ENM531 Project Report
Real Time Reinforcement Learning

Aslamah Rahman, Sonitabh Yadav

May 2019

Contents

1 Introduction

1.1 Overview

Reinforcement Learning is a class of deep learning where an agent learns how to perform in an environment. The learning is done by evaluating the possible courses of actions in a given state and choosing one which maximizes the reward, called the policy. The aim of this project is to build an autonomous bot that can follow a person around by taking in input data about the person's location using camera data and a distance sensor such as an ultrasonic sensor to build a 3D interpretation of the environment. Our work is based on the research published by Wang et.al. (35) and recently Tsun et.al. (33) on combining multiple sensor data to allow optimal path planning to move towards a target. Such bots have applications in a variety fields including healthcare, old-age assistance, agriculture and in aviation systems wherein the bot can be of assistance to the user as a companion or to carry a load.

As a second milestone of our project, we also hope to test the limitations of training and deploying deep learning models on edge processors or microcontrollers (MCU). Microcontrollers are low-cost tiny computational devices often employed in Internet of Things (IoT) devices for capturing sensor data and sending it to the cloud for further computation and action. The requirement of this data exchange between the cloud and the MCU can bring about significant latency, necessity of a connection, energy expenditure and privacy/security risks (20). However, MCUs are also severely constrained by their computing power and memory and is not capable of running existing deep learning libraries on them. This has lead to the development of libraries such as μTensor (1), CMSIS-NN (2) which currently runs on ARM Cortex-M processors. The need also lead companies such as Eta and Tensai to develop MCU with those microprocessors and companies such as Microsoft (3) and other research groups (26) to develop libraries with efficient deep learning algorithms. Since we will be using an MCU on the physical prototype of our autonomous vehicle, we hope to explore different programming paradigms to enable MCUs also to run deep learning models instead of communicating with a central computer for instructions and without employing costly processors on the edge.

1.2 Real-time Machine Learning

Real-time machine learning (RTML) is refers to proactively interpreting and learning from data that arrives in real time. It requires the acting agent to solve unfamiliar problems based on previous knowledge and often make decisions in real time. We discuss RTML in the context of development and deployment on embedded systems.

Currently, there are nearly 15 billion embedded processors active on the planet, collecting and sometimes processing huge amounts of data. Most of this data is discarded because these tiny computers often don't have the hardware capabilities to run complex interpretation task on them. The data is sent over wireless networks for processing to much more powerful centralized computers for this reason. But this form of cloud-based architecture wherein edge devices only perform data collection can be disadvantageous for the following reasons:

- Sending the data requires communication between edge devices and central

computer. This process is very energy intensive on the processor and hence is only done intermittently (4).

- Latency-guarantee trade-off while transferring the data is unavoidable because of the way wireless communication protocols are designed. This can introduce significant delays in taking critical actions from the data.
- Wireless communication protocols also have non-deterministic behaviour with respect to the time is can take for data packets to arrive from and to be received by the microprocessor, which are again highly undesirable for tasks that require critical responses.
- This non-deterministic communication link requires the microprocessor to be active almost all of the time, which is one of the major sources of idle power draw.

This is where the power of deep learning comes in. The largest of most intensive of operations in the implementation of a deep learning algorithm is matrix multiplications, for which hardware efficient implementations already exist and is entirely arithmetic-bound. Microprocessors and DSPs are able to process tens or hundreds of millions of calculations for under a milliwatt, even with existing technologies, and much more efficient low-energy accelerators are on the horizon. Once trained, these models are capable of running complex inference tasks and maintain robustness to noise, which is typical of real-time data. Making edge devices smarter by incorporating deep learning capabilities in them would solve all the issues originating from data exchange and make it an apt and reliable candidate for taking quick and consequential decisions from real world data. Devices can also make continuous improvements after they are deployed in the field.

The following are the some of the broader ways of how deep learning frameworks can be developed and run on resource constrained embedded systems:

1. Memory efficient implementation of existing algorithms
2. Low precision quantized training/inference (19)
3. Low precision multiplication during training (13)(36)
4. Compressing neural networks with pruning and Huffman coding (16)
5. Making floating point math highly efficient for AI hardware (5)

In this study, we focus on the first method and briefly touch upon the second method. To the best of our knowledge, this will be the first deep learning library developed to run on embedded systems with platform independency.

2 Review & Challenges

2.1 Computing Challenges

The computing challenges can be broadly split into two: hardware specific challenges and software specific challenges.

Microprocessors are extremely memory constrained devices and often have only 100s of kB of RAM. A regular fully connected deep neural network with a million parameters represented as 32-bit float would alone take up nearly 3.8 MB of storage space. This is mostly because they are designed to perform very minimal and repeated tasks such as sensor data collection and communication, as explained above. In addition to this, many do not support floating point precision. This is almost inevitable learning based for gradient descent as most of the errors backpropagated have miniscule values. Hence, the lack of floating precision is a bigger obstacle for accurate training than it is for inference, where active research in being performed in the field of quantized networks (19)(16)(13). Quantized networks, with implementation such as Binary Neural Network (14), XNOR-Net (27) etc., achieve memory compression by performing efficient mapping of floating points weights in the network to their low-precision 8-bit counterpart. An implementation of this has already been deployed by TensorFlow as TensorFlow Lite (6).

The software constraints are related to both how Real Time Operating Systems (RTOS) behave and with the design of memory efficient framework for extensive matrix manipulations. Since the data is collected in real-time it is expected to be noisy. Careful design of the systems architecture is also required to avoid avoidable latency in software runtime cycle. Since embedded systems, has to work reliably without the oversee or supervision once deployed, the task execution times are kept in check by means of a watch-dog timer which keeps track of this software runtime cycle and the code must follow the timer's requirements strictly. This can be challenging given the number of matrix operations that need to be performed in a single backpropagation iteration. Also, since performance has to be deterministic with respect to the memory available for performing operations, efficient data structure design is required to avoid memory overflow. An easy work around for avoiding memory overflow can be to dynamically allocate and deallocate memory as required. But this can lead to serious cases of memory fragmentation over time and make the system's execution less reliable. Besides, dynamic memory allocation exhibits non deterministic behaviour; the time taken to allocate memory may not be predictable which could also result in unexpected allocation failure. Lastly, since physical peripherals are often interfaced with these microprocessors, latency constrained by actual physical laws in actuating them is unavoidable. The software design must also handle this latency well without affecting other operations.

