Collision Detection Neural Net

Aryan Kumar
kumar.aryan@gmail.com/akuma420@asu.edu

***Abstract*—This paper goes into detail about the four part project that was completed during CSE 571, the Artificial Intelligence (AI) class at ASU. Those four parts covered various AI concepts like gathering data appropriately, loading the data and making it useful for a neural net model, creating the model itself, writing effective evaluation functions for that model along with training that model.**

I. Introduction

During my time in Artificial Intelligence, I completed a four part project individually. The projects revolved around the core concepts of Artificial Intelligence and involved the creating a neural net that could train a robot to go through a maze with as few collisions as possible and with as little false positives as possible. Each project involved writing some bit of python code and using python libraries like numpy, PyTorch and sklearn which help facilitate the creation of a neural net by providing functions to easily create datasets along with already created activation functions, loss functions and a MinMaxScaler.

II. Neural Network for Collision Prediction Project

Part 1.

In the first part of this project we had to write code that could collect training data for the simulated environment that would later be used by the neural net to conduct its learning.

The code that we were given to edit already had a loop for a total number of actions and getting the action and steering force. In the loop, I had to code the part that would calculate sensor readings using numpy to append the current sensor reading with the action and collision found in that iteration of the loop and then append those readings to a network parameters array. Once the loop was done I exported that numpy array to a csv file.

Part 2.

The second part of the project we used the generated training data from part 1 and put that into a PyTorch Data loader. A data loader is made by the combination of a dataset and a sampler [3]. It uses both to make an iterable over that dataset. There was also a pruning requirement before using the dataset because if the dataset was mostly 0s it could get good loss but not much can be learned from such a dataset. Overall this step is about appropriately preparing the dataset from part 1 so that useful learning can be done in the next two parts.

In the data loaders class itself we leveraged the PyTorch's random split function to create a train and test loader using the nav dataset. The nav dataset was its own class that took the training data then used a MinMaxScaler to normalize that data. It also had a __len__ function which was a standard method to obtain and return the size of the dataset and a __getitem__ function which to take an index as an input and then return a dictionary with and input and label which got derived from the data, in float 32 format, that is present in the dataset at that index.

Part 3.

In the third part of the project we had to start designing the neural net and define the forward pass through the network. Along with that we also had to create a method to evaluate the fit of the created model. To do the evaluation we were given an evaluate function that had a loader and a loss function as parameters. Then I looped through each sample in the passed in loader and added to a total loss variable to get the entire loss of the whole dataset. Then after the loop I divided that total loss with the length of the dataset to get the average loss and that was returned to be the evaluation of the fit.

For the forward pass portion, we were given a class where we defined the three sections of the neural net and then did the actual forward pass through those layers in a separate forward function. The first section converted from the input layer to the hidden layer, then the second section had a nonlinear activation function and finally the third section went from the hidden layer to the output

layer. The first and third sections used a linear transformation and the second section used the Sigmoid activation function.

In a study titled “Sigmoid Activation Function in Selecting the Best Model of Artificial Neural Networks” Prativi et al. referenced another study that used the sigmoid activation function and was able to get a predictive accuracy rate of 87% [4]. So they believed that the sigmoid activation function could form the best architectural model and were able to produce an accuracy rate of 65% [4]. While typically 70% is the standard for great model performance [5], Prativi et al. did say in their study that it was still the best performing model [4] so I also chose the sigmoid activation for my model.

Part 4.

In the fourth and final part of the project we built on the created neural net from part 3 to train the model. In the train model function we defined batch size for the data loaders, the loss function, learning rate and optimizer.

Batch size determined the amount of samples that got used to update the model’s parameters [1], learning rate let the model know the amount each weight should be updated by [1] and the loss function is used to get an error rate for the model [1]. I used the mean squared error loss functionally simply because it was the one I had also seen in my machine learning class was the most familiar with it and from my research I hadn’t found much evidence that changing it around would produce significantly different results for my model. For batch size and learning rate, I played around with many different values as there didn’t seem to be an exact science on knowing with certainty the exact right values that would produce perfect results.

