Learning to Walk: A Tale of Jackrabbots Path-finding Adventures
CS231N Winter 2015 Project Milestone

John Doherty
Stanford University
Stanford, California 94305
doherty1@stanford.edu

Kathy Sun
Stanford University
Stanford, California 94305
kathysun@stanford.edu

Abstract

The Jackrabbot is an autonomous delivery robot designed to share pedestrian walkways. In contrast to autonomous vehicles, this proposes the challenge of interacting safely with humans and bikers in a socially acceptable fashion. The path of the Jackrabbot then becomes nontrivial, as it cannot block pedestrian traffic or scare fellow travelers. The goal of our project is to develop a path-planning algorithm for the Jackrabbot that learns over time how to navigate in the most natural way using both computer vision techniques and convolutional neural networks. This project will combine aspects of CS231A and CS231N with the Jackrabbot Research Project from Silvio Savarese's lab. The optical flow portion will be used for CS231A, and the convolutional neural network portion will be used for the CS231N course project.

1. Introduction

Path-finding and obstacle avoidance have been active areas of research since the development of autonomous vehicles. The Jackrabbot, an autonomous delivery robot, faces the additional challenge of interacting safely with humans and bikers on pedestrian walkways in a socially acceptable fashion. We want the Jackrabbot to learn path-finding behaviors by being driven by humans, given only what it observes from a single camera.

We ultimately hope to train the Jackrabbot to navigate through a diverse set of pedestrian environments including sidewalks and hallways. The goal of this project, however, was to explore the feasibility of learning to navigate using a single camera from training examples provided by a human driver. To achieve this, we decided to constrain the task by focusing on hallway navigation. Specifically we wanted the Jackrabbot to learn to go straight down hallways without hitting stationary obstacles or pedestrians. Framing the problem in this way gives us a clear objective and reduces the number of possible situations we have to learn. Additionally, in the training examples we kept the robot at a constant speed, so the task was simplified to learning the orientation of the robot. The problem of predicting orientation was again simplified by turning it into a classification problem in which the robot chooses between going left, right, and straight at each timestep.

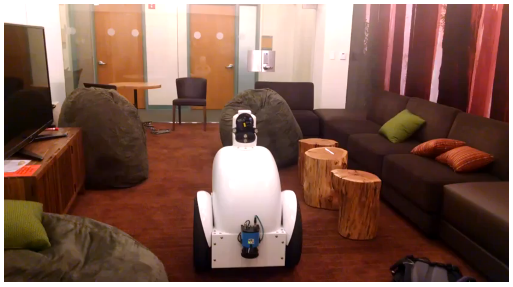


Figure 1: The Jackrabbot. Link to video: `https://www.youtube.com/watch?v=e0sNH9ZUKK0`.

Our work is somewhat similar to that done on ALVINN, an autonomous land vehicle in a neural network, developed at Carnegie Mellon [1]. ALVINN attempted to solve similar challenges but in a driving environment. We wish to apply similar techniques to the Jackrabbot, operating in a pedestrian environment.

We will reference material from CS231A [5, 6, 7, 8] and CS231N [4].

2. Technical Approach

In this section we will describe the approach we used to build a dataset and train convolutional neural networks to learn Jackrabbot navigation from image inputs. As will be described, there are a number of unique challenges that arise when you try to build and use your own dataset to train deep neural networks.

2.1. Dataset

For this project we recorded our own dataset by driving the Jackrabbot through the hallways of the Gates Computer Science building. We simulated and encountered various scenarios while driving through Gates. We navigated around students walking in the hallways as well as various static obstacles including chairs, trashcans, and door frames. We recorded 122 sequences, each consisting of images from the front two stereo cameras and joystick data. Images were recorded at about 10 fps, and we ended up with about 26,000 images from each camera. To build the dataset, we associated each image with nearest joystick data point in time. We then classify this joystick data point as either left, right, or straight.

We ended up with a set of 26,000 images associated with 1 of 3 classes. But because of the nature of driving down a hallway, our class distribution was heavily skewed as seen in Table 1. We resolved this by eliminating some of our data to make the distribution more uniform. This left us with about 13,000 images. As we will discuss later, we experimented with learning on both the uniform and non-uniform class distributions.