2.2 Physical Challenges

Challenges with respect to peripherals which are physically actuated to transition to a new state can also pose problems, especially in the context of real-world reinforcement learning tasks.

When in real physical systems, the training response might differ than in simulations do to the potential variabilities that are expected of real systems. Mahmood et.al (22) argue that choosing a cycle time for a particular task is not obvious, and the literature lacks guidelines or investigations of this task-setup element. Shorter cycle times may include superior policies with finer control. However, if changes in subsequent observation vectors with too short cycle times are not perceptible to the agent, the result is a learning problem that is hard or impossible for existing learning methods. Long cycle times may limit the set

of possible policies and the precision of control but may also make the learning problem easier. If the cycle time is prolonged too much, it may also start to impede learning rate by slowing down the data-collection rate.

Real environments may also an implicit momentum embedded in them depending on the dynamics of the system (18). Furthermore, it could be due to high sensitivity of the loss landscape with respect to small changes in the policy. That is, the loss landscape in certain environments might change significantly from episode to episode even with a minor change in the policy (e.g, environments where the agent might fall over and end the episode early). In such cases, momentum would not bear the same positive impact since the optimum might shift in a different direction. This would negatively affect gradient descent algorithms which seek faster convergence based by updating learning rate for learned momentum.

Also, in a real world scenario, a state change can be brought about by the following means: the actual state of the system changes or the state of the device measuring the state of the system changes. This will make teaching the learning algorithm robust to both kind of induced state changes essential without manifesting itself as noise in the system. These together makes it difficult to benchmark real world performance of reinforcement learning algorithms (23).

3 Implementation & Discussion

3.1 Problem Definition

We explore the field of real-time reinforcement learning by means building an end-to-end robotic system. This "person following robot" has the following capabilities:

- Person tracking: Using input from a camera, the robot will learn to track the position of a person, and execute panning/steering motion to keep the person in the center of its field of vision.
- Obstacle tracking: Using input from a distance sensor, the robot will learn to maintain a minimum distance with the person detected by the camera.
- The robot will repeat these tasks as the detected person moves around.

3.2 The Robot

The main actor in our project was the robot shown in Figure 1. Understanding some features of the robot is important to understand how our problem was broken down. The robot was designed on SolidWorks and laser cut and assembled from the Rapid Prototyping Lab.

3.2.1 Mechanical Design

The robot has 3 wheels – two at the back and one at the front. The two wheels at the back are run by the standard adafruit motor and are not capable of steering i.e. we won't be using differential drive to steer the robot. The robot is only steered by the front wheels which would be controlled by a servo motor that make the front wheel move either left or right. Since, the forward-backward

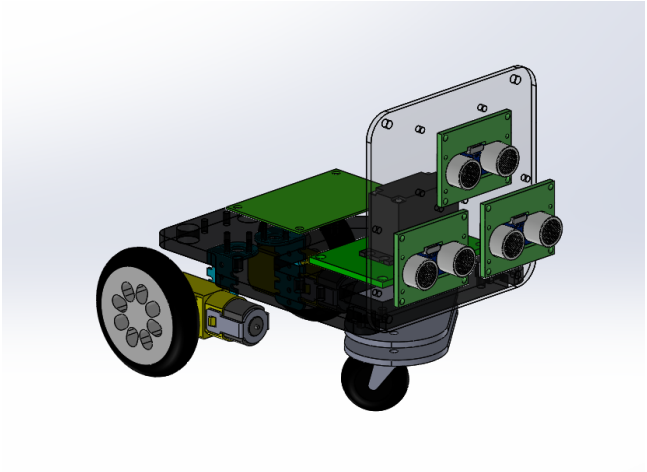

Figure 1: CAD design of the robot - SolidWorks

motion and the turning motion were independent from each other, the problem could easily be broken into two separate domains – moving left or right (Person Tracking) and moving forward and backward (Obstacle tracking).

3.2.2 Sensors

Since the problem is broken down into two segments, we could use a different set of sensors for the implementation of both the problems. For the person tracking implementation, the primary sensor used was the camera. For the obstacle tracking, a couple of ultrasonic sensors were used that detect the distance from the obstacle. The sensor input from the ultrasonic sensor could be used directly as the real time data because it outputs a single numerical value that indicated the distance from the obstacle. For the person tracking, the frame must be interpreted to create a quantifiable data point that can be captured in real time.

3.2.3 Parts

Here are the major parts we used in our robot, in case the reader would wish to replicate the project and the robot.

- ESP32 – The microcontroller used to control the motors and is the processing unit on the robot.
  Buy it here.
- H-Bridge - component used to interface the ESP32 control with the motors. One H-bridge interfaces with two motors.
  Buy it here.
- Servo Motor - Servo motor was used to steer the robot, hence only one was used at the front wheel.
  Buy it here.

- DC Motors - These were used to drive the back wheels. Hence, two of these are required to move the robot forward or backwards.
  Buy it here.
- Body - the body was made with the use of acrylic sheets (thickness - 1/8"). The design may be variable depending how the sensors need to be mounted.
- Ultrasonic sensor - These sensors were used to detect the distance from the obstacle in front. Two sensors were used and the difference in the values were taken to ensure the robot is always aligned with the obstacle.
  Buy it Here.
- Front wheel - A castor wheel was used in the front to steer the robot. Its motion was restricted to stop the swivel and only face in the desired direction.
  Buy it Here.
- Back wheels - We used the adafruit wheels at the back for our implementation because it goes well with the adafruit motor that we were using. But any compatible wheels may be used.
  Buy them Here.

