

Project DeepLearner Report

Senior Design Team 10, May '23

April 30, 2023

Abstract

The DeepLearner Platform is an Interactive, Educational Embedded Machine Learning Platform for Iowa State students, achieved by training autonomous AI models to navigate a racetrack in a simulated virtual environment based on AWS DeepRacer Architecture.

1 Introduction

With limited opportunities to learn about embedded machine learning systems, we set out to create a series of educational labs for our fellow undergraduate students. We found the AWS DeepRacer, a ML autonomous race-car, to be an excellent platform to teach students; however, AWS costs and lack of scalability to classroom quickly deterred some of our earlier efforts. As a result, we set out to teach students about embedded machine learning and to provide Iowa State with a cost-effective and scalable platform to provide to these student as well educational documents and activities.

2 Design

2.1 Functional Requirements

- DeepRacer AI models should be able to be trained on machine(s) located on the ISU Network using our platform.
- Our platform should be scalable and practical for use in a classroom environment. This means that Professors and TAs have access to the infrastructure and that it can handle many different users as well as be cost effective.
- Users should be able to upload custom reward functions, start and stop training on the back-end, evaluate models, view models as they train in the simulated environment, and download their model without direct access to the infrastructure.

2.2 Non-Functional Requirements

- Front-end user experience should be pleasant, with an appealing graphical user interface (GUI) with and intuitive user flow.
- Educational documents should also be pleasant to look at and appealing.

2.3 Relevant Standards

- IEEE 2830-2021: IEEE Standard for Technical Framework and Requirements of Trusted Execution Environment based Shared Machine Learning
 - With a potential fleet of multiple cars in the educational environment, this would effectively fall into the domain into shared machine learning. As a result, we will need to comply with the IEEE standard.
- IEEE 802.11: Standard for wireless LANs
 - We will be using the 802.11 ac Wi-Fi standard for our project. This is because machine learning applications to embedded systems almost always require a network connection to a more powerful computing devices in order to be able to process its data

- IEEE 26531-2015: International Standard for Systems and software engineering
 - As code within a group can become very messy, we have decided to follow the process in this document to ensure that our code conforms to a basic standard to ensure readability and reliability.

2.4 The DeepLearner Platform’s Evolution

When we first started our project and its design, we had originally focused in on first learning how to compete with the DeepRacer, taking paths to find its fullest potential in terms of showing students about machine learning, and the second was to center a few labs around it. Not only did we find how limiting the AWS platform was at the start, but we also came to find how limited the opportunity would be for students to interact with the platform. With the intention of providing set of four labs at the start, this would mean roughly ten hours of model training/evaluation per student per week enrolled in our target course of Introduction to Embedded Systems (CprE 288). Looking at past enrollment hours and the rates of cost for training a DeepRacer, this would equate to around \$42,000 per-year for the university provide this opportunity to students.

Table 1: AWS DeepRacer Platform Costs

Item	Rate
Training/Evaluating	\$3.50 per hour
Storage	\$0.023 per GB per month

With the official rates above, lets place a student in the environment to create a singular model. With the standard time to train a model being roughly 30 minutes, they create 8 different reward functions placing Bob’s usage at 4 hours before accounting for evaluation, and even before comparing results with their peers. The AWS DeepRacer, while great, can quickly pile up costs for the university. \$35.00 may not seem like a whole lot, however, in the context that this is for one student for one week begins to paint the costs of running such a platform for a course. Table 3 , showcases the cost for the course of CprE 288 that typically averages around 300 students per year. This lead us to begin looking for other solutions to keep the DeepRacer as our main platform.

Table 2: Example Usage for an ISU Student per lab week

Student	Item	Rate	Quantity	Cost
Bob	Training/Evaluating	\$3.50 per hour	10 hours	\$35.00

Table 3: Example Usage for CprE 288: Introduction to Embedded Systems

Students	Item	Rate	Quantity	Cost
300 Students	Training/Evaluating	\$3.50 per hour	40 hours per student	\$42,000

This presented the opportunity for us to add a second design element, one that would allow us to provide an environment for us to have the ability to build labs around, but while also saving the university money if they were to implement our platform. This lead us to begin performing research, which included asking AWS employees, talking to various other DeepRacer enthusiasts, and joining various DeepRacer related events. Eventually, we found a committed community full of DeepRacer enthusiasts that had achieved training their models on their local hardware. This was achieved by two main open source projects, a local implementation of the AWS Sagemaker and the AWS Robomaker services. While talking to former AWS employees, we discovered the process for training a DeepRacer AWS. Whenever a user submits a training request, a container is placed into an EC2 instance (elastic computing VM) and is started. At this point, we knew that the AWS DeepRacer was comprised of various open source software, as its own software went open source back in 2019, however, what we didn’t know was that in order to train a DeepRacer, various open source software is also used. Taking this information back, we began looking into the two previously stated open source projects on the DeepRacer community GitHub repository, to our surprise they also followed a similar process of using Docker containers in order to achieve distribution. With this in mind, taking all of our research as inspiration with their designs we

quickly went to work in combining the two projects to create an environment. Technical details can be found in the implementation section of this report.

