide tools with which everyone can learn about basics of AI. Gym doesn't make any assumption about the structure of environment. Gym is easy to set up and improves the reproducibility. Gym's test environments work on RL agent's algorithms with shared interfaces allows to write general algorithms and testing them. Gym is compatible with any numerical computation library, such as TensorFlow or Theano.

5.2.1 Spaces

Spaces are the attributes that are initialized in the beginning of the environment. There are two kinds of spaces called ***action space*** and ***observation space***. These attributes are of type Space which provides the valid format for the actions and observations.

The Space can be either Discrete or Box.

The ***Discrete*** space allows a fixed range of non-negative numbers, so in this case valid actions are either 0 or 1.

The ***Box*** space represents an n-dimensional box and it allows the variables to be continuous. So valid values of action/observation must take a value in the given range of n-dimensional space.

5.2.2 GYM Environment

There are 3 mandatory methods that must present in the gym environment which are described below:

Reset

The process gets starts by calling ***reset()***, which initializes ***observation*** values.

Step

This function receives an input variable 'action' which would be the command given by RL algorithm. It would be a better to actually know how the input action influence the dynamics of the environment. That's exactly what the step function of the environment returns for us. In fact, Step function returns four values. They are:

- ***Observation***: Its an environment-specific object which represents the current observation of the environment.
- ***Reward***(float): This return value is the amount of reward achieved by the previous action. The reward scale varies depending on the environment, but as we know the goal of the algorithm is always to increase your total reward.
- ***Done***: (Boolean): Done returns a Boolean value. It indicates whether it's time to reset the environment again. Usually, most of the tasks would be divided into well-defined episodes. When the task reaches the end of the episode done will return True.
- ***Info***: Info is of dict type. Info contains diagnostic information useful for debugging. The information can be helpful for learning purpose(for example, it might contain the raw probabilities behind the environment's last state change). But, the information in the dictionary should not be used by agent for the learning of the environment.

Render

Render function is used for the visualization of actions taken by the algorithm and resulting changes in the environment. It can be used during evaluation to visualise the learned algorithm. Rendering would allow us to visually analyse how well the algorithm learned/learning the task. It also gives some insight about the dynamics of environment like whether the algorithm correctly learned the tasks or reward values are efficiently improves the learning.

5.3 Programming Language and libraries

PPO Algorithm is implemented in TensorFlow - python code.

Table 5.1: Libraries and versions used in the experiment

Hyperparameters	**Values**
Python	3.7.3
TensorFlow	1.14.0
OpenAI Gym	0.13.1
NumPy	1.16.4
Matplotlib	3.1.0
Tqdm	4.32.2
Six	1.12.0
OpenCV(CV2)	3.4.2
Glob	2.26.2
Tensor board	1.14.0
SciPy	1.2.1

6 Experiments

6.1 Linear Functions

Quadratic function Quadratic function is a polynomial function in which the highest-degree term is 2. Hence, it is also called a quadratic polynomial or polynomial of degree 2.

A quadratic polynomial may involve a single variable x (the univariate case), or multiple variables (the multivariate case) $[Wx - Y]^2$ In elementary algebra, such polynomials often arise in the form of a quadratic equation . $[Wx - Y]^2 = 0$.The solutions to this equation are called the roots of the quadratic polynomial, and may be found through the use of the quadratic formula.

Equation: $[Wx - Y]^2 = 0$

W- coefficients of x (weight)

Y - constant

6.1.1 2D-Quadratic function (continuous Actions)

Spaces

Action Space - Box space of (1,2) - Dimension, It's a continuous action space of range (-1, -1) to (1,1).
Observation space - Box space of (1,3) - Dimension. It's a continuous observation space of range (-1, -1,0) to (1,1, -18)

Step

Input variable 'action' - Discrete
Observation space - Box space of dimension (1,2). Input variable : 'action' from algorithm - Discrete value range of 1 to 16. Increment = [0.05,-0.05,0.1, -0.1,0.4,-0.4,1.6,-1.6] According to the action, X is added by the action indexed element of increment array. 1st element takes X value and 2nd value is the negative cost function calculated for X.
Reward : negative cost function Done - Bool, true when the cost is less than certain threshold say 0.02

Reset

Resets Weights and bias. Resets Variable and make random predictions for first step.

Render

Line plot cost function vs variable

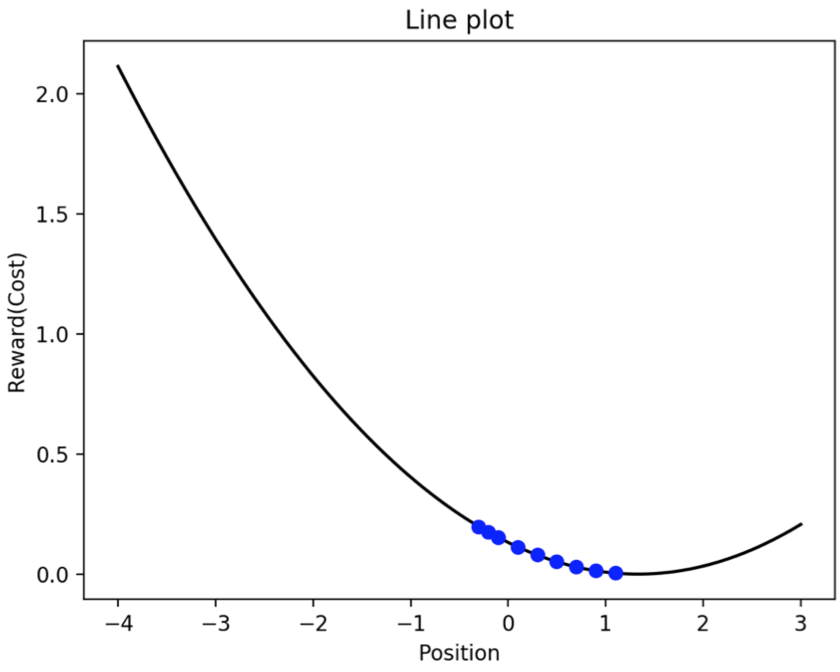


Figure 6.1: Rendering of 1D-Quadratic function (Continuous Actions)

6.1.2 1D-Quadratic function (Discrete Actions)

Explain equation and constants and variable.