3.3 Person Tracking

Our problem statement for this project is broken into two different components – Person tracking (using the camera) and the Obstacle tracking (using the ultrasonic sensor). This section deals with the implementation of the Person tracking which controls how the robot learns to turn looking at the camera input. There are multiple ways to implement these and many of them are relatively uncomplicated, but we wanted to implement the Real Time learning on board the robot, which proved out to be a challenge.

3.3.1 Face Detection

Since the camera outputs frames of images, we initially want to convert the input to some quantifiable value which can then be used to train the turning of the robot in real time. Hence the robot was setup to detect the face of the person. The system setup was detection and not recognition. Hence the system would recognize all faces and not a particular face.

There are multiple CNN based algorithms (12)(37) that can be used for the detection/recognition of faces but we wanted a light weight implementation of the same. Since recognition is not our primary goal, we would ideally want the least amount of processing resources to be taken up by the detection. Since, most CNN based models involve learning multiple feature as well as the weights between different layers, they tend to take up a longer time. Even if the trained weights are used, it leads to a lot of matrix multiplications and computations. Considering that it needs to be done for each loop, while capturing each frame, we need all the speed we can get.

Hence, the face detection algorithm that we chose was presented in a paper in 2001(34). Its is quite famous for not being computationally expensive because

it uses the predefined feature set and the model does not learn anything. These features are called Haar Cascades and act on the image data like a normal Kernel would. They are essentially filters and they go over the image to detect various features. Some of the filters are shown in Figure 2. The weights between theses filters are manually set depending on the kind to object we are hoping to detect. Hence, the model does not even has to learn weights and detect right away.

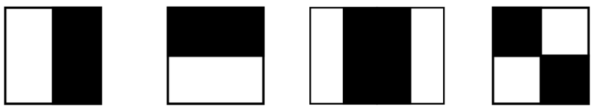

Figure 2: Filters for Haar cascades - (left to right) vertical line detection, horizontal line detection, rectangular feature detection and slant line detection

Apart from the preset weights and filters, the only parameter that needs to be set is the Scaling factor. The scaling factor determines how finely the filters go over the image data. Essentially, the image is scaled by this factor before the filters are run over them. Hence, or a big scaling factor, the filters would go over a smaller part of the image. This is especially useful if the face being detected appears smaller in the image i.e. the person is far away. However, this may lead to detecting faces at parts of images where the faces do not exist. This is because each filter may detect features is smaller parts of the images and may conclude that an face exists where it does not exists. The correct scaling factor (1.3) is shown in Figure 3 and the incorrect scaling factor for the same image (1.05) is shown in Figure 4. Please not the image cannot be scaled down, hence the minimum value of the scaling factor is 1.

Figure 3: Face detection - image scaling factor = 1.3

Summary of advantages and disadvantages:

- Advantages:
  - No training needed, only the weights of the haar features should be determined

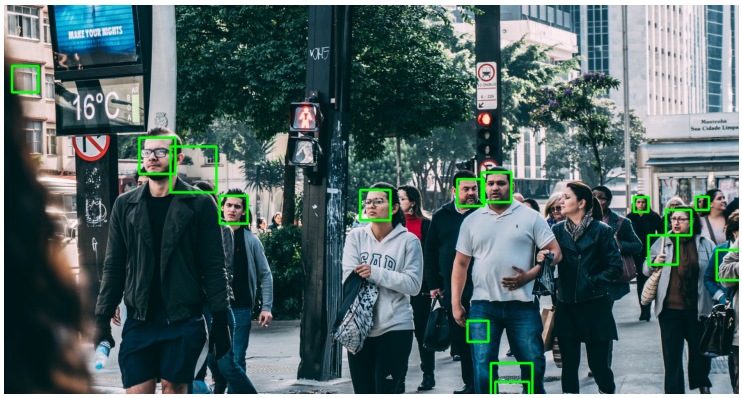

Figure 4: Face detection - image scaling factor = 1.05

  - Higher execution speeds, involves minimal computations

- Disadvantages:
  - Can be implemented only for Face Detection, and not Recognition
  - Not as robust as standard face detection systems using CNNs.
  - Needs a little tuning, accuracy of detection may be weak for a small scaling factor
  - May fail if parts of the face is hidden.

Code implementation - To implement this algorithm, we used the cv2 module in python which is the OpenCV library for python. The frames used the pretuned haar cascade xml feature file. For each frame, the face is detected and shown on a window. The center of the image is marked with the green dot. The red dot marks the center of the face, and the blue dot marks the position of the red dot along the center of the image. This was done to be able to set up and environment for further algorithms such that we can try to make the blue dot to try to match the green dot. So the robot always tries to get the face to be at the center of the frame. Example of a frame is shown in Figure 5.

Name of the code - Face-detection.py

3.3.2 Q-learning

The algorithm we chose for this project is sub set of Reinforcement learning(RL), called Q-learning. An RL implementation has 5 basic components -

- Environment - A setup inside which the agent takes action to change its state.
- Agent - Its an entity that takes an action inside the environment.
- Action - Something that the agent does to change its state inside the environment.
- States - different states in which the agent exists in an environment.

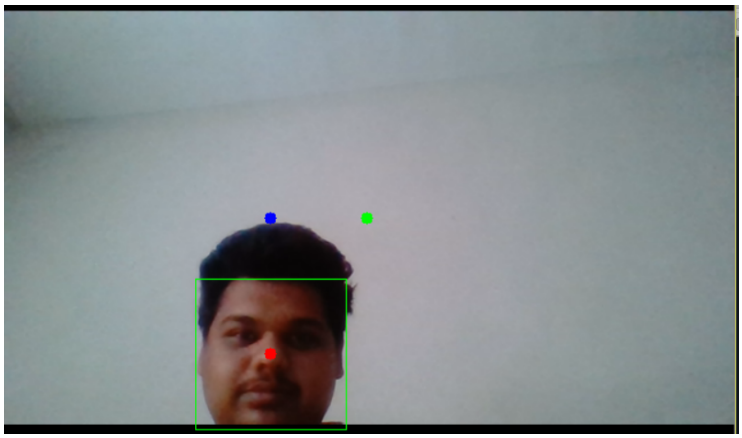

Figure 5: Example frame for face detection

- Reward - Often a numerical reward that agent get when it takes a favourable action. The aim for the agent is to maximize this reward over training iterations.