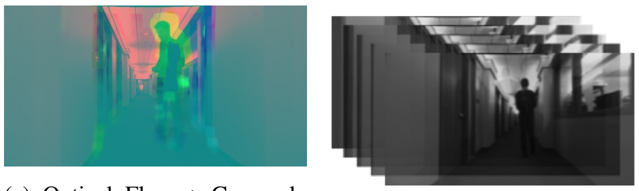
For training and tuning our various networks, we used 85% of our data for the training set and 15% of our data for the validation set. We saved four recorded sequences not in either set for testing the accuracy.

2.2. Convolutional Neural Network

We encountered a number of unique issues trying to learn navigation patterns for the Jackrabbot. First, since we had to collect all of our own data, our dataset was rather limited. While this does not seem like it would be an issue for a three-class classification problem, it is challenging because the amount of variability within each class. For example, the driver may decide to turn left because the robot was getting too close to a wall, or approaching a pedestrian or obstacle. These all look very different in the images, but all result in the same classification.

Additionally, single images are often ambiguous. You cannot usually tell how fast a person is walking from a single image. Thus, we had to incorporate some notion of time in our model. This has been done in other works by using optical flow images or replacing the channels of the input with past frames in the sequence [7]. Examples of these two inputs can be seen in Figure 2. We used optical flow images as a baseline, but focused more on optimizing the latter for this project. As we will discuss later, we experimented with the number of channels in the input (i.e. changing the number of past frames the network examines when performing a classification).

The network structures and other experiments are detailed in the next section.


(a) Optical Flow + Greyscale Image

(b) Stacked Greyscale Images

Figure 2: Inputs that Capture Temporal Features

3. Experiment

3.1. Implementation

To collect data, we used a VirtualBox with Ubuntu to connect to the JackRabbot. We saved data in ROS in bagfile format. From the bagfile format, we exported the left and right greyscale frames as well as the joystick data and saved them as jpeg images and a text file of labels using a Python script. The frame rate of the images (10 fps) is much slower than that of the joystick data, so we sampled the joystick point right before the image was captured.

After the data was exported as jpegs, preprocessing, ie data augmentation, normalization, and feature extraction, was done in Python. The optical flow was extracted from the images along with a new label file using OpenCV [9] in Python. We also doubled our data by mirroring the images and flipping the labels, however this did not perform well, so we used the original dataset.

The data was split into a training, validation and test set and converted into lmdb format using C++. The mean of the images in the training set was also saved in this step.

Different nets were created and ran on different preprocessed lmdb formatted data using Caffe [3]. We used scripts to sweep through the learning rate hyperparameter and plot them.

All the visualizations and testing of our models were done in Python.

All the code can be found in our git repository at `https://bitbucket.org/kathysun/jackrabbot-navigation`

3.2. Sanity Checks

In addition to visualizing our outputs, we implemented a few visualizations for our inputs as sanity checks.

To get a feel for the direction input labels, we plotted the distribution of the raw joystick data shown in Figure 3. As you can see the data is evenly centered around zero with approximately the same number of left and right turns. However, the Jackrabbot is moving straight a significant amount of time. Since there are also two peaks at -1 and 1 (the edges of the joystick), but the rest of the data remains relatively close to 0, it was difficult to determine where the bin

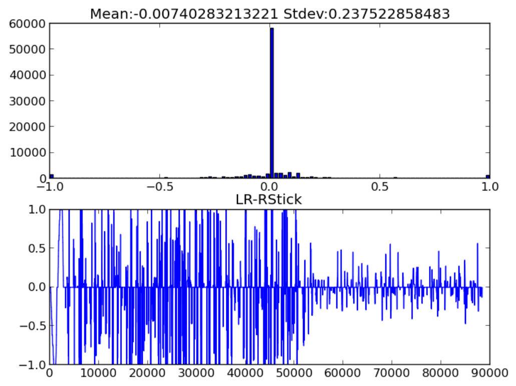


Figure 3: Plot of Raw Joystick Data to determine the distribution of joystick data and characterize the sensor data. The top plot shows the histogram of raw joystick data which ranges from [-1,1] with negative being in the right direction and positive, the left. The bottom plot shows the joystick values over time of all the data sequences.

edges should be when considering more than 3 bins, which is why we decided on 3 bins separated by the sign.