In also did this train and error approach monitoring the missed collision and false positive numbers with number of epochs. Out of all the previously mentioned parameters to me it felt like the number of epochs was especially important to get right as it determined how much learning was done in a very explicit way. Epochs control the number of times that the algorithm is run on the entire dataset [1]. If the epochs are too high then the model would end up learning too much and that would result in too many false positives but too low then the model wouldn’t learn enough that would lead to too many collisions. I concluded this based on the error numbers as I increased epoch count the false positives went up and as I lowered epoch count typically the missed collisions went up. The epoch count that seemed to work best was about 200, it would’ve been nice to find an exact math that could objectively give the right epoch count but with how varied even datasets of the same size can be it appears to be that the best way is to experiment with as many possible values.

Along with that I went back to my code for part 1 of the project and extracted more training data from there so the model could have a larger initial base to learn from instead of running hundreds of epoch on a smaller dataset leading to too many false positives. To do that , instead of only having the loop to generate data run 1,000 times I upped it to run 50,000 times.

Related to manipulating the training data that got fed to the model, there was another strategy that I implemented that helped get the error numbers lower and that was actual pruning the data to only have the most relevant points be considered when training the model. Initially all I had done was normalize the data but coupling that with pruning right after gave the model a much more useful dataset to learn from. This may seem contradictory as it can initially sound like I got more data then immediately made it less data with this method but the pruning process wasn’t about raw size of data but rather size of relevant data. A model with a million junk data points will not perform as well compared to a model with a couple hundred thousand highly relevant data points.

Finally, I had to decide on an optimizer to use. Optimizers are algorithms that will update the parameter’s of a model in an attempt to keep the loss function as low as possible [1].

I had learned about two optimizers from the lectures, SGD and Adam. When deciding between the two I found a 2021 paper by Gupta et al. that stated that while Adam does have better optimization performance in some situations, its solutions generalize worse than SGD and has a lower test performance [2]. I still tried both SGD and Adam just to see what the difference would be and I consistently got better results with SGD with Adam I was only ever able to get either the false positives low or the missed collisions low but never both. I believe Adam would’ve performed better if the environment that the robot was being sent through changed on each run rather than remaining static as Adam dynamically computes learning rate instead of staying constant like SGD [1].

III. Results and Conclusion

In the end my model resulted in there being 6 false positives out of 1000 and 2 missed collisions out of 1000. These numbers were within the acceptable range for total amount of error for the project.

In conclusion the project was a success in terms of not being riddled with false positives and collisions and through trail and error I was able to settle on a workable activation function along with good batch size and learning rate values and verify that the sigmoid activation function and SGD optimizer did seem to trend towards the best results verifying the findings of two studies.

The project was also a success by the standards of what constitutes a great machine learning model. Out of the 2000 inputs only 8 of them were wrong effectively giving my model a prediction rate of 99.6%. For the most part, only accuracy rates at about 90% are seen as realistic any higher isn't typically expected to be possible [5]. Of course that rate isn't something that can be officially stated as the accuracy rate of my model overall as it was only tested on the one environment but it is an indication that it would with the category of great performing learning models.

Acknowledgment

Thanks to professor Taher, assistant professor Amor. Yang, Luca and Strivastava for the detailed and informative lectures that gave me optimal understanding needed to complete each part of the project.

References

1. A. Gupta "A comprehensive guide on Optimizers in deep learning," Analytics Vidhya, https://www.analyticsvidhya.com/blog/2021/10/a-comprehensive-guide-on-deep-learning-optimizers/#:~:text=Unlike%20SGD%2C%20which%20maintains%20a,such%20as%20AdaGrad%20and%20RMSProp. (accessed Dec. 22, 2023).

2. A. Guta et al, "Adam vs. SGD: Closing the generalization gap on image classification," 13th Annual Workshop on Optimization for Machine Learning, pp. 1-7, 2021.

3. A. Paszke, S. Gross, S. Chintala, and G. Chanan, "Torch.utils.data," torch.utils.data - PyTorch 2.1 documentation, https://pytorch.org/docs/stable/data.html (accessed Dec. 22, 2023).

4. H. Pratiwi *et al.*, "Sigmoid activation function in selecting the best model of Artificial Neural Networks," *Journal of Physics: Conference Series*, vol. 1471, no. 1, pp. 1–8, 2020. doi:10.1088/1742-6596/1471/1/012010

5. K. Barkved, ""How to know if your machine learning model has good performance: Obviously ai," Data Science without Code, https://www.obviously.ai/post/machine-learning-model-performance (accessed Dec. 22, 2023).