A brief summary of RL has been shown in Figure 6. There are lot of resources online(7)(8) to learn about RL. We won't be able to explain everything in this report but we will explore Q-learning and its variants in detail.

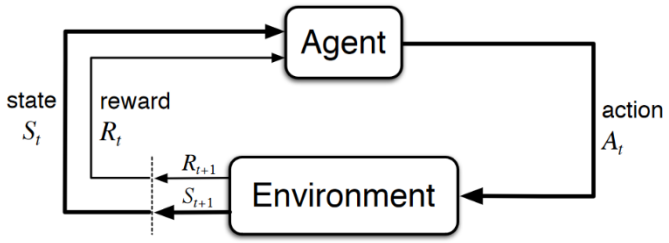

Figure 6: Reinforcement learning summary

The variant of RL we use is called Q-learning. Q-learning formulates something known as the Q-table. A Q-table store Q-value for each of the state-action pair. Hence, if a setup has 3 actions and 4 states, a Q-table is a matrix of (4X3) dimension. The Q-table is initialized to zero values and the values keep updating as the training happens. The Q-values are calculated using the Bellman equation that we will discuss more in detail. The Bellman equation for Q learning has been shown in Equation1.

$$Q^{new}(s_t, a_t) \leftarrow (1-\alpha)\underbrace{Q(s_t, a_t)}_{old\ value} + \underbrace{\alpha}_{learning\ rate}\left(\underbrace{r_t}_{reward} + \underbrace{\gamma}_{discount\ factor}\underbrace{max_a Q(s_{t+1}, a)}_{estimate\ of\ optimal\ future\ value}\right) \quad (1)$$

In the equation, α represents the learning rate. The learning rate is the same as we have had in the class in the various optimizers that we have used. It represents the size of every step while optimizing the function. $Q(s_t, a_t)$ represent the Q value for the state and action that can be taken in the current time step. γ represent the discount rate. Hence, γ is used to account for the future reward as well. It is multiplied to $max_a Q(s_{t+1}, a)$ which represents the maximum Q value out of all actions from the future state. Hence, it accounts for future rewards as well. So, if the agent has to take the action, it not only accounts for the maximum reward it gets from the current state, but also considers the rewards from the future state. So it may maximize the overall reward even if it does not choose the maximum reward action from the current state. Note the γ is always less than 1, because we do not want the weightage of a future reward to be greater than our current reward. We would still like the reward from out current state to be of maximum priority. Some of the math regarding the RL policy math, Bellman equation and the Q-learning equation mathematics have been discussed in detail in some references provided(9).

The Q learning training algorithm has been summarized in the Figure **??**.

Hence, we initialize the Q-table to zero, and the training takes place in episodes.The agent tries to maximize its reward over the episode. The episode ends when either a predefined number of steps have been reached, or a terminal state is reached. A terminal state is user defined and it happens either when the goal is reached or the agent cannot continue. The policy discussed here is the ϵ-greedy policy.

ϵ-greedy policy - This policy is used to balance exploration of the environment vs the exploitation of the rewards. The steps are as follows-

- The value of ϵ initialized to 1, with a decay rate (exponential decay)
- For every step, a random value r is chosen between 0 and 1
- If $\epsilon > r$, a random step is taken - exploration
- If $\epsilon < r$, the action with the maximum Q-value from a particular state is chosen - exploitation

This is done to counter the problem of multiple positive rewards in an environment. If an agent discovers a smaller reward in the initial steps, it may simply exploit that reward again and again to maximize total rewards. Hence its important to randomly explore the environment so some bigger rewards can be explored. However, always exploring the environment might make it difficult to maximize rewards over training.

When the training is over, a final Q-table is formed for each state-action pair.The agent only needs to pick the action with the maximum Q-value corresponding to that state.

3.3.3 Setup

Our Environment - The environment we chose for Obstacle Tracking has been described before. The face is detected and the distance of the face from the center of the frame along the middle plane of the frame is calculated. They are represented in with green and blue dots, respectively in Figure 5.

Our Agent - The agent we use for our case the servo motor controlling the turn of the robot. The servo takes an action and moves the camera mounted on the front towards the center of the face (attempts to match the blue and the green dot).

Actions - Since the camera can move only left or right, we only have a set of 2 action - move left or move right.

States - The state has been defined as the position of the blue dot with respect to the green dot. Since our image size was (640X480), the total possible number of states for the blue dot be 640. In order to reduce the high dimensionality of the states, we decided to half the number of states by considering every set of 2 pixels together. Hence, the final number of states were 320.

Rewards - The rewards were chosen to be linearly varying from the center of the frame (green dot) to the edge of the frame.

Q learning algorithms typically work very well when the Q-matrix is small i.e. the number of states and action are relatively small. In our case the model did not train properly and we kept getting unstable Q-values. The reasons were the following -

- In order for Q-learning to work, we needed the blue dot to be relatively stable with respect to the green dot. The only way to ensure this happened was to make sure that the training happened for each captured frame and the Q-table was outputted. Since we had a live camera feed with continuously changing frame, a proper training could not happen.
- High dimensional state space - although we reduced our state space to 320 states, the Q-table was still 320x2 and had 640 elements. The high number of elements make the Q learning algorithm unstable. Combined the reasons discussed above, training was impossible.

Hence we chose to implement a further variant of the Q-learning algorithm known as the Deep Q-learning. However, one great resource to see a successful implementation of a simple Q-learning algorithm is the Frozen Lake example (10).

Since the implementation for our use case was not successful, we implemented the Frozen Lake problem as an example of Q-learning.

