

Project Proposal



November 5, 2021

1 Overview

1.1 Problem Statement

Our project aims to solve a reinforcement learning (RL) problem on a real-world robot. Specifically, we will train an agent to move its arm to a particular location in a simulated environment, then attempt to transfer this pre-learned policy to a Universal Robotics 5 (UR5) robotic arm. Our main research question is: Can a policy learned in a simulated environment be generalized to a real-world setting, and if so, what is required for this generalization to work?

1.2 Policy Gradient Methods

Since the robot’s observations will be derived from sensor data (in our case, a camera), this problem will require an approximate solution method. Instead of learning values for each state-action pair, however, we will be learning the policy directly using *policy gradient methods*. In addition to the current state, this policy will depend on a parameter vector $\vec{\theta}$ as follows:

$$\pi(a \mid s, \vec{\theta}) = \mathbb{P}(A_t = a \mid S_t = s, \vec{\theta}_t = \vec{\theta})$$

We learn $\vec{\theta}$ through gradient descent on some performance metric $J(\vec{\theta})$. This metric is determined from the action-value function, which derived via the policy gradient theorem [2] to the following equation:

$$\nabla J(\theta) \propto \mathbb{E}_{\pi} \left(\sum_a q_{\pi}(S_t, a) \nabla \pi(a \mid S_t, \theta) \right) \quad (1)$$

Since robot actions will be binned into discrete movement patterns, we define $\vec{\theta}$ as the vector of confidence values assigned to each action. The optimal action is taken as the one with the largest confidence in $\vec{\theta}$, and our policy is determined by acting ϵ -greedily with respect to this action.

1.3 Convolutional Neural Networks

Artificial Neural Networks (ANNs) are networks of interconnected units that emulate the human brain. ANNs consist of an input layer, one or more hidden layers, and an output layer. The units in an ANN compute the weighted sum of inputs and apply a nonlinear activation function to the result to produce an output. Activation functions are typically S-shaped sigmoid functions. Each successive layer of a deep ANN computes an increasingly abstract representation of the network’s raw input, with each unit contributing to a hierarchical representation of the overall input-output function of the network [2].

ANNs usually learn using Stochastic Gradient Descent (SGD). SGD minimizes error on training data by adjusting weights after each example in the direction that would most reduce it. Since policy gradient methods assumes no set of labelled training examples, we use the function $J(\theta)$ defined in equation 1 as our error metric.

In order for the ANN to learn which nodes are responsible for what loss, it alternates forward and backward passes through the network. In the backward pass, it uses backpropagation to tune the weights of the network by feeding loss calculated at the output. The forward pass computes the activation of each unit given the current activation of the network’s input units. After each forward pass, a backward pass computes a partial derivative of each weight, as in stochastic gradient descent. Using the SGD optimization function, the weights should yield a smaller loss in the next iteration.

One specialized type of ANN is the convolutional neural network, CNN. This type of neural network is designed to process high dimensional data such as images or videos. Since our environment will be observed through a camera, this is the network we will be using in our project. CNNs consist of alternating convolutional and subsampling layers, followed by several fully connected layers [2]. Each convolutional layer slides a filter across the data and produces a feature map of convolved features. Each subsampling layer downsamples the convolved features, which is applied throughout the 3D volume. Finally, the fully connected layers at the end flatten the data so that the feature map is transformed into a single column for the output layer.

2 Related Works

2.1 Using Inaccurate Models in Reinforcement Learning

Training in the real world involves a significant challenge because there are things in games and simulation that aren't the same such as the realism of the objects as well as physics (friction, mass, velocity). The movement parameters are also different. Training in the real world is expensive and time consuming because each simulation will need to be done in real time and that could take days or even months to get a large enough data set for learning. This causes a scarcity in data and would mean that our trained model would not be as good as in simulation. Thus, an approximate model can be made in the simulation to assist in the learning in real life situations. With an optimal approximate model, it is possible to transfer the data onto the real world with high performance. (See Kober, J., Bagnell, J. A., & Peters, J. (2013)). This idea can be applied to our robot model by attempting to have similar tasks in the simulation and real world to increase realism and thus able to use the data from simulation to improve our policy on the real world UR5 arm. This brings us to the second major part where the information from the simulation can then be used in a convolutional neural network to train the real robot.[1]

2.2 Sutton Barto Policy Gradient Methods

REINFORCE, A Monte-Carlo Policy-Gradient Method (episodic)

```
Input: a differentiable policy parameterization  $\pi(a|s, \theta)$ 
Initialize policy parameter  $\theta \in \mathbb{R}^{d'}$ 
Repeat forever:
  Generate an episode  $S_0, A_0, R_1, \dots, S_{T-1}, A_{T-1}, R_T$ , following  $\pi(\cdot|\cdot, \theta)$ 
  For each step of the episode  $t = 0, \dots, T-1$ :
     $G \leftarrow$  return from step  $t$ 
     $\theta \leftarrow \theta + \alpha \gamma^t G \nabla_{\theta} \ln \pi(A_t|S_t, \theta)$ 
```

