Real-Time Anomaly Detection on Streaming Race Car Data

Yashvardhan Jain
Computer Science
Indiana University - Bloomington
Bloomington, IN
yashjain@iu.edu

Vrinda Mathur
Data Science
Indiana University - Bloomington
Bloomington, IN
vrmath@iu.edu

INTRODUCTION

The IndyCar Series is the premier level of open-wheel racing in North America [4]. The series offers its international lineup of drivers to compete in the most challenging of motorsports. Part of the team is a dedicated group of researchers who are working to improve the safety and performance of race car driving. Motivated by this and for the greater purpose of safety in transportation, anomaly detection is the chosen topic of research. This research may be applied to prevent accidents in everyday traffic having different types of vehicles.

Our objectives for the research project included:

1. Working with large datasets that simulated a real-world dataset in terms of signal and noise.
2. Since models trained on large time-series datasets on distributed systems are difficult to implement, understanding the implementation of distributed training of machine learning models is an important objective.
3. And lastly, detect anomalies in real-time with low-inference latencies since currently, the real-time detection is slow when it needs to be fast as well as accurate.

To achieve the above goals, we used distributed training in Pytorch and inference-time data streaming in Apache Storm for our research. We use DistributedDataParellel class for distributed training in Pytorch. It implements distributed data parallelism by splitting the input across the specified devices, parallelizing computations across processes and clusters of machines. Apache Storm is a distributed real-time computation system that makes it easy to process unbounded streams of data for real-time data processing [2].

ARCHITECTURE/IMPLEMENTATION

1 Data cleaning and data preprocessing

Data was received from a log file that contained telemetry data about car number, vehicle speed, engine speed, lap distance, brake, steering, throttle, gear, acceleration, tire type. 33 cars participated in the race and data stored for each of them, for every feature at every millisecond. This brought the total number of data points to 4,099,635. As part of the data manipulation:

1. The dataset was reduced down to store 1 data point per second by taking an average of the values of the features.
2. Removed data that measured the various features before the race began

As part of the initial analysis, we visualized each feature individually. As an example, in Figure 1, it is easy to see which cars completed the race and which did not. The plotting of each feature, for each car over time, corroborates this.

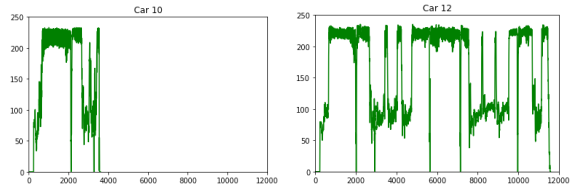

Figure 1(a) : Speed data

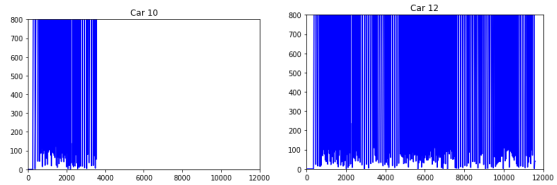

Figure 1(a) : Distance data

To build our univariate time series model, we used the speed data. It was restructured to have speed values for a single car in a column while each row stored the speed of the car at that second. This dataset stored 11585 data points each for 33 cars. Further in data cleaning:

1. Multiple cars were removed from the dataset having a large proportion of NA values.
2. The first 227 seconds worth of data was removed as it was mostly NA values.
3. Our final dataset contained the speed data of 24 cars having speed data for 11358 seconds.

2 Long short-term memory (LSTM) model

Since we are working with time-series data, we need to deploy a machine learning model that can exploit the temporal dynamics of the data. For this reason, we use a Recurrent Neural Network, specifically an LSTM (Long short-term memory) model. We built an LSTM model, having 2 LSTM layers, each with 128 hidden units and 0.5 dropout, followed by one Linear layer having a single neuron. Using the Adam Optimizer, L2 (Mean Squared Error) loss, and a learning rate of 0.0001, a suitable model was built that explained the time series data. The model was varied by varying the timestep between 80, 120, 160. It is important to note that we used a stateless LSTM model implying the hidden and the cell states were reset to zero after every batch. This leads to faster training.