Name of the code - Frozen-Lake.py

3.3.4 Deep Q-Networks

Deep Q-network is a combination of Q-learning and Deep neural networks. The major difference is instead of having a well defined function to approximate the Q-values, the Q-function is now approximated by Neural Network.

The Neural network accepts states from the given frame as an input.For the given state, the network outputs all the estimated Q-values for each action corresponding to that state. The objective of this network is to approximate the $Q(s_t, a_t)$ given by the Bellman equation, given in the Equation 2

The objective of this network is to approximate the optimal Q-function. Hence the loss for the network becomes the mean squared error between the outputted value and the optimal value from the RHS of the Bellman equation. Further, it uses standard stochastic gradient descent methods to minimize the loss.

$$Q(s,a) = \mathbb{E}\Big[R_{t+1} + \gamma max_a Q(s',a')\Big] \tag{2}$$

The only major advantage of using a neural network to approximate the Q-function is that a Neural network does well when the input is high dimensional data. It helps us approximate a function instead to continually updating a look up table as in Q-learning. The training was attempted but was not successful using a DQN as well. The first problem in discussed in the section **??** still persisted despite the second problem being solved. The following reasons were identified:

- Again, we needed the blue dot to be relatively stable with respect to the green dot. The only way to ensure this happened was to make sure that the training happened for each captured frame. This was too expensive considering we wanted a real time system.
- We naturally wanted to train the network with the continuous camera input. However, the training was difficult because the relative position between the blue dot and the green dot were not only dependent on the agent's action, but it was also dependent on the swift movement made by target person.
- Computation for a single system took a longer time than before because a neural network was involved and standard training time for a single frame was about 30 seconds on an Nvidia GEForce GTX 1060. This was not ideal for our Real Time implementation.

3.3.5 Identified Solutions

Since, we started to drift out of the real time domain in our attempt to have a live training on the robot with the above 2 algorithms, we had to try to find some solutions to the problem. We identified two possible solution:

- **Deep Q-learning using CNN:** This solution is a step further than the DQN discussed in the previous sections.Hence, it was going to be even slower. However, it had a major advantage that the training did not need to be done for every single frame. Hence, it was possible to do it with a live camera feed, and it may reduce the time of implementation because it would reduce the total number of training instances.
- **Continuous action controller**: This solution was identified as a simple implementation given the Real time nature of out implementation. There would not be any parameter training involved and it would just do follow the face using simple movements.

We will discuss our solutions in detail in the further sections.

3.3.6 CNN Based DQN

The major issues that we faced in deploying the previous 2 algorithms was that either the training needed to be done for each frame, and that pushed it out of the Real time domain. Or the training when the camera is live, the way

we defined our stated fed values to the system that confused it because of the misalignment in action and space, and we got random values in return.

The first benefit that a CNN based DQN offers is the explicit definition of a state is not required. The input to the Convolutional Neural Network is the frames that the camera sees. Hence, it determines what the states are and what the terminal states should be depending on which frame gives it a higher reward. The is revolutionary work is often used in teaching computers to play games.The first instance of this was recorded by a company called DeepMind based in London where they taught a video game to a computer by feeding the game screen to a CNN based DQN (25).

The algorithm is quite novel and needed time for implementation. However, we knew that it would be out of the Real time domain. The training time for one episode for some of the simplest implementations with 50000 steps was about 30 minutes. Hence, we decided to drop implementing DQN CNN based training. But it would make for a very interesting implementation for future projects because it would take some time. This is revolutionary work that led to the Google acquiring DeepMind in 2014. This collaboration later led to the development of AlphaGo (31), which was an AI player who beat the Go world champion Lee Sedol in a challenge match 4 games to 1(15).

Since, the major reason to dismiss the implementation of DQN CNN was time, we looked at some of the recent attempts to alleviate the problem by the EECS Department at the University of Michigan (15), who are also working on this problem. However, it still was out of real time domain.

3.3.7 Continuous action controller

This was our chosen solution for this problem of physically tracking the person through the robot. This a controller that will simply react to the relative positions of the blue and the green dot in an attempt to follow the face. The positions of the dots will be outputted from the face detection function and the code will determine which direction to move in.

Name of the Code - Cont-act.py

3.3.8 Results

Th goal of this section of Person tracking was for the robot to be able to follow a person's face and track them. It was implemented in three stages -

- Building the robot - The robot was successfully designed and build as described in the section 3.2.1. The building and assembly of all the sensors and actuators was done using the Rapid Prototyping Lab at the University of Pennsylvania.
- Face Detection - Successfully implemented using Haar Cascades instead of CNN base detection system for the sake of speed and real time implementation.
- Real Time Learning on the robot.
  - Q-learning - The implementation of real time Q-learning failed primarily because of highly random motion of the target. Secondly, because of being unstable at high dimensional state space.

- Deep Q-learning - The implementation of DQN helped us alleviate the problem of high dimensional states but the high random movement of the target made the system impossible to train. Another reason of failure was that it was difficult to model and train the system of explicit states.
- DQN-CNN - This is identified as the primary solution for the implementation of Q-learning when there is a problem with explicit definition of states. The input for CNN is just the Image containing different states, and the current states and the maximum reward state is implied by the system. However, this is a novel architecture that is out of the Real Time implementation was an issue since it takes a longer time to train. Hence, the model was discarded.
- Continuous action controller - A simple Continuous action controller was successfully implemented for the system to be able to work in Real time. The physical implementation could not be performed in the given time frame, as hardware takes time to assemble.

Although the correct implementation of learning on the robot could not be done, it leaves a great problem for solving future project where we may try to apply a real time CNN-DQN, when our environment and problem set is better defined. The project led to the introduction to the paradigm of Reinforcement learning, understanding Q-learning and its variants through various examples, and attempting to apply this to a real time system. The project was insightful as involved everything from building the robot, assembling electronics, sensors, actuators and implementing the code.