As we completed our first few tests with training a local DeepRacer model, we found just how delicate the process can be to initiate the containers and for said containers to function properly. Till this point, most of the DeepRacer container interaction had been performed through a command-line interface, this would be acceptable for engineering students - however, we felt that in order to have a cohesive platform for students to learn about machine learning, we cannot put control of the container in the hands of the students to break. We also had to figure out how to install the software on lab computers, avoiding complex processes that can also break the container. This led to the creation of the front-end GUI and the idea of having a DeepRacer server to serve students in a similar method of how AWS controls its DeepRacer containers. During the development of the front-end GUI we also hit many roadblocks, including the front-end GUI not being consistent in its operation of the container. This led to yet another idea, we needed a framework to control both data access and the container control - leading to the development of the Racing Environment Framework, REF, built on the Django framework to handle our student's requests as they interact with the GUI. Technical details can be found in the implementation section of this report.

Shortly after, with the three main components to have a similar, if not better, experience than on AWS, our machine learning environment was ready for testing with the labs developed alongside the platform. These labs were also written through tedious research and a lot of interviewing our peers, TA's, and some professors. We successfully tested the platform as a whole in the early month of April 2023, finalizing a design for the first time since we started the project.

Through various design changes and design directions, the DeepLearner Platform is a culmination of research and development in an effort to teach students about machine learning. The team's ability to adapt to changing requirements allowed us to rapidly develop complex systems that can be used to follow complex labs.

3 Implementation Details

3.1 DeepLearning Platform Architecture Overview

The platform aims to provide students with an interactive and educational sandbox to learn about machine learning. There are two design efforts to our project, the educational and the environmental. Below is the structure of students using our platform.

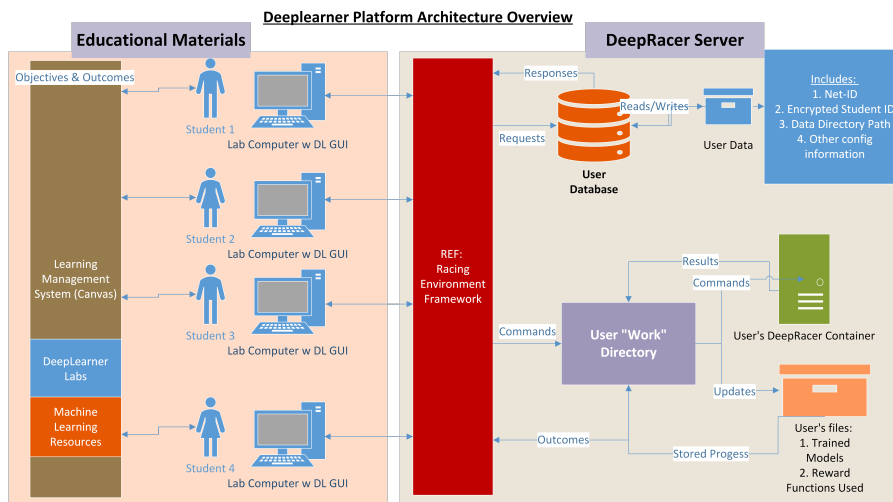


Figure 1: The DeepLearner Platform's Architecture Overview.

3.2 DeepRacer Container

The DeepRacer container is built on two open-source projects, the DeepRacer Community’s Robomaker project and the Sagemaker project. These two have very different purposes, the first allows you to evaluate your models locally by simulating the DeepRacer and the environment whereas the Sagemaker performs reinforcement learning coaching and enables the training of models. While there are similar platforms to ours, such as the DeepRacer for Cloud, these wouldn’t work in our environment - as they were built with the purpose of training a singular model to compete in the AWS DeepRacer Leagues. These stray on the commercial side, with multiple copy-cats also used for commercial purposes. Ours, however, takes on the two projects, conjoining them with scripts, and applies a different approach to the deployment. We needed an environment that could take on the workload of various students - for this purpose we have isolated environments for containers, using a nuclear core as the inspiration behind this approach. When generating nuclear power, the core is controlled through control logs, similarly our server is the core and control rods are inserted. These rods represent a student using their custom environment. These rods are dropped in and removed accordingly by the REF to ensure that all of the server’s resources aren’t hogged by rogue sessions. Inspired by the official implementation of deploying a container on an EC2 instance when using the