Reset

Resets Weights and bias. Resets Variable and make random predictions for first step.

Step

Input variable ‘action’ - Discrete. Observation Box space of dimension (1,2) Input variable : ‘action’ from algorithm - Discrete value range of 1 to 16. Increment = [0.05,-0.05,0.1, -0.1,0.4,-0.4,1.6,-1.6] According to the action, X is added by the action indexed element of increment array. 1st element takes X value and 2nd value is the negative cost function calculated for X. Reward negative cost function Done Bool, true when the cost is less than certain threshold say 0.02

Render

Render function is as same as Continuous action Line plot cost function vs variable Image learned 1D (discrete)

6.1.3 2D-Quadratic function (Discrete Actions)

Reset

Reset Resets Weights and bias Resets Variable and make random predictions for first step

Step

Input variable ‘action’ - Discrete

Render

Contour plot - explains the trajectory of variables (reaches the minimum) for each episode Image: learned 2D (discrete)

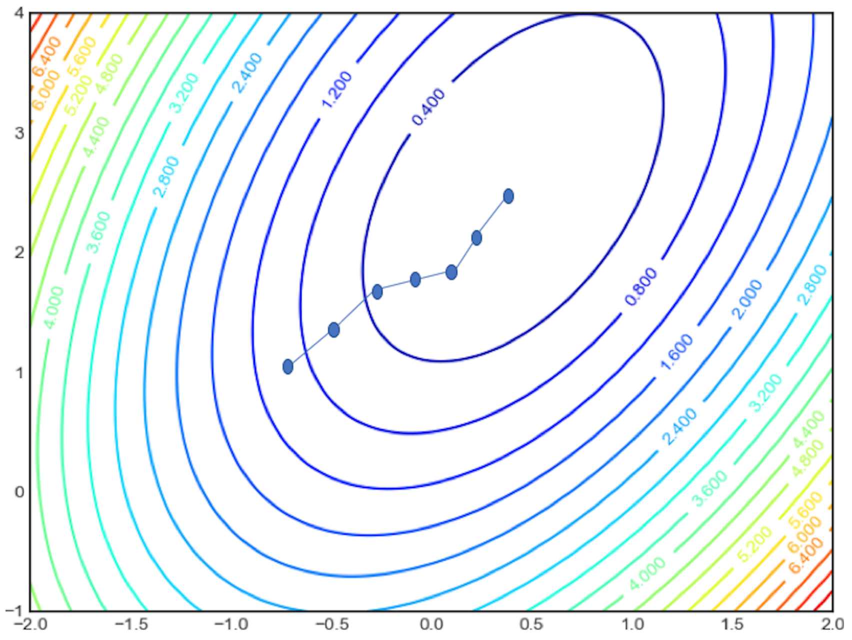


Figure 6.2: Rendering of 2D-Quadratic function (Discrete Actions)

6.1.4 Rosenbrock Function

Step

Observation space - Box space of (1,3) dimension It’s a continuous observation space of range (-1,-1,0) to (1,1,-18) . Input variable : ‘action’ from algorithm - Discrete value range of 1 to 16 Increment = [0.05,-0.05,0.1, -0.1,0.4,-0.4,1.6,-1.6] According to the action, X is added by the action indexed element of increment array. 1st two element takes value of X variable which the agent incremented wrt action and 3rd value is the negative cost function calculated for the incremented value X.

Reward negative cost function For the given input action negative cost function is calculated
Done Condition - Bool, True when the cost is less than certain threshold say 0.02
Info No info about debugging is created

Reset

Resets Weights and bias Resets Variable and make random predictions for first step and assigns to a variable X.

