James Butcher - jbutche6@asu.edu - ASU ID: 1220960052

Project 2: Neural Network for Collision Prediction

Introduction

The aim of this project is to train an agent to navigate a virtual environment without collisions. The agent is a virtual "robot" equipped with five distance sensors that measure the distance between itself and an obstacle such as a wall. The robot's brain consists of a neural network that takes these distance readings and the robot's movement action as input and produces a prediction of the likelihood of a collision.

This is a supervised learning problem. In supervised learning, a model learns a function to map an input (X) to an output (Y) by training on a dataset containing the correct outputs for the given inputs. The goal is to approximate that mapping function well enough so that feeding new input data to the model generates the correct output [1]. In this project, the model learns its function well enough so that it correctly predicts a collision or non-collision over 99% of the time.

The Solution

Collecting the data: The first step for training the agent to avoid collisions is to collect a set of data. This was done by allowing the robot to wander randomly throughout the environment and collecting data samples. Each sample consisted of six "X" features (five distance readings plus the action code) and one "Y" output label: a 0 or a 1 representing whether a collision occurred or not.

Since collisions happened much less frequently than non-collisions, I pruned out many of the non-collision samples until the ratio of collisions to non-collisions was 50:50. This was to help avoid the model from only learning how to predict non-collisions while neglecting collisions, since it would predict correctly in the vast majority of cases anyway. This was recommended by the project guidelines. I collected 10,000 raw samples, which reduced to 3,562 samples after pruning.

Training the model: Now that I had the data set, the next step was to split it into a training set and a testing set. I did this with the help of the Pytorch data.random_split function, which splits the data randomly into two groups. I chose an 80:20 split ratio of training to test samples. This was all packaged into a Data_Loaders object, using the Pytorch Dataset module, which automatically loads batches of samples for training.

The training loop ran for 20 epochs using a batch size of 10, a learning rate of 0.01, ReLU as the activation function, torch.nn.MSELoss (Mean Squared Error) as the loss function, and torch.optim.SGD (Stochastic Gradient Descent - Actually, *mini-batch* gradient descent, since the weights are updated after each mini-batch of 10 samples, not after every sample as in SGD [2]) as the optimizer.

Testing the model: Once the model had been trained, it was time to test it out in goal-seeking mode. In this mode, the robot tries to reach goals in the environment while avoiding collisions. If it predicts that a collision is imminent, it will change direction to avoid it, as shown in Figure 1.

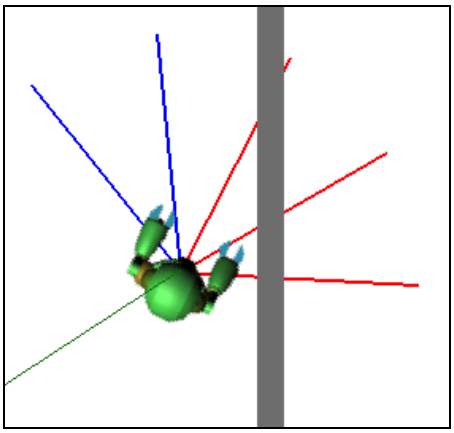


Figure 1: Robot detects that a collision is imminent so it gets ready to turn around, as indicated by the green line.

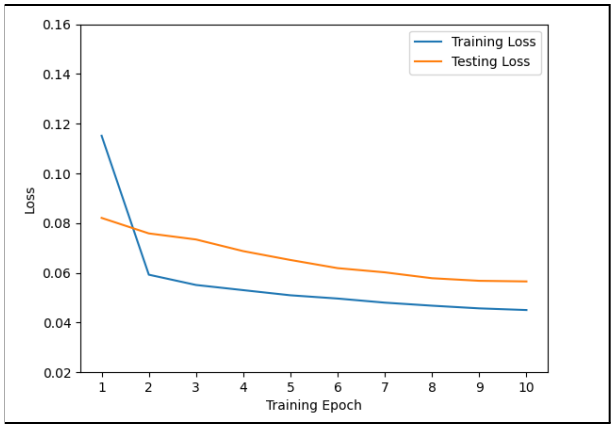
Improving performance: The final grade on the project was penalized for every false negative (a collision) and for every false positive above 10. To get the best possible score, I had to test out different hyperparameters and different choices of network architecture.