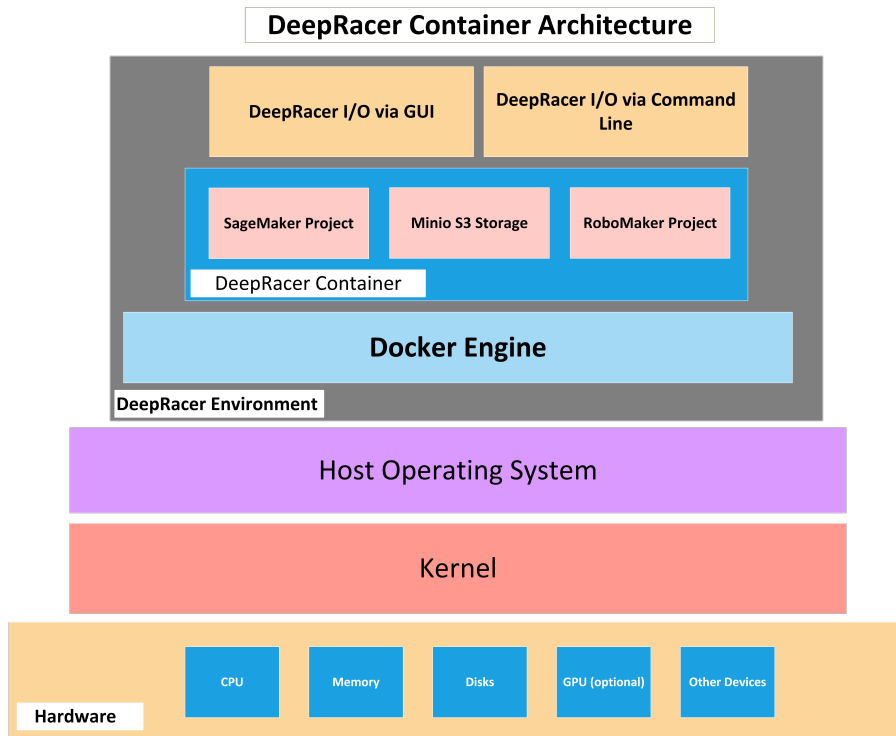


Figure 2: Abstraction of the DeepRacer container on local hardware.

AWS implementation of the platform, we opted to use Docker containers to ensure all dependencies are included in a packaged format that can be quickly deployed to different students. Since each student has their own container, we use one base container to duplicate. In the container there are three main services that allow for the training, evaluation, and racing of the virtual DeepRacer. The Sagemaker implementation, which is based on the open source project from the community - is responsible for training the DeepRacer and providing updates to the Robomaker project. The Robomaker project, simulates a virtual environment and the robot, and returns updates to the Sagemaker to update the model. Since AWS uses their S3 storage for connecting the two services, we too have a local Minio instance that connects the two. The Minio instance allows us to store objects that are then used by the two projects. It is important to note that we do not claim having developed the two main services, nor the system that connects it to the Mino database, however, we did implement the scripts that control and initialize the containers with the help of the AWS Community as we queried them with research questions.

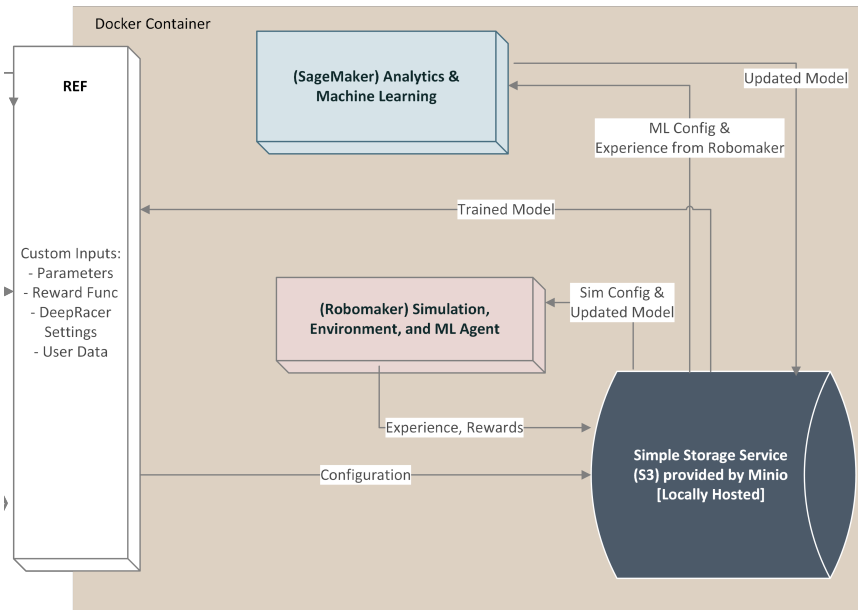


Figure 3: Container Architecture.

3.3 Racing Environment Framework

The Racing Environment Framework, REF, was developed to abstract control of the DeepRacer and to implement a few more features that would make this platform work in a classroom setting. Firstly, when we moved from installing the container on each dedicated lab machine to having a centralized server, we found that we could create a dedicated environment for each student with ease, removing hours of maintenance labor and giving TA's tedious tasks of replicating containers for each student on different computers, and having to deal with varying errors. Secondly, when we moved to developing a GUI to make it easier for students to use our platform - we were simply initiating our scripts that we had developed to run the container, while this was effective to a degree - this could become rapidly complicated for anyone working on the system, and we could not account for all issues that may arise. With these two situations in mind, we decided to implement a dedicated application that would handle requests from the front-end, performing tasks the same exact way - regardless of button combination pressed, this also enabled us to have a singular, base container that can be duplicated for each student. One base container to maintain is a lot simpler than 300 student's containers. Finally, by giving each student their own dedicated environment, we needed authentication to ensure that students could only access their own files and container.

This, is where the Racing Environment Framework was developed to feel the need to simplify the day-to-day operations of our users and of the administrators, further removing control from the students - creating a platform comparable to AWS. To truly understand its full implementation, we will walk through the framework's functions and additional background.