Render

Render function plots contour plot for each step TODO: put example contour plot

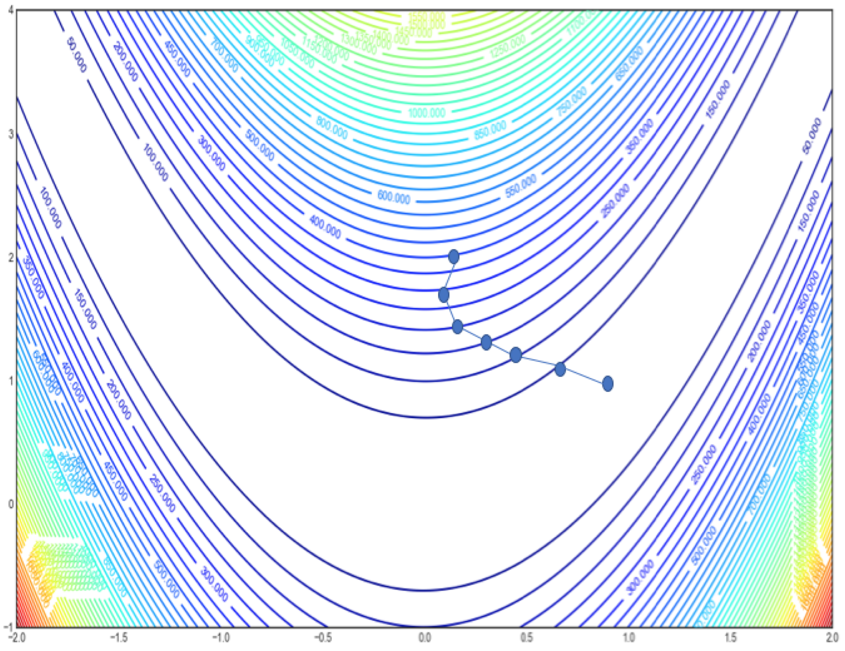


Figure 6.3: Rendering of Rosenbrock function (Discrete Actions)

6.1.5 Image Registration

Dataset

Maximum steps per episode – 30 Explain about Dataset The Dataset were acquired from cadavers. The dataset 2D Dataset images consist of fluoroscopic and CT images of the spine. There are images present in the dataset. The images are taken from the different region of the spine. Dataset consist of images that has the following regions. Lumbar, Head, Spine-thorax, Thoracic-cervical, Sacrum, Spine cervical , Spine lumbar, Spine lumbar thorax , Spine thorax cervical, Thoracic. Some of the above images also have metal implants. For each fluoroscopic image has acquisition image so that both images can be registered. In X-ray imaging, there are two acquisition modes, they are called “fluoro” and “acquisition”. Both of them are transmission X-ray images. The main difference is that the amount of radiation is much higher for acquisitions than for fluoro. Usually, fluoros are used continuously like a video during the intervention, and acquisitions are used only in highly important steps, such as during contrast agent injection or for documentation of treatment results.

Only the 3d images are CT or CT-like, so if you only do 2d registration they are not used.

Step

Input variable ‘action’ - Discrete Observation According to action, the moving image is shifted and gradient correlation is calculated If the index is negative/out of range, penalty is given

Reward if there is a penalty -> assigns to reward If no penalty reward -> GC ***Done*** No done condition (assigns False)

Reset

Reads the two images to be registered (one fixed and 1 moving) Calculate gradient correlation for the two images Return state value

Render

Checkerboard is created which tells interactively moves according to the action value.

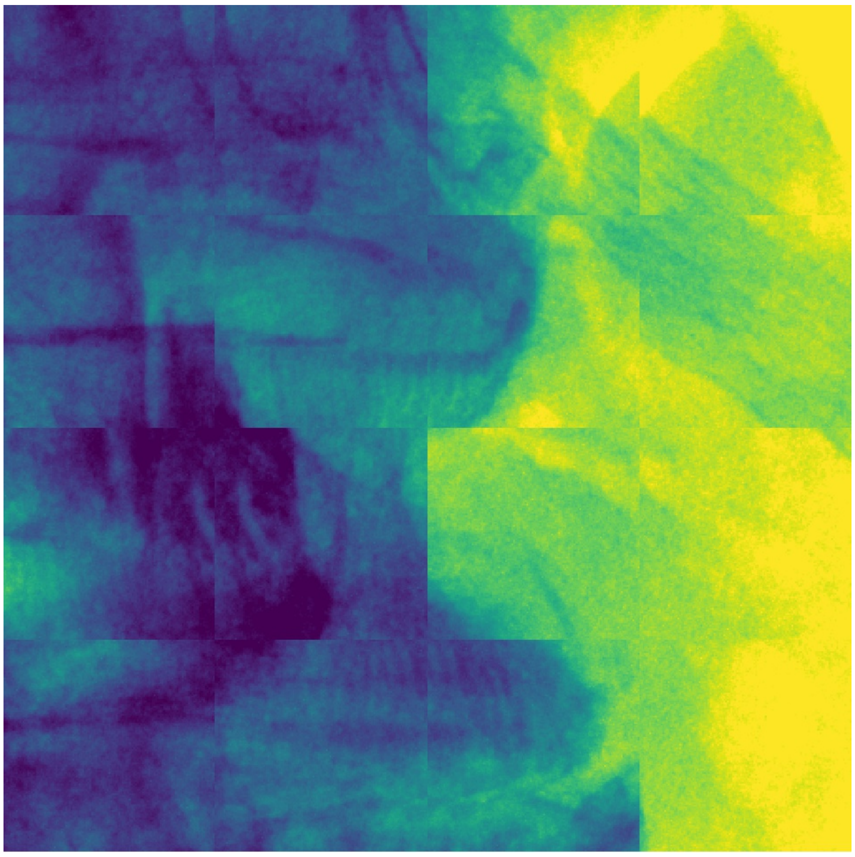


Figure 6.4: Rendering of 2D-2D image registration

7 Discussion

7.0.1 Learning Rate and Epsilon Decays

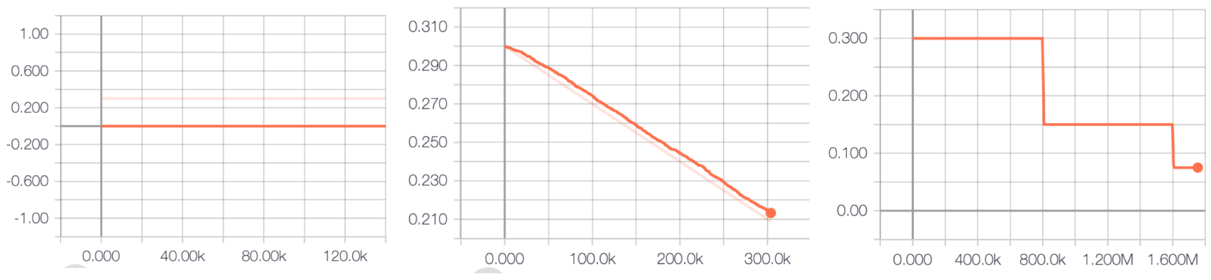


Figure 7.1: Types of Learning rate used in the experiments: constant, linear and step.