Since most of the data collected is of the Jackrabbot going straight, over 70% of the joystick data is at 0. As a baseline for our model, we'd like it to perform better than a naive model that just went straight no matter what, meaning it'd have to achieve an accuracy $> 71\%$. We redistributed our dataset by removing frames in the straight class to normalize the class distribution to be uniform before training. Table 1 shows the class distribution before and after redistribution. Before normalizing the classes, the point at which the loss would start at would vary wildly between training sessions. After creating a uniform class distribution, the loss started ≈ 1 and the accuracy started at $\approx 33\%$.

$$SanityCheckLoss_{Softmax} = -ln(1/numclasses) = -ln(1/3) = 1.0986$$

	Left	Straight	Right
Before	15%	71%	14%
After	33.3%	33.6%	33.1%

Table 1: Class Distribution Before and After Normalization

To ensure our labels were properly lined up with our images, we plotted the average optical flow of the Jackrabbot's camera feed at each timestep against the labels we assigned to it. The reasoning being that the average optical flow will reflect the direction the Jackrabbot has turned, which should be a reaction to the control mechanism, the joystick. As you can see from Figure 4, the joystick control precedes the flow as expected. To our surprise, however the latency is quite noticeable, lasting over multiple frames, explaining why visualizing via a labeled video was not obvious.

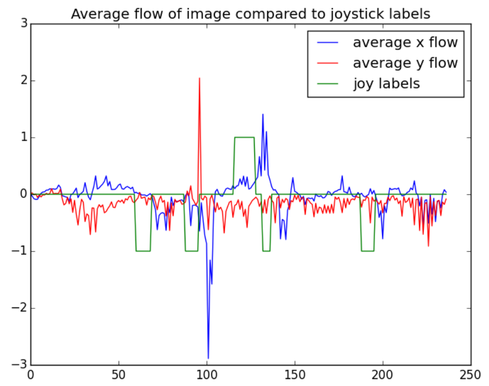


Figure 4: Plot of Labels vs. Average Optical Flow of Image to determine if labels are lined up correctly with each image.

4. Results

4.1. Before Normalization of Class Distribution

Since our learned task depends quite a bit on the movement of obstacles in front of it as well as its own movement, we knew we had to integrate some sort of temporal feature into our inputs. We did this in two ways, using optical flow and stacking our images into 3D images.

For comparison, we used a linear classifier on optical flow images as an additional baseline to the always-go-straight model (70% accuracy). This baseline had trouble generalizing from the training set to the validation set, visible in Figure 6a, and performed well worse than the always-go-straight model. When we put a single greyscale image into a simple two-layer convnet, Figure 6b, the accuracy improved but was still less than always going straight method. Once we introduced the time dimension into the convnet by stacking a sequence with the 5 frames preceding the label, the accuracy of our model increased passed our always-go-straight baseline to 80%. From Figure 6c, the validation accuracy closely follows the training accuracy after a couple hundred iterations, suggesting that the capacity of our model is not large enough and that we should add more data. The learning rate also appears to be too high since the loss levels off and stops decreasing. We also suspected that we needed more data. In an attempt to double our dataset, we augmented our dataset by mirroring the images and flipping the labels, but this resulted in a decrease in accuracy, Figure 6d. As a result, we did not use the augmented data and instead collected more data.