The REF is based on the Django Framework, and it sits on our server. It has a database to keep track of its users. The database, currently, is simply a JSON file named `users.json` - in the future it may be beneficial to implement a dedicated structured query language database to ensure data integrity - however, since the file only holds user's encrypted student id, their net-id, work directory, a randomly assigned port for the user's container's output (spans from 8100-8500 on the server, no two users will have the same port) and a few "check" variables to keep track of the user's progress through the platform such as "has the user's container been initialized?" we figured the JSON would suffice to service 400 students at a time.

The GUI implements a sign-up sequence and a login sequence. The sign-up sequence, requests that the framework creates a new user with the information it collects during the sign-up process. When a user is created, it will firstly calculate the hashed student-id (more on this in later in this section) store it on the database, and also store a pseudo-random generated salt. Additionally, their work directory is created and the base DeepRacer container is copied over to the user's directory.

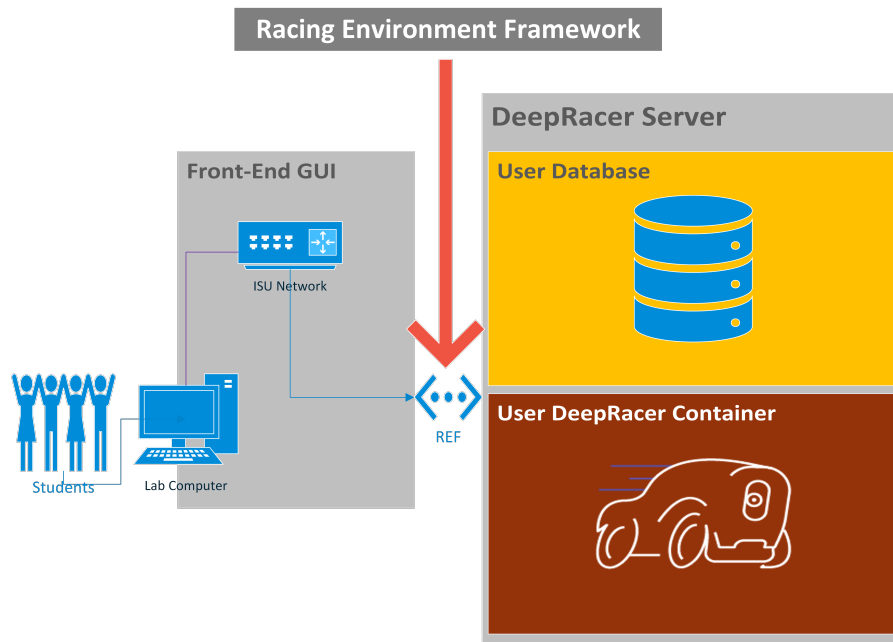


Figure 4: Abstraction of the Racing Environment Framework. Connects the front-end GUI to back-end services.

To authenticate, users will enter their university net-id and their student id. Since we did not want the student id's being stored in plain-text, the student id is treated similarly to how passwords are treated on UNIX-based systems. The student-id is hashed using the secure hash algorithm (SHA) in 256 operation mode. Once the ID is hashed, it is then sent to the framework using an HTTPS POST request, where it is then concatenated with the user's previously generated salt, then hashed one more time using SHA-256, then compared. If correct, the framework provides a pseudo-generated token that is used through the entirety of their session. This token is refreshed at the end of every training or evaluation cycle, or when a new connection is initialized.

Additionally, the framework also provides a CSRF token to protect against cross-site scripting attacks at the start of every session that is periodically updated when a user performs an action. Using Django as our base framework, we implemented HTTPS requests to ensure data is also encrypted through POST and GET actions through the GUI.

Once a user is authenticated, the user's data is returned to the GUI automatically, this data is used to provide a seamless experience. As mentioned, among the stored data is the port numbers, the user's data directory, and a few other check variables. These are used on the front-end to ensure that the user is able to work through their labs with minimal obstruction. For example, if the user is looking to view their robot's training progress, a browser window is automatically opened with their assigned port for them to view a stream or to view analytical data.

Additionally, but most importantly, is the control of the container. As mentioned, the DeepRacer container is mostly two open source projects bridged together using scripts to provide a seamless machine learning environment, however, these scripts must be executed at specific times as there are different dependencies throughout - matter of fact, even our initial implementation of the GUI only called the scripts and waited for scripts based on estimated time using busy waiting, this could cause deadlocks and lead us to automate the control via the REF. The REF, most importantly, has a system of checks that ensures all of the necessary components are running correctly and if they are not, will retrieve updates, or initialize the missing components. No longer using busy waits, it'll check to see if a service has been initialized correctly before continuing onto the next step. With this handled for the user, all the user has to do is click a button and all the container has to do is wait for instruction from the REF, making it easier for us to modulate our front-end actions and operations.