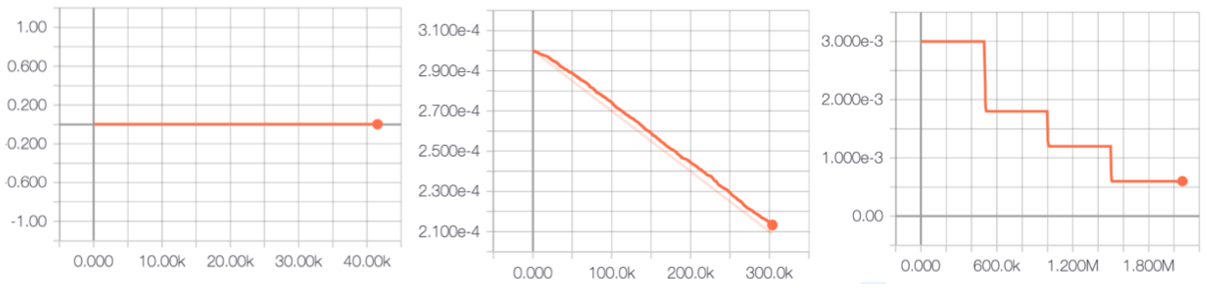


Figure 7.2: Types of Epsilon decays used in the experiments: constant, linear and step.

7.0.2 1D-Quadratic function (Continuous Actions)

7.0.3 1D-Quadratic function (Discrete Actions)

$$\begin{pmatrix} w \end{pmatrix} \begin{pmatrix} action \end{pmatrix} - \begin{pmatrix} y \end{pmatrix}$$

7.0.4 2D-Quadratic function (Discrete Actions)

$$\begin{pmatrix} w1 & w2 \\ w3 & w4 \end{pmatrix} \begin{pmatrix} action1 \\ action2 \end{pmatrix} - \begin{pmatrix} y1 \\ y2 \end{pmatrix}$$

7.0.5 Rosenbrock Function

$$(a-x)^2 + b(y-x^2)^2 \tag{7.1}$$

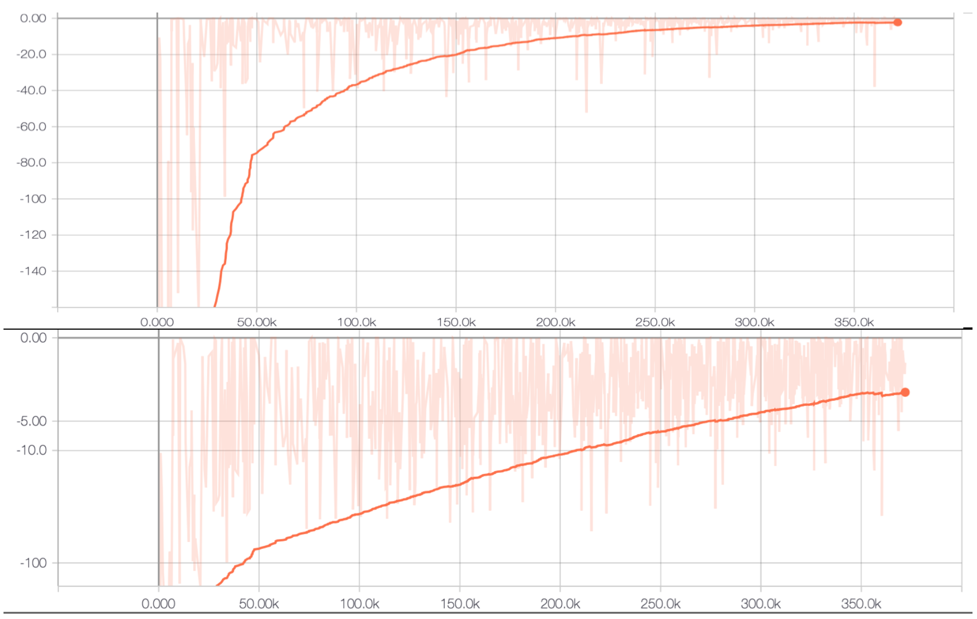
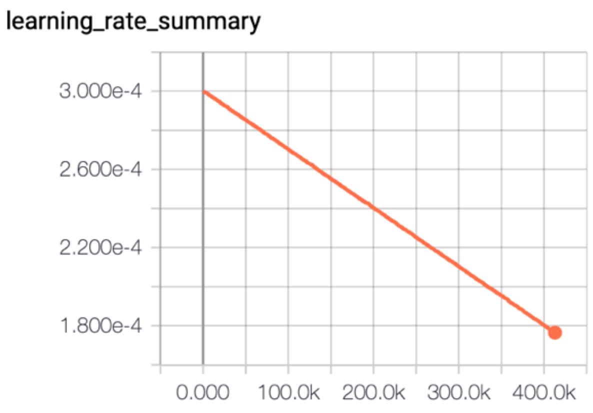