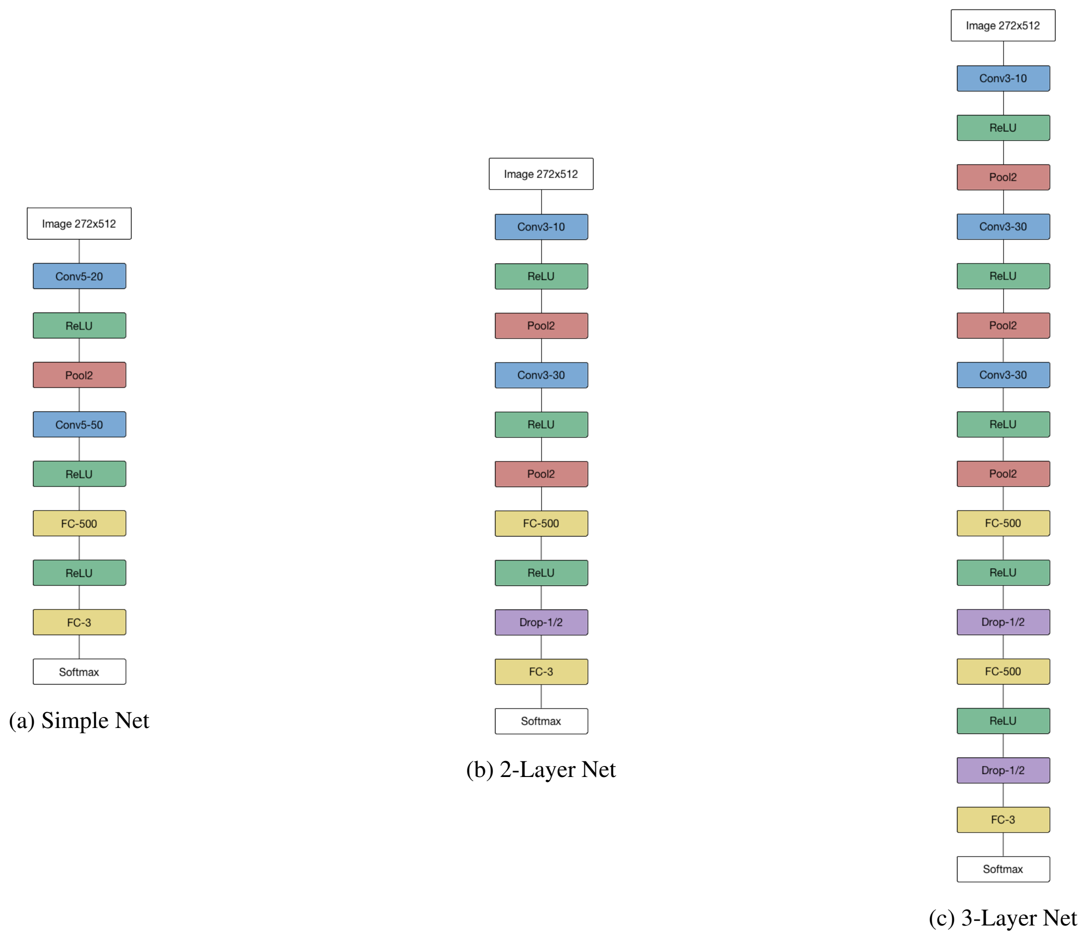


Figure 5: Convolutional Neural Net Architectures

Input Data	Depth	Net	Test Accuracy
Optical Flow + Greyscale	3	Linear Softmax Classifier	51.68%
Single Greyscale	1	Simple Net	70.96%
Stacked Greyscale	5	Simple Net	81.02%
Mirrored Stacked Greyscale	5	Simple Net	67.20%

Table 2: Results Before Normalization of Class Distribution

4.2. After Normalization of Class Distribution

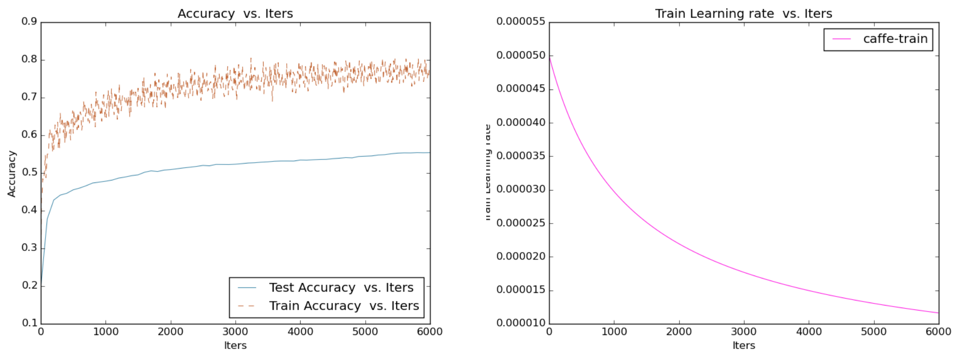
In addition to collecting more data, we normalized the class distributions to a uniform distribution to help the Jackrabbot better learn the left and right classes. This means our always-go-straight baseline now has 33% accuracy. When we trained these on 3-layer convnets, the models seemed to heavily overfit the data, Figure 8a, so we added dropout for regularization. The results from the best nets after normalizing the class probabilities are shown in Table 3.

We used an inverse decay for the learning rate update policy for all our nets.