Results

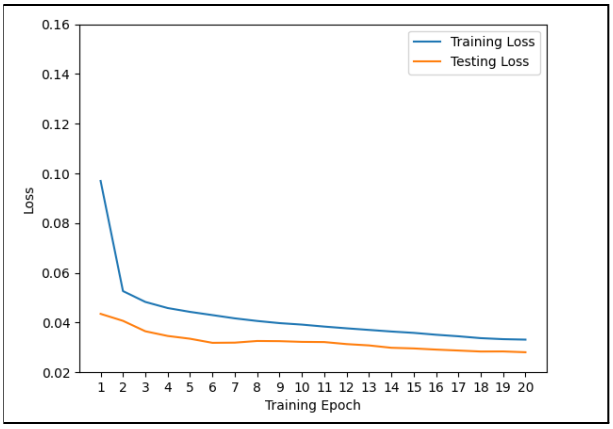
For the first attempt, I chose an architecture with two hidden layers of 10 neurons each. I ran the training loop for 10 epochs. This resulted in 8/1000 false positives and 5/1000 missed collisions. There was clearly room for improvement, especially since the loss plot did not completely flatten out after 10 epochs. This suggests that the model was underfitting the data.

For the second attempt, I decided to increase the number of hidden layers from two to three, and increased the number of neurons in each layer from 10 to 100. This resulted in 8/1000 false positives and only 1/1000 missed collisions. Although increasing the complexity of the model tends to increase the chances of overfitting [3], there was no evidence of overfitting in the loss plot (Figure 2b). This setup resulted in receiving the maximum score on the project.

	1st attempt	2nd attempt
Epochs:	10	20
Number of hidden layers	2	3
Neurons in each hidden layer	10	100
Collisions per 1000	5	1



(a)Attempt 1: 10 epochs, 2 hidden layers, 10 neurons/layer



(b) Attempt 2: 20 epochs, 3 hidden layers, 100 neurons/layer

Figure 2: Training and Testing Losses over 20 training epochs.

Lessons Learned

This was an individual project so all of the work was my own. I gained valuable practice building and optimizing a neural network for a specific task. I learned how to implement mini-batch gradient descent using DataLoaders, which I had never done before. I became more familiar with Pytorch, Numpy, and matplotlib. I gained experience generating a data set and pre-processing it (balancing the distribution of positives and negatives) so that the model learns well.

In the beginning of the project I ran into an unexpected challenge that happened to be very valuable. I had a lot of trouble installing all of the package requirements for the simulated environment. Some worked only on Windows, some worked only on my Mac, some worked only when installed with pip, and some only when installed with Anaconda, but no combination of these was able to get all of the packages to work at the same time. This led me to finally figure out how to set up and use Linux. I spent about a day and a half scouring the web in order to learn how to install Ubuntu on my PC. After overcoming many challenges, I succeeded. And, finally, I was able to get all the packages to run. I am now an official Linux user.

One lesson that really stood out for me related to the actual project was the importance of recording metrics. I needed to keep track of the exact results for each setup and keep them organized, in order to know whether I was making progress or just moving in circles. To do this, I created a text log of the changes I made to the network and the resulting performance. This really helped when it came time to write this report.

Completing this project also helped me gain some insight into how the solution could be improved further. Instead of splitting the data set into two groups - training and testing - I could split it into three groups - training, validation, and testing - in order to get an even better idea of how well the model generalizes. I could also implement k-fold validation so that random imbalances in the data split would cancel out. There are also many additional tricks that could improve learning such as dropout, using the Adam optimizer, LSTM, and so on.

References

[1] J. Brownlee, “Supervised and Unsupervised Machine Learning Algorithms,” Machine Learning Mastery, March, 2016. Available: https://machinelearningmastery.com/supervised-and-unsupervised-machine-learning-algorithms/. [Accessed Nov. 15, 2020]

[2] S. Patrikar, “Batch, Mini Batch & Stochastic Gradient Descent,” Towards Data Science, September, 2019. Available: https://towardsdatascience.com/batch-mini-batch-stochastic-gradient-descent-7a62ecba642a. [Accessed Nov. 15, 2020]

[3] J. Brownlee, “How to Avoid Overfitting in Deep Learning Neural Networks,” Machine Learning Mastery, December, 2018. Available: https://machinelearningmastery.com/introduction-to-regularization-to-reduce-overfitting-and-improve-generalization-error/. [Accessed Nov. 16, 2020]