3.4 Obstacle Tracking

3.4.1 Q-learning

For the problem of obstacle tracking, our environment consists of the agent which is the robot and the state s_t at time t is the distance to the nearest obstacle measured using a proximity sensor. The possible action a_t that can be taken at a time t includes moving forward, moving backward or staying in position. The goal g is the distance that is to be maintained with the obstacle and the reward r_t is calculated to be representative of a desirable state transition to s_{t+1} and for approaching the goal as in Equation 3.

$$r_t = \alpha(s_t - s_{t+1})(s_{t+1} - g) \tag{3}$$

Where, α is a reward amplifying/normalizing constant. The reward function ensures that if the agent is far from the goal, moving towards the goal will be positively rewarded with proportion to its proximity to the goal. Also, if the goal line has been crossed, moving away from the obstacle will be positively rewarded. Any action taken after the reaching the goal will have a zero reward reward prompting the agent to maintain position.

We started our attempt to model the learning process using the simplest of reinforcement learning algorithms, the Q Learning model. The algorithm is characterized by means of a Q table, which is a matrix of possible future rewards for a given action taken at a given state. The goal of the algorithm is

to learn to take suitable actions which would maximize the Q value for a given state. The learning process in the algorithm is described in Section **??**. But since Q tables requires state-action spaces to be discrete, this largely limited the flexibility with which we can fine tune the performance of the agent. One way of working around the problem of discreteness would be to create a Q table with all possible states to give an illusion of continuity in states. But this approach would take a huge hit on the memory constraints we have with respect to using microprocessors for computation and would also take a long time for all Q values to be learned reliably. Another limitation with the deep learning implementation of Q learning, known as Deep Q Networks (DQN) is that both the network output and target value is based on the same set of parameters that we are training for, as illustrated in Equation 4(25).

$$\Delta\theta = \alpha\Big(\underbrace{(r_t - \gamma max_a Q(s_{t+1}, a_t, \theta))}_{target} - \underbrace{Q(s_t, a_t, \theta)}_{prediction}\Big)\nabla_\theta Q(s_{t+1}, a_t, \theta) \quad (4)$$

This implies that as the parameters are optimized to get closer to the target, the target also moves, making the training process highly unstable. The policy is also hard-coded in the form of the ϵ-greedy strategy, which may be a poor representation of the actual policy that the agent needs to follow for optimal action in the environment. ϵ-greedy strategy can also lead to a case similar to overfitting wherein the agent will continuously take the same action with slightly higher Q value than the rest for a given state and not explore the other actions enough. This kind of reinforcement is not desirable. Although these problems has been tackled using tricks such as reducing the frequency of update of Q table(21), implementing different networks for target and prediction(17), replaying experiences(28) etc., due to several other constraints cited above as well, we moved our implementation to another algorithm based on policy gradients which suits our problem better.

3.4.2 Policy Gradient Network

Policy gradients networks takes in the state of the system and outputs the probability distribution over all allowed actions for a given state instead of learning the value function which calculated the expected reward for an action given a state(30)(29). The algorithm tries to optimize the policy $\pi(a_t|s_t)$ instead of the value function $Q(s_t, s_t, s_{t+1})$. Policy based methods have better convergence properties since the situation of moving target does not arise. Also, since the network is capable of taking in continuous states and outputting continuous actions, they are much more effective in high-dimensional environments(30). Lastly, policy gradient can learn a stochastic policy, while value functions can't. This has two consequences. A stochastic policy allows our agent to explore the state space without always taking the same action. This is because it outputs a probability distribution over actions. As a consequence, it handles the exploration/exploitation trade off without hard coding it. We also get rid of the problem of perceptual aliasing. Perceptual aliasing is when we have two states that seem to be (or actually are) the same, but need different actions.

Using policy gradients for learning allows us to make the following modifications to our environment's definition:

- The input states no longer need to be binned and discretized. The direct input from the proximity sensor can be used as input.
- Instead of restricting the action space to moving forward, moving backward and staying in position, we can further extend this to include a component of speed as well. This can enable the the agent to learn to move quickly towards the goal and appropriately adjust speed as it nears the goal to prevent missing it.

While training the network, the objective of the algorithm is to maximize the expected value of score function described in Equation 5.

$$J(\theta) = \mathbb{E}_{\pi\theta}\Big[\sum \gamma r\Big] \tag{5}$$

Where, θ is the parameters of the network, γ is the reward discount factor and r is the reward earned. The training in implemented using REINFORCE algorithm(32), as explained below. Sampling has been performed using inverse transform sampling.

while Training **do**
  for Episode $t = 1$ *to* T **do**
    Sample (s_t, a_t, r_t)
  end for
  $\theta = \theta + \alpha \sum_{t=1}^{T} [\nabla_\theta \log \pi(s_t, a_t, \theta)] R(\tau)$
end while

Where, R_τ is the average value of discounted reward for each episode, calculated as given in Equation 6

$$R_1 = r_1 \tag{6}$$

$$R_t = r_t + \gamma R_{t-1}, t = 2 \text{ to } T \tag{7}$$

$$R(\tau) = \frac{1}{T} \sum_{t=1}^{T} R_t \tag{8}$$

Taking a step with this gradient pushes up the log-probabilities of each action in proportion to R_τ. Agents should really only reinforce actions on the basis of their consequences. Rewards obtained before taking an action have no bearing on how good that action was, only rewards that come after do. For this, we implement Reward-To-Go policy(11) and modify the update rule as in Equation 9.

$$\theta = \theta + \alpha \sum_{t=0}^{T} [\nabla_\theta \log \pi(s_t, a_t, \theta)] \sum_{t'=t}^{T} [R(s_{t'}, a_{t'}, s_{t'+1})] \tag{9}$$

A key problem with policy gradients is how many sample trajectories are needed to get a low-variance sample estimate for them. The formula we started with included terms for reinforcing actions proportional to past rewards, all of which had zero mean, but nonzero variance: as a result, they would just add noise to sample estimates of the policy gradient. By removing them, we reduce the number of sample trajectories needed.