Lastly, the REF is also used to perform periodical "garbage" collection, in which it is invoked at the top of every hour to check for rogue training/evaluation sessions, closing ports, and stopping Docker services if there has been a set period of inactivity (more than 20 minutes). This is

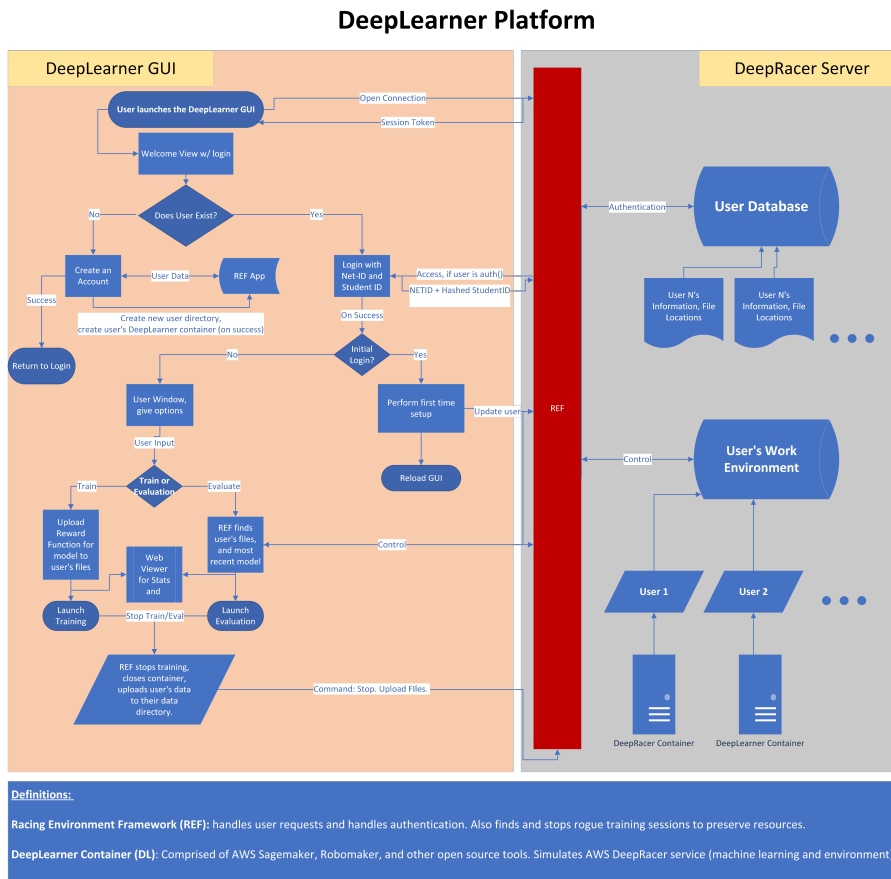


Figure 5: Illustration of the REF's Operation (in red).

to preserve resources within the server, and ensure students have the most available resources available to their experiments.

3.4 Front-End Implementation

The front-end exists for two reasons: to make for a better experience that is similar to training models on AWS, and also to keep students from having access to more commands on the server than they need. We first figured out how to send basic commands to the back-end. These commands were things like uploading a reward function, stop and start training, create an SSH connection to the server, and was designed mostly ad-hoc and features were added as they became necessary and are present in the current GUI that is located on our repository. The GUI was coded in Python and relies on the following libraries for core functionality: Tkinter, Paramiko, PIL, and OS. The code for the GUI itself will store information on the local server for the purpose of creating SSH connections. Other information like specific port numbers for viewing training over HTTPS will be given to the front-end by the server as needed. Other important features were also added to the current version like feedback for invalid reward functions and downloading models. The front-end mostly comprises the GUI but also the website where you can view training. We will be focusing mostly on the GUI in this section.

Something that became evident during creation of the GUI that influenced design of the back-end was the need for multiple user directories on the server. Multiple students would be training models at the same time and it would be necessary to keep track of these models. Although creating multiple local machines for each student to access individually would be a solution to this problem, we had another idea that fit into our vision better. This was the REF architecture, where each student had their own container on the same machine, and simply used different port numbers to access their models. This introduced the issue of security and authentication on the front-end.

To keep students from accessing and accidentally or intentionally modifying another student's

model, we use a student's NetID and student ID number for authentication. Students or TAs can add users and login credentials from the front-end. In order to keep the student ID number secure, it is encrypted using SHA-256 before it is sent over the network to the server. When it comes to accessing the server itself, SSH connections are authenticated using a username and password which are enumerated in the code itself. In practice, these credentials, along with the code itself would be encrypted before being installed on lab machines to protect that information.

Some other relevant information on the GUI is that file transfer for reward functions and model files are conducted within the SSH session using Secure File Transfer Protocol. Session tokens are also given to each SSH connection and used to prove data integrity to the back-end, however low the chance of a replay attack might be.