Figure 7.3: Reward Summary of Quadratic-1D(continuous actions) function: Normal scale[top] and log scale[dowm]



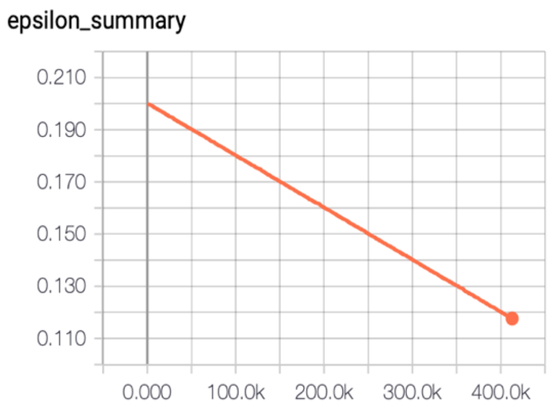


Figure 7.4: Linear learning rate and epsilon decays are used to train the network in this experiment

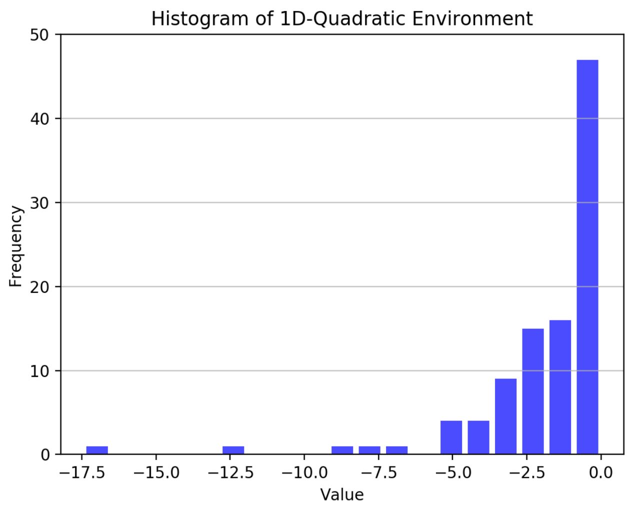


Figure 7.5: Histogram of 1D-Quadratic environment (continuous actions) using trained network

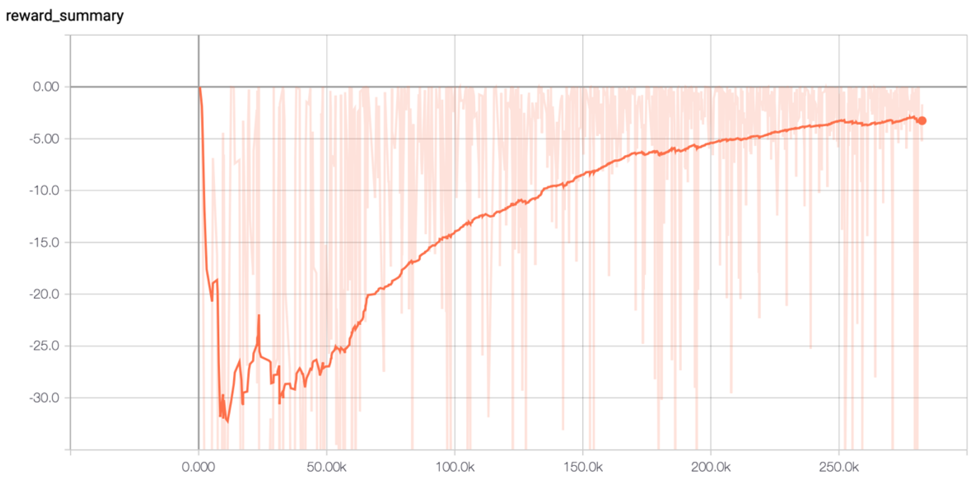


Figure 7.6: Reward Summary of Quadratic-1D(discrete actions) function.

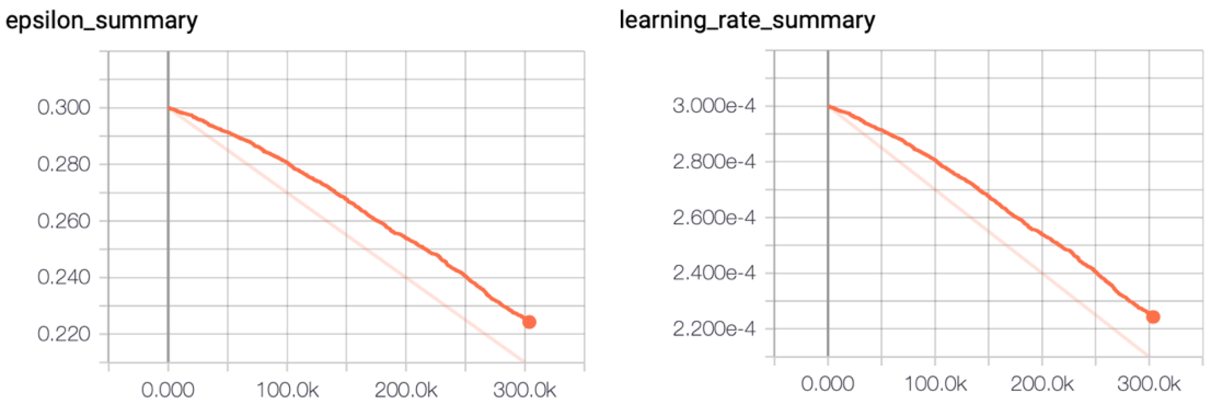


Figure 7.7: Linear learning rate and epsilon decays are used to train the network in this experiment