Figure 1: Pseudocode for Policy Gradient Method using Monte Carlo [2]

Sarsa (on-policy TD control) for estimating $Q \approx q_*$

```
Initialize  $Q(s, a)$ , for all  $s \in \mathcal{S}, a \in \mathcal{A}(s)$ , arbitrarily, and  $Q(\text{terminal-state}, \cdot) = 0$ 
Repeat (for each episode):
  Initialize  $S$ 
  Choose  $A$  from  $S$  using policy derived from  $Q$  (e.g.,  $\epsilon$ -greedy)
  Repeat (for each step of episode):
    Take action  $A$ , observe  $R, S'$ 
    Choose  $A'$  from  $S'$  using policy derived from  $Q$  (e.g.,  $\epsilon$ -greedy)
     $Q(S, A) \leftarrow Q(S, A) + \alpha [R + \gamma Q(S', A') - Q(S, A)]$ 
     $S \leftarrow S'; A \leftarrow A'$ 
  until  $S$  is terminal
```

Figure 2: Pseudocode for On-Policy Temporal Difference Learning algorithm [2]

3 Experimental Design

In general we will be using a policy gradient method with a CNN as a function approximator. The agent's goal will be to select a sequence of actions that leads its arm to a specific location in the environment marked by a red square. Specifically, an action is defined as a 5cm movement in one of the four directions: forward, backward, left, or right. This representation gives us a discrete action space and restricts arm movement to a single plane. At every time step, the CNN will take in a 640 x 480 pixel image overlooking the environment. This image is processed by the CNN using random-initialized weights for each layer, and a confidence value for each action will be produced as output. Ideally, the CNN will learn to discern the goal state and the position of the arm from the input pixels. To ensure that it does not 'memorize' a solution and ignore visual input, the red goal state will be randomized at the start of each new episode.

As described in the overview, the network will output confidence values for each action, and the policy is determined by sampling ϵ -greedily with respect to the highest-confidence action. We will start ϵ at a high value of 0.2 and slowly decrease it over time as the network converges, which will ensure that the agent explores sufficiently but also exploits the (locally) optimal policy when it is found. Each episode will continue until the goal is reached or the arm goes off the table. In both of these scenarios, the episode will end and the arm will be brought back to the initial position using a reset function. The agent receives 100 reward for reaching it's goal, -100 reward for going out of bounds, and -1 reward for any other move it makes. Once an action is taken, the loss function in equation 1 is calculated for all actions up to that point and that error is back-propagated. The weights of each layer are then updated according to stochastic gradient methods which will move them in the direction that decreases the loss. This will ideally cause or network to converge to an optimal policy over each time step.

3.1 Simulated Training

The learning process explained above will first be applied in a simulated environment. The simulated agent will include a 'camera', which outputs the robot's view of it's environment in the same way that a real-world camera would. Because of this, it is crucial to match the real-world environment as closely as possible in our simulation, as we will later attempt to transfer the weights learned by our CNN to the real-world setting. As far as it's implementation, the simulation will be based on the PyBullet library using OpenAI gym to handle the interface with our reinforcement learning algorithm.