3.5 Educational Material

Our project relied heavily on the implementation of the educational aspect. Without a well thought out plan for creating and using the device and software in a classroom, the project would not be successful. We first started with researching machine learning with outside resources such as research papers, notes from other machine learning based classes at Iowa State, and a coursera course. This allowed us to find the most relevant topics in machine learning to apply to keep students from being overwhelmed. With this information, we created the prelabs to set a base understanding of what machine learning and reinforcement learning are. After this, we researched and looked deeper into what makes a lab "good". We looked at the CyBot and reflected on other labs at Iowa State to help us understand how professors keep students engaged and interested in a lab. With this information, we created the basis of the labs, setting clear goals at reachable intervals for the students to reach. For the second lab, we focused our research more on the physical DeepRacer bot. To create this lab we looked through various AWS DeepRacer documentation, DeepRacer GitHub repositories, ROS documentation, as well as the source code of the physical bot. From these materials we were able to gain a better understanding of the DeepRacer bot and create a lab around our findings. The lab focuses on the inner workings of the bot and how communication between the sensors and motors and bot work. The lab is split into three parts with the first part focusing on the ROS nodes that make up the DeepRacer core application running on the bot. The second part focuses on Pulse Width Modulation and how it can be used to control the servo and motors. The third part focuses on the AWS project "Follow the Leader" and introduces students to the many functionalities the DeepRacer can achieve through custom projects. Both labs should give students a well rounded understanding of how DeepRacer works both virtually and physically.

The most difficult aspect of the educational implementation was putting ourselves in the position of a younger and more naïve student. Explaining complex topics in a digestible manner is a challenge which takes planning and thought and with our testing of the lab documents we were able to accomplish this and overcome the difficulty.

3.6 Physical Implementation

For the physical implementation of the track, we opted to use interlocking foam workout pads for the track. This allowed us to meet the design constraints given by Amazon for the DeepRacer bot as well as gave us a modular approach which allows the contraction or expansion of the track should we need to make it smaller or larger given different size constraints. It also allows us to easily set up and take down the track to allow easier integration into CprE 288 as that class will already have a course for the CyBot.

The physical implementation of the bot is extremely simple. On setup of the bot, you are taken through a calibration sequence to be sure the bot's movements are consistent with the commands simulated training.

4 Testing

Software testing and verification took 2 different approaches. The first was partner programming. This allowed us to stop many issues before they happened as there were twice the eyes validating the code was correct before true testing. Next, we implemented peer testing, where a peer unfamiliar with the specific section being tested would use the program to find any poor designs or mistakes in the function or usability of the code the GUI. Finally, we implemented test cases, such as a

faulty reward function, a program left to run too long, or abnormally large model sizes. These tests allowed us be certain that the software would be adequate for our needs.

Testing of the labs was simple after the software had been fully tested. As we knew the software worked, we walked through the model training process to be sure that the lab documents had the steps in a correct and intuitive order. One thing we did was ask fellow students we know if they could read through our lab documents and understand the goals and information. This is how we measured how well we were able to explain the material to a student unfamiliar with our design and project. Doing this was a good way to adjust the prelab information to be sure we did not overload a new student.

Testing of the bot was extremely simple. We simply connected to the bot using the instructions provided by the AWS DeepRacer website and put it into manual mode. In this mode, we were able to drive the bot to confirm the turning angle, speed, cameras, and alignment were as required for the model to work properly. After these tests, we were able to confirm the bot was ready to accept a model and perform a lap on the track.

5 Related Projects and Literature

5.1 DonkeyCar

DonkeyCar is a related product that was considered for the project. It was an open-source RaspberryPi based platform which was a very compelling option compared to the DeepRacer. It was cheaper (at MSRP) to the DeepRacer bot, however it needed to be assembled with a parts kit and one of 4 supported RC car chassis.

What made this platform very appealing to us was the open-source nature of the bot. The RaspberryPi base made it much easier to understand how it was driving the car as the documentation for it was very nice. As the project has a focus on an embedded systems lab, the ability to get closer to the hardware was appealing, however the DeepRacer community also provides lots of documentation which helps negate this difference between the bots. The DeepRacer community is also much more active, which allows faster and more efficient debugging.

The DonkeyCar platform also used a different form of machine learning; While the DeepRacer uses reinforcement learning, which bases its training on a reward function, the DonkeyCar uses a technique called behavioral cloning, where a user drives the car around a track and the model is created based on the inputs. This was appealing as it would help increase the hands-on aspect of the lab, however we ultimately decided that reinforcement learning would be the better choice. Behavioral cloning converges faster as the parameter estimation is simpler, however this method is used much less than reinforcement learning, and the maximum quality of the model is much lower than what can be achieved with reinforcement learning. This is one of the reasons we chose the DeepRacer as our platform.

The other reason we chose the DeepRacer as the platform was the scalability of it. The DonkeyCar supported RC chassis were very often sold out and difficult to get. When we searched, we had the option of buying second hand, which could have introduced lots of issues, or waiting for a restock. RaspberryPi's are also currently experiencing a shortage which has caused the models we needed to be far above MSRP.

These issues and decisions ultimately led us to using Amazon's DeepRacer as our platform of choice.

5.2 CyBot

The CyBot is the robot used in the CprE288 Labs currently. It is a Roomba based device equipped with a Tiva micro-controller for programming. It is very well adapted for the CprE288 labs as it gives the students close access to registers and sensors to give students good experience dealing with embedded hardware.

When designing the DeepRacer educational material, we looked at the success of the CyBot to see how we could improve our labs. The first was to see what about the CyBot made it so engaging to students. We determined 2 big things: The process of creating and optimizing the code and the reward of having a live demo you can watch. To give the same feeling of accomplishment to coding the DeepRacer as the CyBot, we opted to use a continuous state model to give students the maximum choices when creating their reward function. We also dove deeper into the DeepRacer

Core files to extract very hardware-oriented files such as the PWM.sh file. This file controls the motors and by giving them access to it we give students the same feeling of control as with the CyBot.