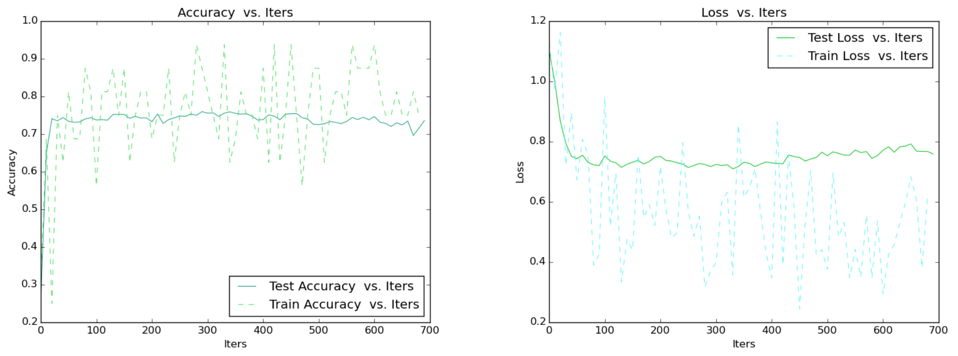
4.3. Best Performance

The model that gave the best performance was the 2-layer convnet with dropout, trained on a 5 consecutive stacked greyscale images. This gave us an accuracy of 65.12%.

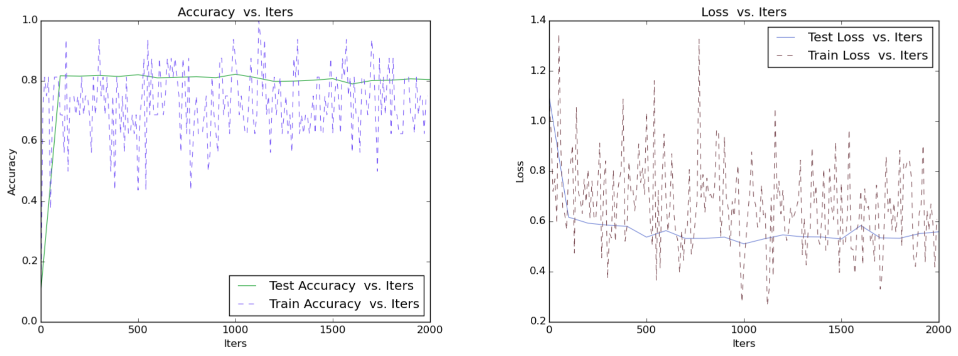
To qualitatively view the performance of our model, we labeled videos of our test sequences with the predicted and correct labels, Figure 9. Even though some images were mislabeled, they still made sense and some were just off by



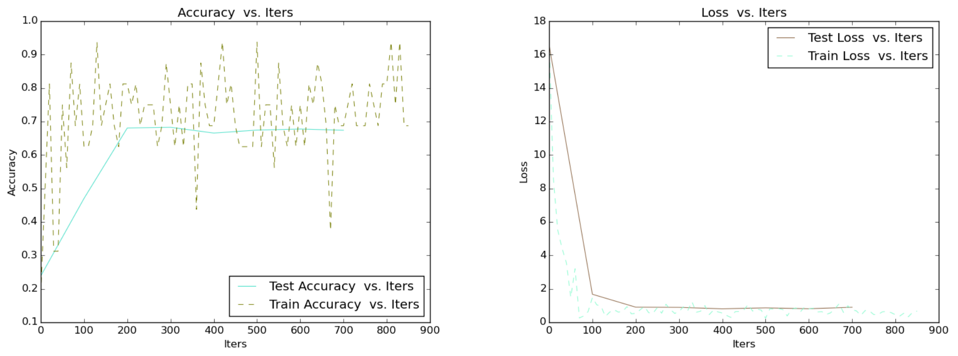
(a) Optical Flow + Greyscale - Linear Softmax Classifier