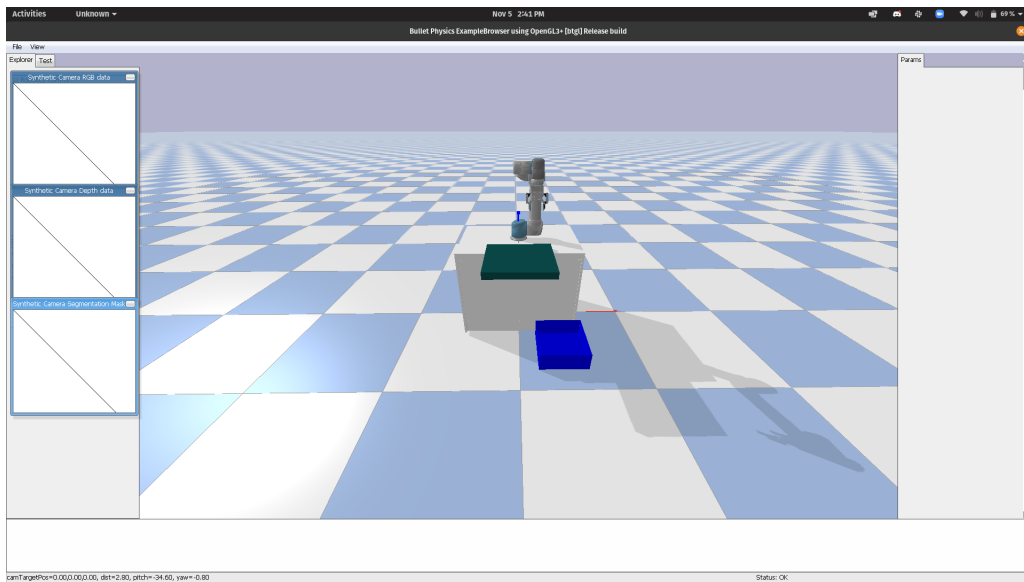


Figure 3: PyBullet UR5 Model

We currently have a baseline model already implement as shown in figure 3, although modifying this model to more closely match our real-world environment will be necessary. To do this, we will add a table,

adjust the lighting, and potentially add some textures to the objects within the simulated camera’s field of view. The goal state will be marked by a red square on the surface of the table, an example of which is shown in figure 4.

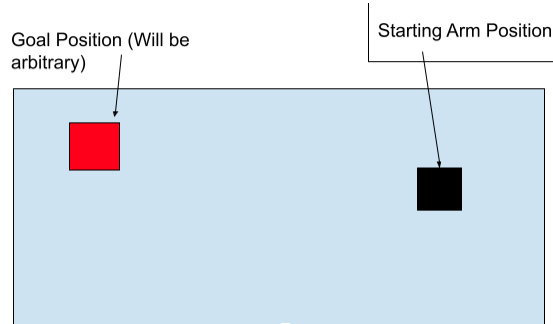


Figure 4: The figure above is the example environment.

3.2 Generalizing to a Robotic Arm

Once a (locally) optimal policy is achieved in the simulated environment, we will attempt to transfer it to a real-world UR5 robotic arm. We hope to achieve this transfer by using the simulation-trained network to interpret the pixels from the UR5’s real-world camera. Since it is unlikely that the learned weights will generalize perfectly, it will be trained until convergence on real-world input. To measure the success of this transfer process, we will perform a baseline experiment on the UR5 where the network is trained from scratch on the real-world camera input. Since real world runs can be time-costly, this experiment will be limited to 100 runs and performance will be extrapolated from the available data. This performance will be compared to the simulation trained network to determine if there is any advantage to one model over the other.

The arm itself consists of 6 degrees of freedom including a gripper, and will be controlled at a high level using PyBullet functions. At a lower level, the UR5 is programmed using MoveIt and Robot Operating System (ROS) via the `move_ee` function, which handles end effector movement. We will need to define four action functions (forward, backward, right, left) that correspond to the more abstract set of actions defined earlier in this section. The purpose of using these abstract movements instead of directly controlling the end effectors is to limit the action space. With a full set of robot actions, real-world learning would be infeasible in any reasonable time constraint.

4 Timeline

Nov. 12 : Start simulation environment
 Nov. 19 : Finish actions in simulation
 Nov. 26 : Start implementing policy-gradient algorithm
 Dec. 3 : Finish algorithm
 Dec. 10 : Transfer to real-world and finish training in simulation
 Dec. 17 : Testing
 Dec. 23 : Finish Testing/Write Report




5 Evaluation

We hypothesize that the simulation trained policy will achieve a faster learning rate on the robot when compared to the real-world only policy. To evaluate this claim, we will compare extrapolated metrics from the real-world only policy to those from the simulation trained policy. If we see that the simulation-trained policy is able to consistently achieve a larger reward sooner than the real-world only policy, we can be

certain that some of the simulation training was able to transfer to the real world environment. We will also compare these metrics to the simulation trained policy before running any additional training iterations on the real-world robot to help give us an idea of what direct benefit the transfer may provide. Ultimately, we define success as being able to transfer some of the simulation-learned policy to the real robot and show that it will reach an optimal policy faster than the baseline method.

6 Individual Responsibilities

While all team members will work together to complete each element of the project and no responsibility will be left solely to one member, we delegate a key responsibility to each member based on their interests and areas of expertise.

-  Responsible for overseeing the Python implementation process and understanding the documentation for any libraries that will be necessary.
-  Responsible for overseeing the implementation process for the simulated environment in Python, ROS, and MoveIt.
-  Responsible for coordinating experiments, keeping track of relevant experimental data, and evaluating results.

References

- [1] J. Kober, Bagnell A., and Peters J. Using inaccurate models in reinforcement learning. 2013.
- [2] R. Sutton and A. Barto. *Reinforcement Learning: An Introduction*. MIT Press, 2017.