CSE 571-Artificial Intelligence Portfolio

SAILENDRI GUNNIA RAVIKUMAR
ID:1229372289
sgunniar@asu.edu

***Abstract*—This is a portfolio for Artificial Intelligence course and it focuses on the collision prediction of a robot using the data from the sensor. This project is divided into four parts namely Data collection, Data preprocessing and creating data loaders, Preparing a custom neural network and Collision prediction. The final goal is to predict collision. Details were available in Overview document[1]**

Keywords—Prediction, Neural networks, Data_loaders,Data separation, Train and Test data, Feed forward neural network .

I. INTRODUCTION

The first phase of this project centers around the acquisition of data, a fundamental step facilitated by the Simulation Environment and Steering Behaviors Python scripts provided. These scripts orchestrate the movement of the robot, enabling it to traverse its environment for a specified number of actions. Crucially, the sensors' readings, the action taken at each instance, and the occurrence of collisions are meticulously recorded. The culmination of these data points is systematically stored in a database, forming the foundation for subsequent phases.

The subsequent stage of the project is dedicated to the normalization of the data and its subsequent integration into separate data loaders, strategically designed for future applications. The choice of batch size assumes significance in this context, influencing the efficiency of data processing. Moreover, the meticulous typecasting of normalized data contributes to the overall cohesiveness of the dataset

The third facet of the project involves the development of a feed-forward neural network, an architecture designed to harness the comprehensive capabilities of the model. This network encapsulates crucial functions such as the init function for initialization, the forward function to dictate the sequence of operations during the forward pass, and the evaluate function, in assessing the loss incurred during model evaluation.

The conclusive segment of the project pivots on the evaluation of predictions for the test dataset, leveraging the powers of the trained model. This evaluation involves a meticulous comparison of the model's predictions with the ground truth labels, culminating in the computation of training loss and accuracy, as well as testing loss and accuracy. The objective is to minimize instances of missed collisions and false positives, instances where the model erroneously identifies a non-collision event as a collision during its predictive processes. This meticulous evaluation process is integral to refining and enhancing the model's predictive capabilities.

II. EXPLANATION OF SOLUTION

A. Part 1

The process begins by collecting data using two scripts named "SteeringBehaviours" and "SimulationEnvironment." These scripts serve different purposes in the context of controlling a robot within an environment. The "SteeringBehaviours" script is responsible for directing the robot to move in random directions, halting upon collision with a wall, and then choosing a new direction. The robot continues to move until a specified number of actions is reached. On the other hand, the "SimulationEnvironment" script is used to define the environment where the robot can roam, with the walls serving as boundaries. When a collision occurs with a wall, the robot changes color to pink, indicating the collision event.

To collect the data, I used a new script importing the above mentioned scripts[1]. We have a main function where I mentioned the total number of actions and the results of user defined function collect_training_data is stored. The results are converted to a pandas dataframe and stored in a csv file, in the same directory.

The core function for data collection, "collect_training_data," takes the total number of actions as input. Inside this function, two variables are created for the "sim.SimulationEnvironment()" object and the "wander" function. A total of seven parameters are saved during this process. The first five values correspond to sensor readings, providing information about the environment. The sixth value represents the action taken based on those readings. Lastly, the seventh value indicates whether a collision occurred (1) or not (0), serving as the prediction for the given set of sensor readings and actions.

B. Part 2

In this segment, we leverage the CSV file generated in part 1, comprising approximately 10,000 records. Within this dataset, we observe a distribution of around 30 percent for collided samples and 70 percent for non-collided samples. My task involved configuring a main function in which I explicitly set the batch size to 16 and invoked the Data_loaders class.

The Data_loaders class, taking the batch size as its parameter, in turn, invokes the Nav_Dataset class. Within the Nav_Dataset class, the CSV file is read, and the values are extracted. To ensure the data's consistency and comparability, the MinMaxScaler is employed to both fit and transform the values into normalized data.

The Nav_Dataset class encompasses two additional functions. Firstly, one function determines the length of the dataset. The second function focuses on extracting each row from the normalized data, storing it in a dictionary format denoted as input:label. For instance, the input comprises the first six values of a row, while the last value is designated as the label. To maintain precision, both the input and the label are typecasted into float32. Following this data preprocessing phase,I made an attempt to partition the data into training and testing sets, allocating a test size of 20 percent. Leveraging the DataLoader utility, the train_loader is constructed with the training data, utilizing the specified batch size and setting

shuffle=True for optimal randomness. Similarly, using DataLoader for the test data, the test loader is configured. The train_test_split() method from the sklearn package is used in effecting the separation of train and test data. Using for given loops, I was able to visualize how the values are stored in the data loader.

C. Part 3

This part deals with the construction of custom neural networks. The main function calls a single class and as we use pytorch, we use the nn.Module[2]. The init function is meticulously defined with the input size set to 6, hidden size to 180, and output size to 1.