(b) Single Greyscale - Simple Net

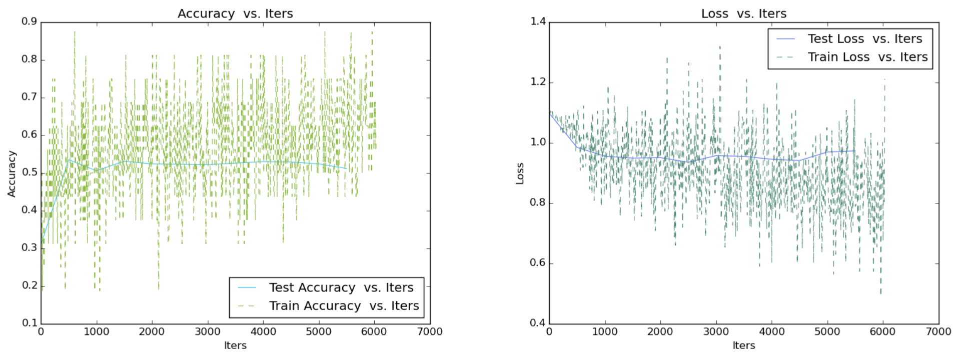


(c) Stacked Greyscale - Simple Net

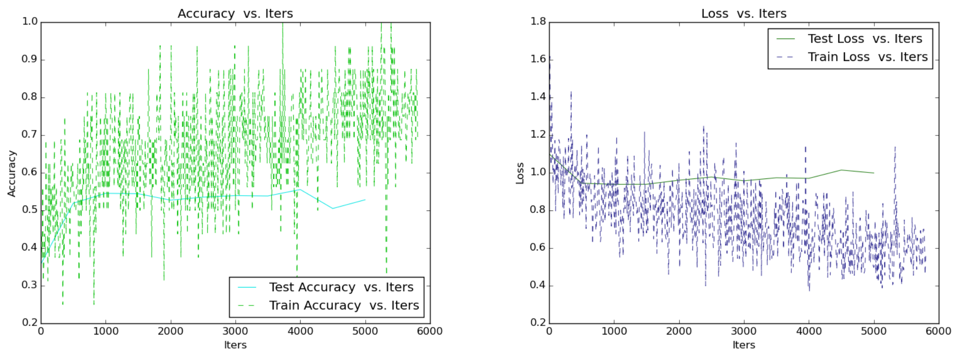


(d) Mirrored Stacked Greyscale - Simple Net

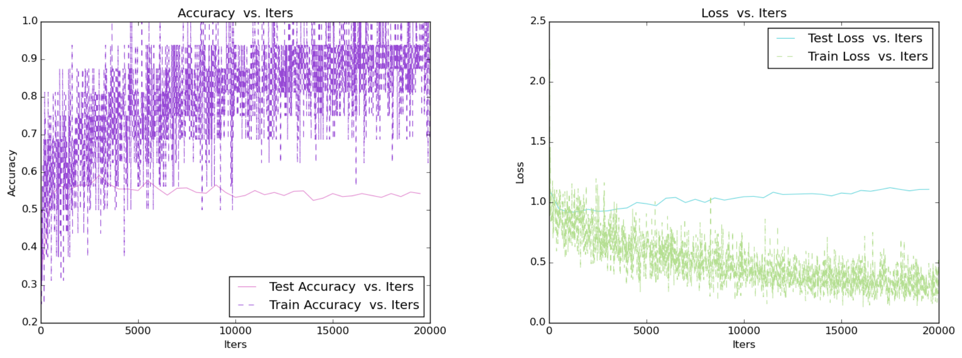
Figure 6: Results Before Normalization of Class Distribution



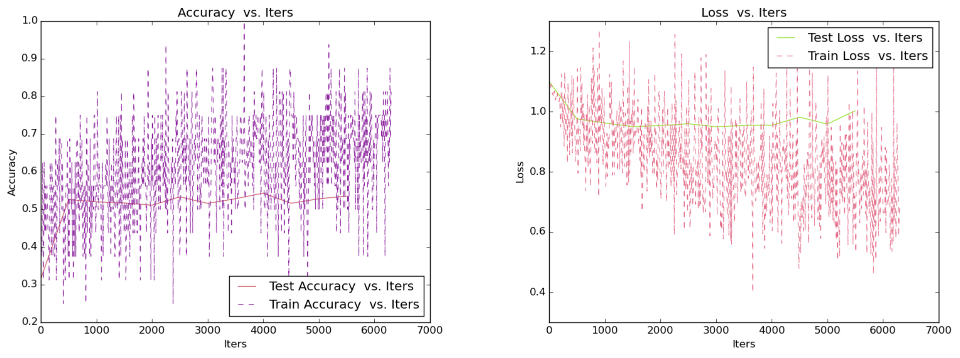
(a) Single Greyscale - 3-Layer Net



(b) Stacked Greyscale - 2-Layer Net



(c) Stacked Greyscale Every Other Frame - 2-Layer Net



(d) Stacked Greyscale Every Other Frame - 3-Layer Net

Figure 8: Results After Normalization of Class Distribution

Input Data	Depth	Net	Test Accuracy
Single Greyscale	1	3-Layer Net	59.95%
Stacked Greyscale	5	2-Layer Net	65.12%
Stacked Greyscale Every Other Frame	5	2-Layer Net	62.47%
Stacked Greyscale Every Other Frame	5	3-Layer Net	54.14%