To give students the same feeling of reward when testing the robot as well, we added a non-simulated portion to both labs. This gives the same reward of watching their code work as with the CyBot and keeps students from becoming bored or uninterested in the labs and topics.

6 Appendix

6.1 User Manual for Students

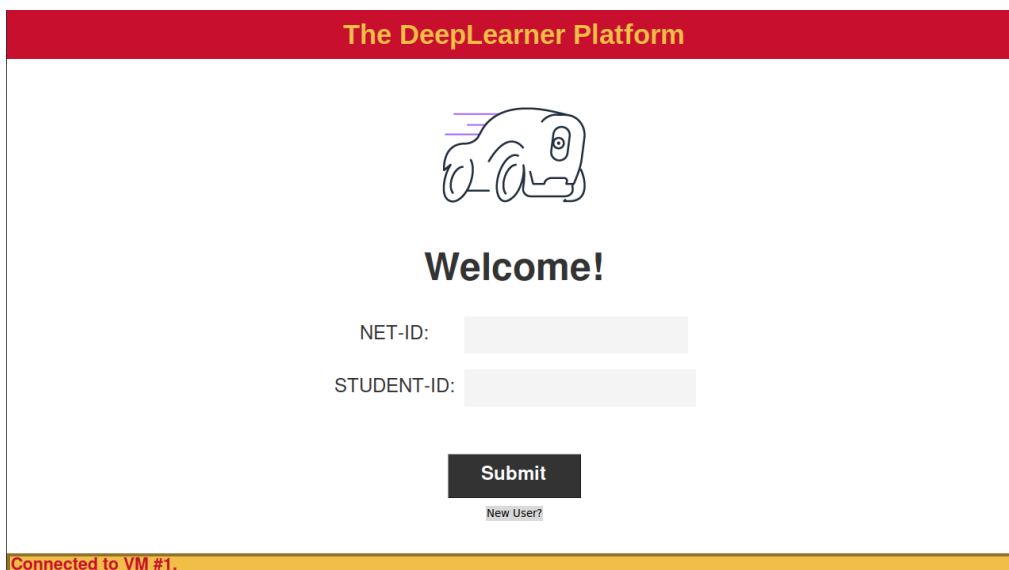
Using the DeepLearner Platform

Student Guide

1. Who is this guide for?

This part of the guide is specifically for students. Students get access to limited commands and primarily start and stop training with custom reward functions.

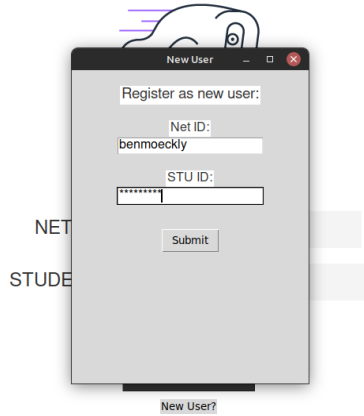
2. Creating a User Account



The screenshot shows the login interface of the DeepLearner Platform. At the top, a red banner reads "The DeepLearner Platform". Below this is a white background featuring a stylized elephant logo. The word "Welcome!" is centered. There are two input fields: "NET-ID:" and "STUDENT-ID:". Below these fields is a black "Submit" button and a smaller "New User?" link. At the bottom, a yellow bar indicates "Connected to VM #1."

When you start the DeepLearner GUI, you will be greeted with a login screen that you use to manage AI models and initiate training and evaluation. If this is your first time using DeepLearner you'll want to create an account by clicking on the "New User?" button. You will be prompted with the following pop-up. This is where you will enter your NetID and nine digit student pin number. These will serve as your login credentials. Make sure you enter them correctly.

The DeepLearner Platform



Connected to VM #1.

3. Training a model

Now that you have created your own account, you have dedicated resources at your disposal and you can upload a reward function to use for training your own AI model. Log in using the credentials you just made.

The DeepLearner Platform



Welcome!

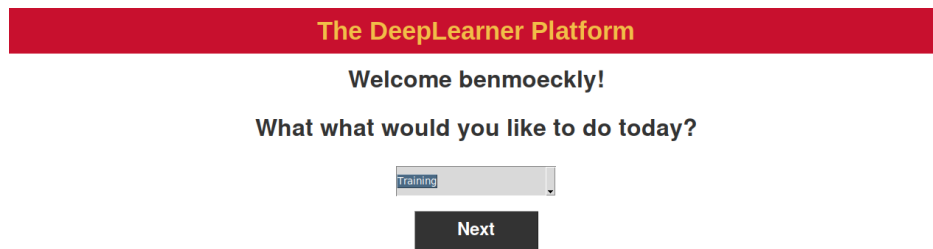
NET-ID:

STUDENT-ID:

New User?