Another problem with Policy Gradient methods is that it averages over the total reward instead of rewarding or penalizing for individual action taken. This

can imply that the training process can take a long time to converge. A suitable alternative would be the Actor-Critic(24) method wherein the actor network approximates the policy function and the critic network evaluates how good the action was. This configuration allows them to act in tandem with each other, which is especially suited to for running in parallel. This would also solve the problem of having to tune the number of samples required in every episodes to achieve convergence fast. But this would also mean more memory requirement and we have not explored it in the context of this study.

3.4.3 Embedded Architecture

To make it possible to perform learning and inference on microprocessors, we have explored the possibility of resource efficient implementation. All the software for the same has been written in C++, to make it platform independent and capable of running on every microprocessor with support for floating point operations. The code has been designed with specific focus on reusability of code by following an object oriented framework which can be ported for any other application which involves deep learning.

To increase the operational efficiency of the runtime architecture, prevalent and reusable operations such as matrix manipulations have been wrapped in functions so that they can be stored in the cache and enable faster execution. Also, special attention has been paid in the order in which floating point calculations have been handled to minimize precision loss as much as possible. The matrix operations involved in backpropagation has been written to be as close to the implementation in major deep learning libraries as possible to benefit from the optimized architectures.

To tackle the memory overflow challenges associated with RTOS, memory allocation for most space consuming data structures, being the matrices which make up the neural network, is attempted at the beginning of the code execution cycle. This is to ensure that there are no memory leaks and that the system will never run out of memory, given that sufficient overall memory exists to implement the desired network architecture. To minimize issues with latency introduced by software runtime cycle and peripheral actuation cycle, an asynchronous/non-blocking programming architecture has been adopted. On microcontrollers with only one microprocessor, this has been implemented using an events and services model. If more than on processor is detected, automatic multithreading of tasks has been enabled.

All the codes for this project is available on Github[1]. The final software flow is given in Figure 7

3.4.4 Results

The demonstration of the results are available on `https://youtu.be/d7NcoepWlyU`.

The Q-learning implementation was done with 8 possible states and 3 possible actions. Although the learning was performed quickly due to the simplicity of the system, the performance was sub-optimal for the following reasons:

- Discretization of states was done by binning the continuous distance values to classes based on range. This meant that on the time scale, the transition from one state to another, when the action taken permits, occurred

[1] https://github.com/aslamahrahman

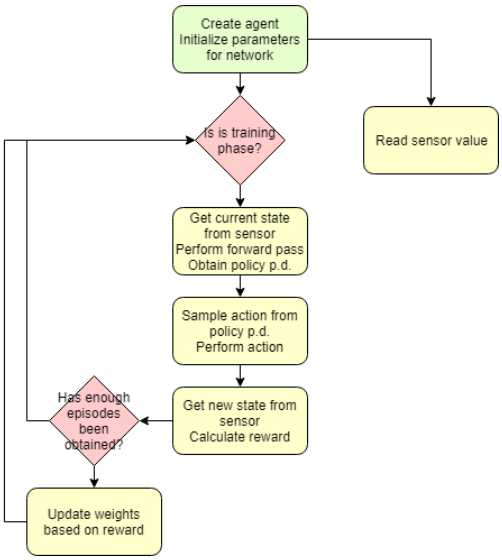

Figure 7: Code flow

almost instantaneously and when not, the feedback to the algorithm in the form of reward would be incorrect. Perfect synchronization of actuation time and state measurement time would have to be performed to avoid this. Even then, the required synchronization would be different on different terrains because actual state change that occurred during the actuation time would differ depending on the dynamics of the actual physical environment, friction etc.

- Since the exploration/exploitation trade-off has to be hard coded in, it is possible that some states would remain relatively unexplored and some states would continue to be more prevalent than the other. If the agent encounters an unexplored state during deployment, it could lead to undesirable results.

Shifting to the Policy Gradient Network based implementation resolved the problem related to continuity of states. It allowed the agent to learn the correlation between actions and state changes better as the environment was perceived to be more responsive. Although the learning process took longer to converge as compared to the Q-Learning implementation, choosing an optimal learning rate and incorporating Reward-To-Go policy allowed for a desirable learning response.

4 Conclusion & Future Work

Through this work, we have successfully explored the nuances in using the state-of-art methods and implementing some on our own optimized for embedded systems for building a physical product that would perform in real time. We have explored the scope of our problem using multiple reinforcement learning methods; Q-Learning, Deep Q Networks and Deep Policy Gradient networks

and their limitations and possible solutions. We have also experimented with performance of neural networks in real-time on embedded systems and their interactions with physical systems.

In terms of specific implementation, we have trained two different networks to perform the two task our robot can accomplish:

- For person tracking, we trained a Deep Q Network which learns to rotate to face the person directly. However, it did not lead to any favourable results, and we settled on the use of a Continuous action controller.
- For obstacle tracking, we have created a reinforcement learning and deep learning library for microcontrollers from scratch and trained it to learn to maintain a certain distance with the obstacle

In the future, we hope to accomplish the following:

- Implement a continuous action controller which can respond to state changes faster and in a smooth and continuous fashion
- For person tracking, attempt a real-time implementation which would combine a Convolutional Neural Network with a Deep Q Network
- Expand the scope of and optimize the deep learning library for microcontrollers to run reliably with even more memory and computing restraints
- Implemented asynchronous learning methods such as Actor Critic Method (24) so that physical delays because of the dynamics of the system does not affect or slow down the training process

References

[1] https://github.com/uTensor/uTensor.

[2] https://www.keil.com/pack/doc/CMSIS/NN/html/index.html.

[3] https://github.com/Microsoft/EdgeML.

[4] http://www.ti.com/lit/wp/slay023/slay023.pdf?DCMP=fr4x&HQS=ep-mcu-msp-fr4x-mspblog-20141023-whip-en.

[5] https://code.fb.com/ai-research/floating-point-math/.

[6] https://www.tensorflow.org/lite.

[7] https://towardsdatascience.com/introduction-to-reinforcement-learning-chapter-1-fc8a1

[8] https://deepsense.ai/what-is-reinforcement-learning-the-complete-guide/.