Table 3: Results after Normalizing Class Distribution

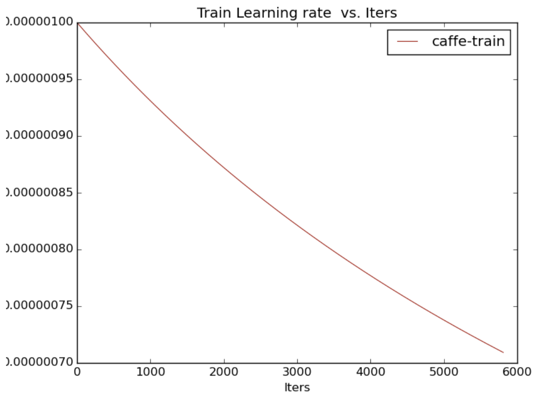


Figure 7: Learning Rate Inverse Decay

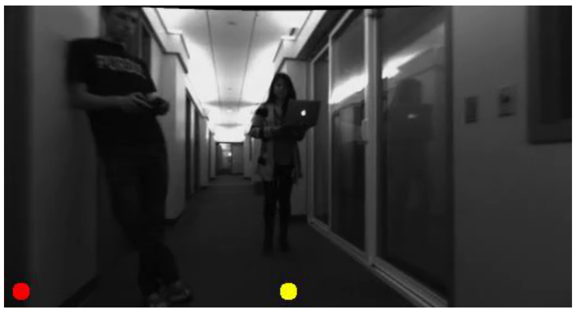


Figure 9: Labeled Input Stream. The green dot represents a correct prediction and the yellow and red dots represent the ground truth and the predicted label respectively in a mislabeled instance. The position of the dots on the input images represent the direction of the joystick data. A dot on the left edge, center, and right edge signify a left turn, straight, and right turn respectively. Link to video: `https://www.youtube.com/watch?v=54lI4STjJQk`.

a frame or two. This could still be acceptable behavior when deployed to the Jackrabbot in real life.

To better understand when there is the discrepancy between the predicted and ground truth labels, we plotted the value of the predicted and ground truth labels over time in Figure 10. Predicted labels seem to be well correlated with

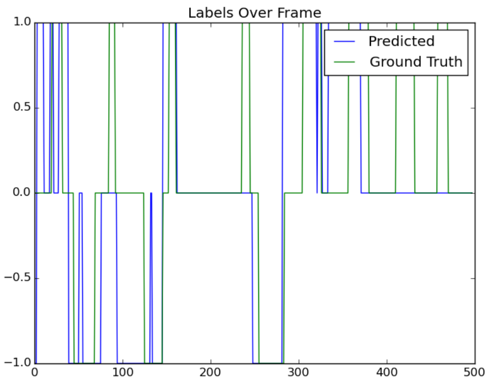


Figure 10: Plot of Predicted Labels vs. Ground Truth Labels Over Time.

Figure 11: Top View of the Predicted and Ground Truth Labels.

the correct ground truth labels with some slightly offset-ted. As you can see, the predicted labels jump around less than the correct labels, indicating this may be desirable behavior that would be low in accuracy.

We also wanted to visualize the physical distance between the predicted and correct trajectories Jackrabbot would take, so we plotted the top view in Figure 11. These trajectories are approximate since our labels are separated into 3 classes. The Jackrabbot was moving at a constant velocity of 0.5 m/s forward in our dataset and the maximum angular velocity was set at 1.571 rad/s. Each label was taken at 10 frames/s. From this, the average angle of rotation was found from the average joystick positive and negative values, ± 0.2, and used to plot the vector at each timestep.

$$(0.2 * 1.571) * 180/\pi = 18 \deg / s = 1.8 \deg / frame$$

5. Conclusion

The results from the 2-Layer net with 5 consecutive greyscale frames stacked as input achieved the highest accuracy of 65.12%. This is 32% better than random guessing. Despite having an accuracy of only 65.12%, from the video stream, we think we are close to being able to deploy and test on the Jackrabbot itself. Even though, we called our human-controlled joystick data as ”ground truth”, there are many paths Jackrabbot could take that would be nearly identical to ”ground truth” and produce the desired behavior. Futhermore, our human-controlled driving is not optimal and varies from driver to driver, so our metric of matching the ground truth labels exactly is indicative but not precise in measuring the optimal behavior.