To articulate the design, we employ two distinct linear layers, each serving a unique purpose. The first layer facilitates the transition from the input to the hidden layer, with input_size specifying the number of in_features and hidden size dictating the out_features. The second layer, in turn, utilizes hidden size as in_features and output size as out_features. To introduce non-linearity into the network, a pivotal role is played by the ReLU()[3] activation layer.

Having established these layers within the init function, their orchestrated utilization is articulated in the forward function, following a sequential order. The progression involves:

Output = Input to Hidden layer(input)

Output_1 = Non linear Activation (Output)

Final_output = Hidden to Output layer (Output_1)

Subsequently, the evaluate function takes center stage, accepting the model, test_loader, and loss_function as inputs. For each value within the test loader, the input and label values are extracted. The input is then passed through the forward function, generating both the predicted value and the label. This pair is subsequently presented to the loss function. The evaluation process culminates in the computation of the Testing Loss and Testing Accuracy. The Testing Loss is derived by dividing the total loss by the length of the test data. Meanwhile, the Testing Accuracy is computed as the ratio of samples predicted correctly to the total number of samples, providing a comprehensive measure of the model's performance on the test dataset.

D. Part 4

In this phase, the focus shifts towards the training of the neural network model, which was meticulously designed in part 3. To facilitate this process, a dedicated script is introduced, importing the essential components of the data loader and the neural network. Within this framework, a set of hyperparameters assumes a pivotal role, and distinct combinations of these values can yield varying models with unique training outcomes.

In this specific instance, a batch size of 16 is chosen[4], and the data loaders are crafted accordingly, leveraging this batch size. The feed-forward neural network is then invoked, resulting in the instantiation of the model. To gauge the model's performance during training, the Mean Squared Error (MSE) Loss function is employed. Another critical hyperparameter, the learning rate, is set at 0.01[4]. The Adam Optimizer[3] dynamically utilizes this learning rate in conjunction with the model parameters to iteratively update the gradients, steering the model towards convergence.

Across the stipulated number of epochs, the training unfolds as follows. For each batch of samples within the data loader, the input and label are segregated into distinct variables. The model is then tasked with predicting the output for each input value, and the loss function calculates the discrepancy between the predicted value and the actual label. Subsequently, through the process of backpropagation, the gradients are updated, refining the model's parameters.

Concurrently, metrics such as training accuracy and training loss are computed, mirroring the methodology applied in assessing testing loss and accuracy. The evaluate function is then enlisted to calculate the test loss and test accuracy, providing insights into the model's generalization capabilities

As a culmination of the training phase, the trained model is preserved by saving it as a pickle file through the torch.save() function[2]. To verify the model's efficacy, a purpose-built Python script for goal-seeking is provided, offering an additional layer of validation and ensuring the model performs optimally in diverse scenarios.

III. Description Of Results

A. Part 1

Through the integration of the SteeringBehaviors and SimulationEnvironment scripts, a meticulous process unfolds, resulting in the systematic storage of collected data. The structured order of this data encompasses a comprehensive set of attributes, each playing a distinct role in encapsulating the essence of the robot's interaction with its environment. The sequence unfolds as Sensor reading 1, Sensor reading 2, Sensor reading 3, Sensor reading 4, Sensor reading 5, Action taken, Result (collided (1) or not (0)).

This structured arrangement serves as a comprehensive representation of the robot's sensor readings, the action it takes in response, and the consequential outcome, particularly whether a collision event has occurred (1) or not (0).

The training_data.csv file serves as a database having the sequential arrangement of sensor readings, actions, and collision outcomes. This file serves as a key reference point, allowing for a more nuanced understanding of the data's structure and significance.

I was not able to achieve balancing of data, at first. As the collected data had more of non-collision instances, even though the robot experienced a collision, it was not recognized and we had more number of missed collisions. To correct this, data pruning of non-collision data has been done manually. By doing so, it was helpful in further parts to train neural network with almost equal amount of collision and non-collision instances.

B. Part 2

Upon the normalization of the data, the utilization of a data loader introduces a structured and iterative approach to processing the information. The designated batch size, set at 16 in this instance, results in the organization of the data into numerous distinct set of tensors. This batch-wise organization ensures a systematic and efficient handling of the dataset, optimizing the computational processes involved.

A similar procedural approach is undertaken for the test input and test labels, a meticulous process that involves

traversing through the data loaders. The careful implementation of typecasting assumes more significance in this context, as an oversight in this step could jeopardize the attainment of such systematically formatted results. The application of typecasting is vital in ensuring that the data maintains its intended structure, adhering to the prescribed format. I have similarly sorted out for test input and test labels also by looping through the data loaders. If the typecasting is not done properly, we may not be able to achieve such formatted results. This format proves to be particularly advantageous in the separation and organization of the training and testing datasets. The dictionary datatype is used to store key value pairs for each row of data. This format facilitates the separation of train and test dataset.