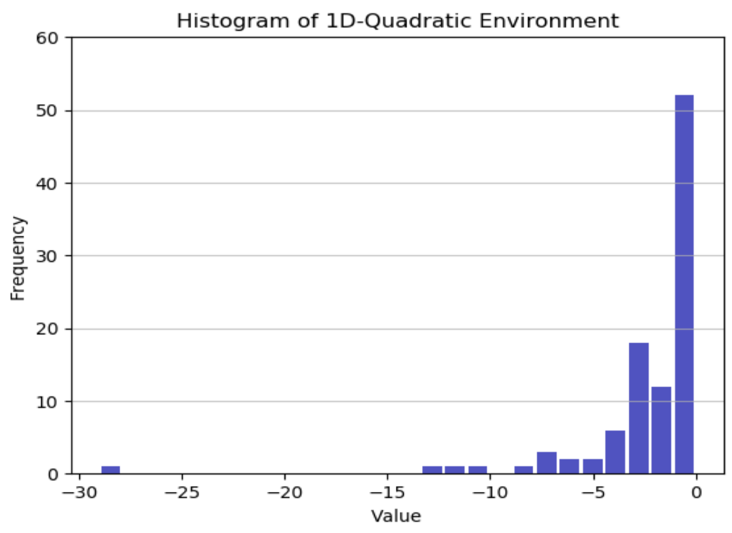


Figure 7.8: Histogram of 1D-Quadratic environment (discrete actions) using trained network.

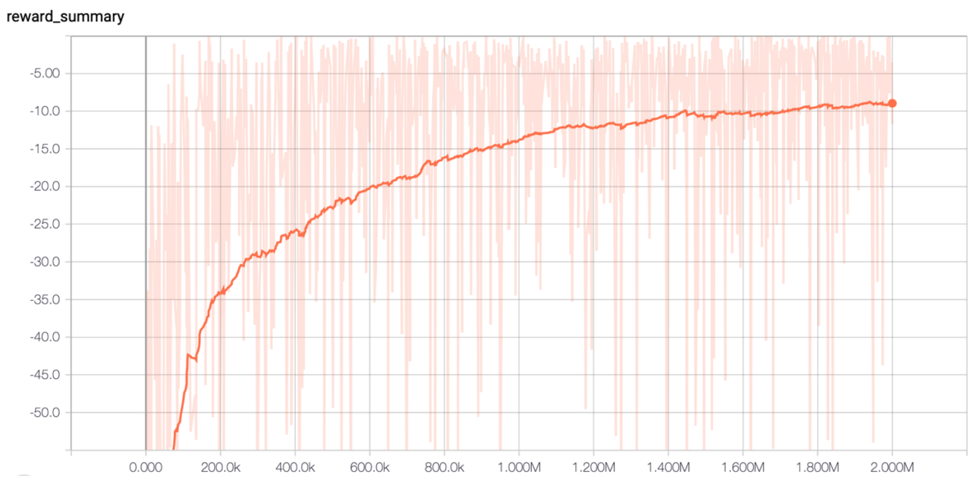


Figure 7.9: Reward Summary of Quadratic-2D (discrete actions) function.

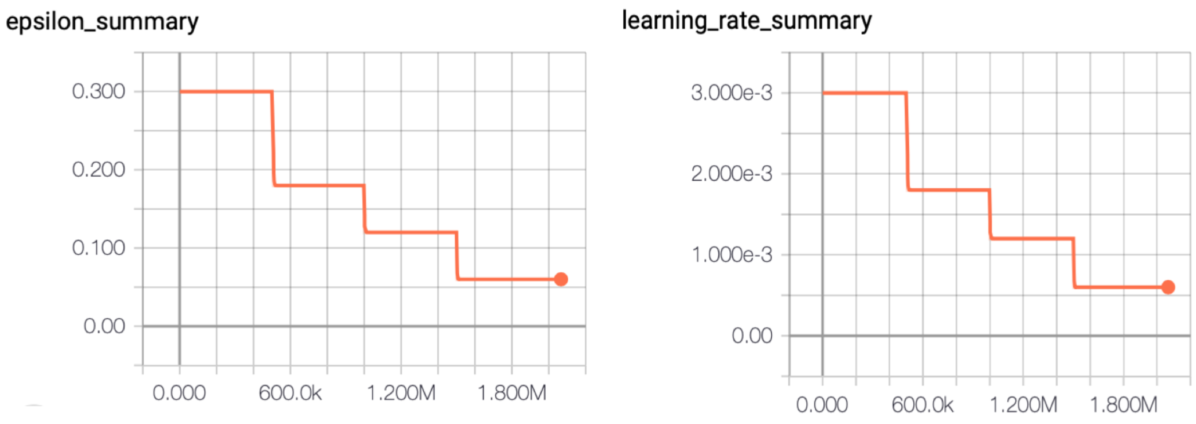


Figure 7.10: Step learning rate and epsilon decays are used to train the network in this experiment.