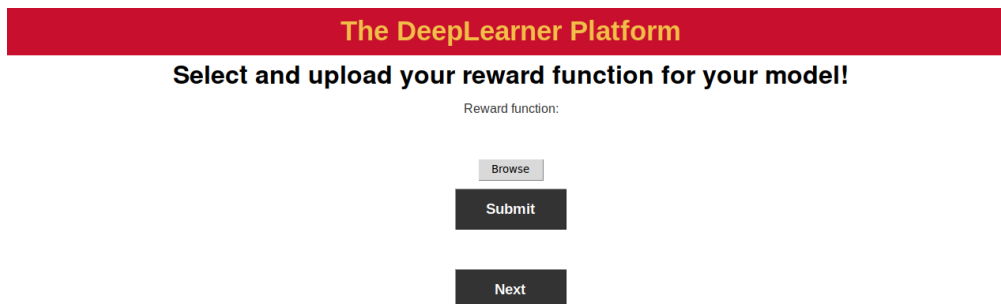
Connected to VM #1.

The next screen will ask you what you want to do. The drop down box gives you two options. To start you want to select “Training” and press “Next.”



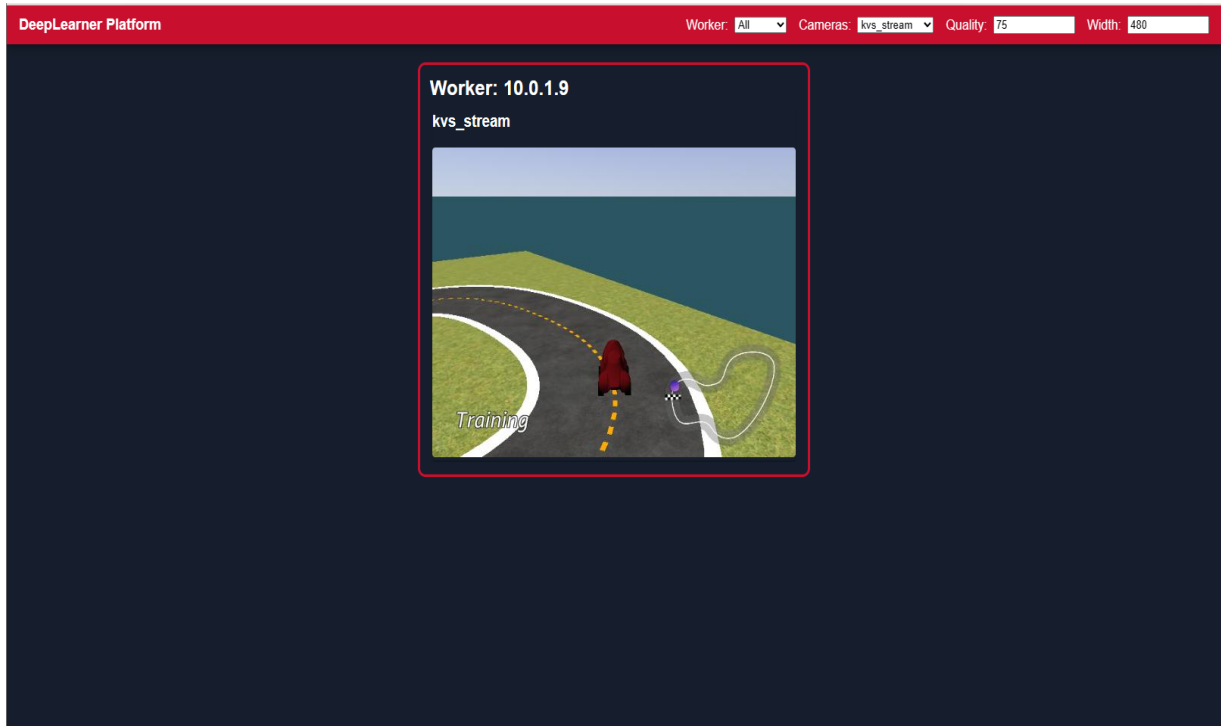
Welcome benmoeckly, successfully authenticated!

The next screen will give you the option to upload your reward file. This is what will dictate the behavior of your model



Preparing DeepLearner

Press “Next” to move on. The GUI will open up you default web browser and direct you to the port and address on which you will be able to view your model.



6.2 Alternative/Original Designs

At the beginning of the year, we thought that just creating different reward functions and optimizing a robot might be enough for the entire project. We were proven wrong because within two weeks of getting a DeepRacer and signing up for AWS we had already figured out how to get the robot to run a lap and get it ranked on the top 100 models in the country on Amazon's website. After that we thought that we might try to organize a team to compete in the official Amazon competitions and start a club at ISU. However, the timing was not right as we would have all graduated by the time the team would go to competition. Finally, we decided the best way to give ourselves a proper project was to make our own machine learning platform for the DeepRacer.

Some other scrapped ideas were that we thought that models would be trained on the desktops in lab computers. While this is technically possible with our current design, it would be impractical because there would be issues with students trying to get models on the same computer as well as varying computing power between machines.

6.3 Acknowledgements

This project would not have been possible without open source software licensed by Amazon and maintained by the DeepRacer community.

- <https://github.com/aws-deepracer-community/deepracer-sagemaker-container>
- <https://github.com/aws-deepracer-community/deepracer-simapp>