So, as a result, we will be having the batches of inputs and labels of the train data and test data. This kind of data loading helped to speed up the process as we have nearly 10000 instances altogether, instead of lagging in time by loading instances one by one.

C. Part 3

This particular phase of the project focuses exclusively on the construction of the neural network architecture, with no emphasis on conducting tests or assessments. The formulation of the neural network is meticulously detailed. However, it's imperative to note that, at this juncture, no testing or evaluation has been initiated.

I initialized the neural network with three types of layers, Input to hidden layer, a non linear activation layer and a hidden to output layer. We can usually align the initialized layers using the forward function. This function governs the step-by-step progression of data from the input layer through the hidden layers, incorporating non-linear transformations, and ultimately leading to the output layer. This sequential arrangement ensures that the network can process and interpret input data in a meaningful way. These layers, when properly configured and initialized, form the structural backbone of the neural network. The number of hidden layers was fixed during the training and testing phase because, for a fixed number of epochs and learning rate, the accuracy varied drastically due to the recognition of more number of missed collisions by increasing the hidden layers count[4]. Finally I fixed it to 180 hidden layers.

The instantiation of the neural network, lays the groundwork for subsequent stages where the feed-forward neural network and the data loaders are brought into play. It's essential to recognize that, at this juncture, the model remains untrained, and only a scalar.pkl file is available. Despite the existence of the evaluate function, its utility is currently limited, as testing and accuracy assessments cannot be effectively carried out without first undergoing the crucial training phase.

D. Part 4

In the conclusive phase of the project, a noteworthy achievement was reached as I successfully extracted both training accuracy and testing accuracy for each epoch. An intriguing observation emerged during this process: with an increase in the number of epochs, the test loss demonstrated a corresponding increase. This phenomenon underscores the delicate balance required in model training, prompting a nuanced strategy such as early stopping to discern the optimal number of epochs[4].

By properly setting the epoch number, a meticulous calculation of accuracies was executed. This approach proved instrumental not only in achieving commendable training accuracy but also in safeguarding against the pitfalls of both overfitting and underfitting[4]. The delicate equilibrium achieved through this methodology contributes significantly to the model's generalization capabilities, ensuring its proficiency in handling new and unseen data beyond the training set. The strategic prevention of overfitting and underfitting further attests to the robustness and efficacy of the implemented model. Testing of the model with the goal seeking script, also confirmed the model works perfectly.

IV. Contributions

These are individual projects, and I started the project with the course videos [2] for the projects and the documentation [1] helped me to pursue the projects in right manner.I have set the technology requirements to the versions as mentioned. For example, Python-3.10.9, Matplotlib-3.7.1, Scipy-1.8.1, Pymunk-5.7.0, Pillow-9.5.0, Torch-2.1.0., etc. For this, I created a new environment in Anaconda with all the above-mentioned packages and versions. Initially, I was taking time to understand the Whole Collision Prediction System. Then I realized pytorch has wide range of in-built functions that can be used on every part of the project[2]. By using the concepts mentioned in the live events, I was able to debug and figure out wherever I got errors and unexpected behaviors. I understood pruning and balancing of data is vital for a model to work perfectly.Without proper data modification, eventhough my model has the ability to perform correctly, it failed. Having balanced data (50% collided samples and 50% non collided samples), considers both type of samples and gives importance equally.

V. Lessons Learned

First, I learned the setup of new environment in Anaconda, as we usually work on the base or root environment. In this new environment, I had to upgrade and few packages, I degraded few packages.

The framework's diverse array of functions has not only broadened my knowledge but also instilled a sense of comfort in navigating its intricacies. I find great convenience in employing the built-in models, and the process of constructing a neural network has become notably straightforward. By adeptly utilizing the various modules and functions provided by PyTorch, the task of building a neural network is simplified, making the entire process more accessible and user-friendly. This newfound ease of use enhances my overall confidence in leveraging PyTorch for future endeavors in model development and machine learning applications.

We should build our models such that it does not overfit on the data causing high generalization error and also it should not underfit the data causing high training error.

I learnt switching between different environments of anaconda. The csv files used as the dataset, should be in the same folder as the ‘ipynb’ file. The pickle file and the training data file containing results are also stored on the same folder.

By end of these projects, I learnt to do data preprocessing, employing the necessary activation

functions[3], creating a neural network from scratch and its minimum requirements, split train and test data, Check their discrepancies thoroughly.

REFERENCES

[1]https://www.coursera.org/learn/cse571artificialintelligence/home/week/1

[2]https://pytorch.org/tutorials/beginner/pytorch_with_examples.html

[3]https://builtin.com/data-science/feedforward-neural-network-intro

[4] https://www.geeksforgeeks.org/hyperparameter-tuning/