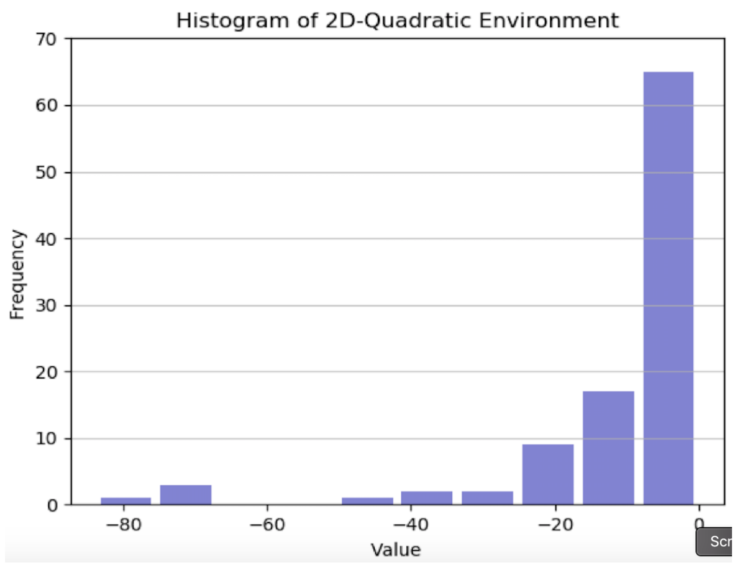


Figure 7.11: Histogram of 2D-Quadratic environment (discrete actions) using trained network.

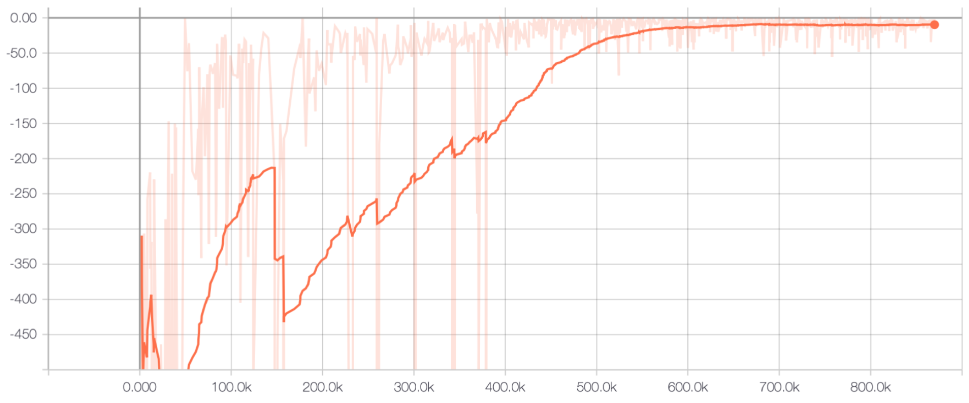


Figure 7.12: Reward Summary of Rosenbrock (discrete actions) function.

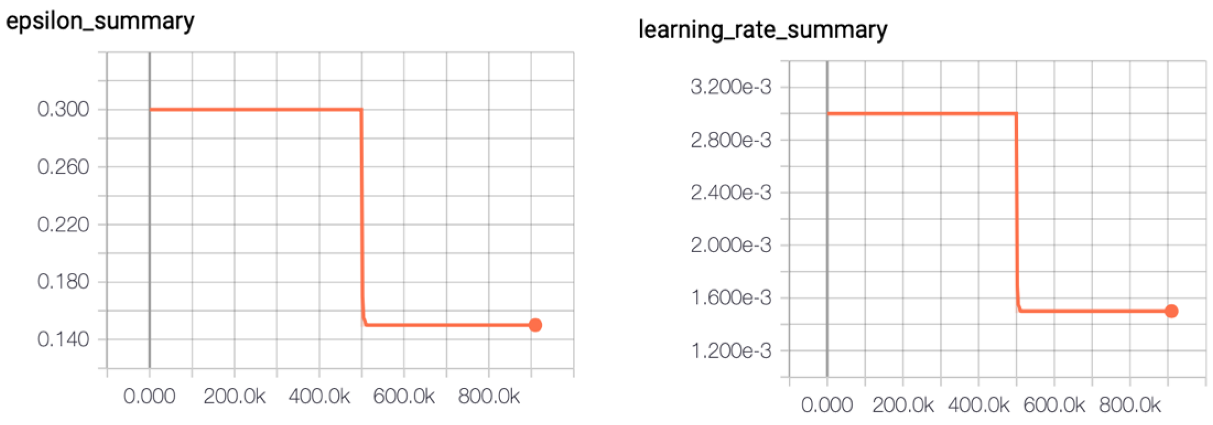


Figure 7.13: Step learning rate and epsilon decays are used to train the network in this experiment.

Table 7.1: 1D-Quadratic function (continuous Actions).

Hyperparameters	**Values**
Horizon	8096
Epoch	20
LSTM Unit	256
Minibatch	128
Clipped parameter (ϵ)	0.2
Discount (γ)	0.99
GAE (λ)	0.97
Number of actors	1
VF coef c1	0.1
Entropy coef c2	0.0
Epsilon Decay	linear
Gradient clip	0.5

Table 7.2: 1D-Quadratic function (Discrete Actions)

Hyperparameters	**Values**
Total step	500000
Horizon	2086
Epoch	10
LSTM Unit	256
Minibatch	64
Clipped parameter (ϵ)	0.3
Discount (γ)	0.99
GAE (λ)	0.97
Number of actors	1
VF coef c1	1.0
Entropy coef c2	0.0
Epsilon Decay	linear
Gradient clip	0.5

7.0.6 Image Registration

Table 7.3: 2D-Quadratic function (Discrete Actions)