Stacking the frames and normalizing the class distributions were the most effective, each improving our accuracy by approximately 10%.

We learned that good visualizations are key to making valuable and sensible adjustments to the network. Without them, results can be very misleading and efforts to tweak to hyperparameters fruitless. Quantitative measures, such as accuracy, can also be misleading since skewed datasets to a specific class will give high accuracy for models that do nothing, such as the always-go-straight model. That is why having baselines to reference and having qualitative measures to gauge whether the predictions make sense is really important.

5.1. Future Work and Improvements

Our next steps include deploying this model onto Jackrabbot and characterizing its performance in real time. But before that, we would like to further improve our model by means of more preprocessing of the label data and trying Recurrent Neural Networks for videos.

Most the joystick data is close to zero (Figure 3) since our turns are mostly small turns, which depending on the driver or the sense of urgency, is controlled via small adjustments to the joystick or quick bursts to the edges of the joystick axis. This variability in driver-to-driver data is also apparent in the bottom plot where the second half of the sequences have a smaller amplitude than the first half. We also do not know the exact mapping of joystick control data to the angular velocity of the Jackrabbot. For this reason, we think it would be a good idea to smooth out the joystick data either with a simple moving average or a Kalman filter or use accelerometer data as the labels. This way, the labels would be easier to map and bin to several (> 3) direction vectors. They also wouldn’t jump around so much over time, and two very similar images won’t produce vastly different labels, making it easier to learn.

In the future we would like to also experiment with recurrent neural networks. While we were able to improve performance by including previous frames from the sequence, we believe that it would also be useful to learn output sequences. For example, we should be able to learn that a turn in one direction will be quickly followed by a turn in the opposite direction. Recurrent neural networks have proven useful for tasks such as activity recognition and may be useful here [2].

As for extending this to more complex environments, there is a lot of work that needs to be done. First of all, we need to encode some sense of destination or goal. In our experiments that was given to us because we were trying to go straight and had only one way to get there. In any complex environment, there will be a number of ways to avoid obstacles and reach the destination. This and other challenges will need to be solved for a fully automated system, but we believe this proof of concept is a good first step.

References

[1] Pomerleau, Dean A., ”ALVINN, an autonomous land vehicle in a neural network”, Carnegie Mellon University, 1989. http://repository.cmu.edu/cgi/viewcontent.cgi?article=2874&context=compsci.

[2] Donahue, Jeff and Hendricks, Lisa and Guadarrama, Sergio and Rohrbach, Marcus and Venugopalan, Subhashini and Saenko, Kate and Darrell, Trevor. *Long-term Recurrent Convolutional Networks for Visual Recognition and Description*. CoRR, vol. abs/1411.4389, 2014. http://arxiv.org/abs/1411.4389.

[3] Jia, Yangqing and Shelhamer, Evan and Donahue, Jeff and Karayev, Sergey and Long, Jonathan and Girshick, Ross and Guadarrama, Sergio and Darrell, Trevor. *Caffe: Convolutional Architecture for Fast Feature Embedding*. arXiv preprint arXiv:1408.5093, 2014.

[4] Li, Fei-Fei and Karpathy, Andrej. ”CS231n: Convolutional Neural Networks for Visual Recognition”, 2015. http://cs231n.github.io.

[5] D. A. Forsyth and J. Ponce. *Computer Vision: A Modern Approach (2nd Edition)*. Prentice Hall, 2011.

[6] R. Hartley and A. Zisserman. *Multiple View Geometry in Computer Vision*. Cambridge University Press, 2003. http://searchworks.stanford.edu/view/5628700.

[7] R. Szeliski. *Computer Vision: Algorithms and Applications*. Springer, 2011. http://searchworks.stanford.edu/view/9115177.

[8] D. Hoiem and S. Savarese. ”Representations and Techniques for 3D Object Recognition and Scene Interpretation”, *Synthesis lecture on Artificial Intelligence and Machine Learning*. Morgan Claypool Publishers, 2011. http://searchworks.stanford.edu/view/9379642.

[9] Gary Bradski, Adrian Kaehler. *Learning OpenCV*, O’Reilly Media, 2008. http://searchworks.stanford.edu/view/7734261.