[9] https://joshgreaves.com/reinforcement-learning/understanding-rl-the-bellman-equations/.

[10] https://gym.openai.com/envs/FrozenLake-v0/.

[11] https://spinningup.openai.com/en/latest/spinningup/rl_intro3.html#expected-grad-log-prob-lemma.

[12] F. Ahmad, A. Najam, and Z. Ahmed. Image-based face detection and recognition: "state of the art". *IJCSI International Journal of Computer Science Issues*, 9, 02 2013.

[13] M. Courbariaux, Y. Bengio, and J.-P. David. Training deep neural networks with low precision multiplications. 2014.

[14] M. Courbariaux, I. Hubara, D. Soudry, R. El-Yaniv, and Y. Bengio. Binarized neural networks: Training deep neural networks with weights and activations constrained to +1 or -1. 2016.

[15] X. Guo, S. P. Singh, H. Lee, R. L. Lewis, and X. Wang. Deep learning for real-time atari game play using offline monte-carlo tree search planning. In *NIPS*, 2014.

[16] S. Han, H. Mao, and W. J. Dally. Deep compression: Compressing deep neural network with pruning, trained quantization and huffman coding. *CoRR*, abs/1510.00149, 2016.

[17] H. V. Hasselt. Double q-learning. In J. D. Lafferty, C. K. I. Williams, J. Shawe-Taylor, R. S. Zemel, and A. Culotta, editors, *Advances in Neural Information Processing Systems 23*, pages 2613–2621. Curran Associates, Inc., 2010.

[18] P. Henderson, J. Romoff, and J. Pineau. Where did my optimum go?: An empirical analysis of gradient descent optimization in policy gradient methods. *CoRR*, abs/1810.02525, 2018.

[19] H. Li, S. De, Z. Xu, C. Studer, H. Samet, and T. Goldstein. Training quantized nets: A deeper understanding. In *NIPS*, 2017.

[20] H. Li, K. Ota, and M. Dong. Learning iot in edge: Deep learning for the internet of things with edge computing. *IEEE Network*, 32(1):96–101, Jan 2018.

[21] T. P. Lillicrap, J. J. Hunt, A. Pritzel, N. Heess, T. Erez, Y. Tassa, D. Silver, and D. Wierstra. Continuous control with deep reinforcement learning. *CoRR*, abs/1509.02971, 2015.

[22] A. R. Mahmood, D. Korenkevych, B. Komer, and J. Bergstra. Setting up a reinforcement learning task with a real-world robot. pages 4635–4640, 10 2018.

[23] A. R. Mahmood, D. Korenkevych, G. Vasan, W. Ma, and J. Bergstra. Benchmarking reinforcement learning algorithms on real-world robots. In *CoRL*, 2018.

[24] V. Mnih, A. P. Badia, M. Mirza, A. Graves, T. P. Lillicrap, T. Harley, D. Silver, and K. Kavukcuoglu. Asynchronous methods for deep reinforcement learning. In *ICML*, 2016.

[25] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, and M. A. Riedmiller. Playing atari with deep reinforcement learning. *CoRR*, abs/1312.5602, 2013.

[26] F. Ortega-Zamorano, J. M. Jerez, D. Urda Muñoz, R. M. Luque-Baena, and L. Franco. Efficient implementation of the backpropagation algorithm in fpgas and microcontrollers. *IEEE Transactions on Neural Networks and Learning Systems*, 27(9):1840–1850, Sep. 2016.

[27] M. Rastegari, V. Ordonez, J. Redmon, and A. Farhadi. Xnor-net: Imagenet classification using binary convolutional neural networks. In *ECCV*, 2016.

[28] T. Schaul, J. Quan, I. Antonoglou, and D. Silver. Prioritized experience replay. *CoRR*, abs/1511.05952, 2016.

[29] J. Schulman, S. Levine, P. Abbeel, M. I. Jordan, and P. Moritz. Trust region policy optimization. In *ICML*, 2015.

[30] J. Schulman, P. Moritz, S. Levine, M. I. Jordan, and P. Abbeel. High-dimensional continuous control using generalized advantage estimation. *CoRR*, abs/1506.02438, 2016.

[31] D. Silver, J. Schrittwieser, K. Simonyan, I. Antonoglou, A. Huang, A. Guez, T. Hubert, L. R. Baker, M. Lai, A. Bolton, Y. Chen, T. P. Lillicrap, F. F. C. Hui, L. Sifre, G. van den Driessche, T. Graepel, and D. Hassabis. Mastering the game of go without human knowledge. *Nature*, 550:354–359, 2017.

[32] R. S. Sutton, D. McAllester, S. Singh, and Y. Mansour. Policy gradient methods for reinforcement learning with function approximation. In *Proceedings of the 12th International Conference on Neural Information Processing Systems*, NIPS'99, pages 1057–1063, Cambridge, MA, USA, 1999. MIT Press.

[33] M. Tee Kit Tsun, B. T. Lau, and H. Siswoyo Jo. An improved indoor robot human-following navigation model using depth camera, active ir marker and proximity sensors fusion. *Robotics*, 7(1), 2018.

[34] P. Viola and M. Jones. Rapid object detection using a boosted cascade of simple features. In *Proceedings of the 2001 IEEE Computer Society Conference on Computer Vision and Pattern Recognition. CVPR 2001*, volume 1, pages I–I, Dec 2001.

[35] Y. Wang and D. Lee. Reinforcement learning for a human-following robot. In *ROMAN 2006 - The 15th IEEE International Symposium on Robot and Human Interactive Communication*, pages 309–314, Sep. 2006.

[36] S. Wu, G. Li, F. Chen, and L. Shi. Training and inference with integers in deep neural networks. *CoRR*, abs/1802.04680, 2018.

[37] X. Zhang, T. Gonnot, and J. Saniie. Real-time face detection and recognition in complex background. *Journal of Signal and Information Processing*, 8:99–112, 05 2017.