Hyperparameters	**Values**
Total step	2000000
Horizon	2096
Epoch	5
Minibatch	64
LSTM Unit	256
Clipped parameter (ϵ)	0.3
Discount (γ)	0.98
GAE (λ)	0.95
Number of actors	1
VF coef c1	0.3
Entropy coef c2	0.0
Epsilon Decay	linear
Learning rate	0.003
Gradient clip	0.7

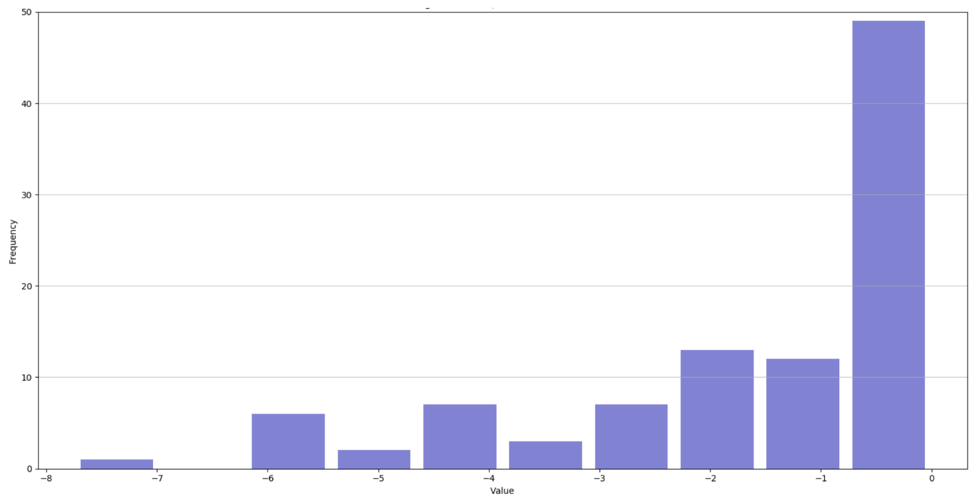


Figure 7.14: Histogram of Rosenbrock environment (discrete actions) using trained network.

Table 7.4: Rosenbrock Function

Hyperparameters	**Values**
Total step	2000000
Horizon	2086
Epoch	5
Minibatch	64
LSTM Unit	256
Clipped parameter (ϵ)	0.3
Discount (γ)	0.98
GAE (λ)	0.95
Number of actors	1
VF coef c1	0.3
Entropy coef c2	0.0
Learning rate	0.003
Epsilon Decay	step
Learning rate Decay	step
Gradient clip	0.7

Table 7.5: Image Registration

Hyperparameters	**Values**
Total step	2000000
Horizon	2086
Epoch	5
Minibatch	64
LSTM Unit	256
Clipped parameter (ϵ)	0.3
Discount (γ)	0.98
GAE (λ)	0.95
Number of actors	1
VF coef c1	0.3
Entropy coef c2	0.0
Learning rate	0.003
Epsilon Decay	step
Learning rate Decay	step
Gradient clip	0.7

8 Future

Bibliography

[1] Marcin Andrychowicz, Misha Denil, Sergio Gomez Colmenarejo, Matthew W. Hoffman, David Pfau, Tom Schaul, Brendan Shillingford, Nando de Freitas. *Learning to learn by gradient descent by gradient descent.* Google DeepMind, arXiv:1606.04474v2 [cs.NE],2016.

[2] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, Oleg Klimov. *Proximal Policy Optimization Algorithms.* OpenAI, arXiv:1707.06347v2 [cs.LG] ,2016.

[3] Yutian Chen Matthew, W. Hoffman, Sergio Gomez Colmenarejo, Misha Denil,Timothy P. Lillicrap, Nando de Freitas. *Learning to Learn for Global Optimization of Black Box Functions* DeepMind, arXiv:1611.03824v1 [stat.ML],2017.

[4] John Schulman. *Optimizing Expectations: From Deep Reinforcement Learning to Stochastic Computation Graphs* University of California at Berkeley, UCB/EECS-2016-217,2016.

[5] Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning: An Introduction.* The MIT Press, Cambridge, Massachusetts, 2018.

[6] Ian Goodfellow,Yoshua Bengio,Aaron Courville. *Deep Learning.* The MIT Press, Cambridge, Massachusetts, 2016.
`http://www.deeplearningbook.org`

[7] John Schulman, Sergey Levine, Philipp Moritz, Michael I. Jordan, Pieter Abbeel. *Trust Region Policy Optimization* OpenAI, arXiv:1502.05477v5 [cs.LG],2015.

[8] John Schulman, Philipp Moritz, Sergey Levine, Michael Jordan, Pieter Abbeel. *High-Dimensional Continuous Control Using Generalized Advantage Estimation* University of California at Berkeley, arXiv:1506.02438v6 [cs.LG],2015.

[9] Rui Liao,Li Zhang,Ying Sun,Shun Miao,Christophe Chefd'Hotel. *Registration Techniques* IEEE TRANSACTIONS ON MULTIMEDIA, VOL. 15, NO. 5, AUGUST 2013.

[10] Fakhre Alam, Sami Ur Rahman, Muhammad Hassan, Adnan Khalil. *challenges in medical image registration.* J Postgrad Med Inst 2017; 31:224-33.

[11] Colah: Understanding LSTM Networks,
`https://colah.github.io/posts/2015-08-Understanding-LSTMs/`

[12] OpenAI:Gym,
`https://gym.openai.com`

[13] OpenAI: Spinning Up,
`https://spinningup.openai.com/en/latest/`

[14] A Visual Guide to Evolution Strategies,
`http://blog.otoro.net/2017/10/29/visual-evolution-strategies/`