3 Distributed Training in PyTorch

PyTorch provides distributed training through its PyTorch.Distributed module, which enables parallelizing model training across multiple GPUs on the same machine or even across multiple machines. The distributed training of a model can be parallelized across multiple GPUs on a single machine, multiple machines (each with a single GPU or no GPU), and multiple machines (each with multiple GPUs). PyTorch leverages message-passing semantics allowing each process to communicate data with all other processes by enabling them to share their locations. The processes that are spawned as a result of distributed training are blocking in nature, hence each node/process waits for the other nodes/processes to complete a batch of training before continuing with the next batch of training. One issue that can arise with distributed training across multiple machines in a cluster can be the high data transfer latencies between the nodes during training. Since all the processes on all the nodes in a cluster need to be properly synchronized, they might need to share various data between them multiple times during the training, such as gradients, parameters, input data, etc. This data transfer can lead to high latency between batches as well as a high data transfer bill on cloud systems. PyTorch solves this issue by implementing the synchronization of models across nodes in a very specific way. Instead of sharing the gradients between the nodes, PyTorch averages the gradients between the nodes so that each model is in sync in terms of its learning capability. Since all the gradients are averaged across all the nodes in the distributed training cluster, all models learn identically. Furthermore, the data input needs to be present on all the nodes and the data is partitioned so that different nodes don't train on the same batches of data. PyTorch also provides lower-level methods for more fine-tuned control over how distributed training is implemented which enables users to implement their own distributed training algorithms. PyTorch provides support for multiple communication backends such as gloo (supports both CPU and GPU training), nccl (supports only GPU training and is better than the GPU implementation of gloo), and mpi (supports both CPU and GPU training but needs to be built from source on the host machines). For our purposes, we use gloo for the CPU trained models and nccl for GPU trained models. The models are trained on 2 AWS EC2 (p2.xlarge) instances which have a single Tesla k80 GPU each. For distributed training, PyTorch provides a command-line launch utility for distributed training called torch.distributed.launch which is used to launch the .py scripts that implement distributed training. The different processes/nodes are distinguished by a rank attribute assigned to each node/process in the distributed cluster. The launch utility takes the master node IP, master node port, the rank of the node, number of nodes, and number of processes per node as command line arguments when launching the script. Our experiment setup consists of 4 different setups: single-machine only-CPU, single-machine single-GPU, multi-machine only-CPU, multi-machine single-GPU. For the multi-machine setups, we use 2 machines. We calculate the training time, MSE value on validation data, MSE value on test data, RMSE value on validation data, and RMSE value on test data for each setup. Since CPU training takes a

long time, we only train the CPU setups for a single epoch. The GPU setups are trained for 50 epochs. The model with the best MSE/RMSE value is then finally used for real-time inference on Apache Storm.

4 Apache Storm

Apache Storm is a real-time data processing system that allows reliably processing unbounded streams of data with no latency. It provides guaranteed message passing and fault tolerance. Storm requires zookeeper as a dependency, which provides highly reliable distributed coordination. Storm includes 3 main concepts namely, spouts, bolts, and topology. Spouts are responsible for generating streams of data and bolts are responsible for processing the data that was generated from the spouts. All the spouts and bolts defined in a system create a connected computation graph, which is called the topology of the storm system. We use storm for real-time inference on the streaming speed data for anomaly detection. We define a spout that reads the preprocessed speed data of all race cars and streams it to the bolts. We define two bolts in our system. One bolt is responsible to process the data and create a visualization of the streaming data in a web browser. The second bolt contains our trained PyTorch model and does inference on the streaming data and sends it to the first bolt for visualization. We visualize the speed on the y-axis and timestep on the x-axis. We show anomaly scores for each prediction as a bar. If the anomaly score is higher than a threshold (220 in our case), then that is visualized as red, otherwise, it is visualized as blue.

Hence, we are able to tackle different aspects of real-time machine learning model deployment that includes data cleaning and preprocessing, distributed training of machine learning models, and real-time inference on streaming data.

EXPERIMENTS

1 The LSTM model

The experiments were conducted using the speed data, structured to have all the cars that have completed the race as columns and 1 record per second in each row. The dataset was divided into training, validation, and testing datasets having a division of 40-10-50 % The first task was to build a suitable model that explained the fluctuations in speed with the time that the racecar data provided. We built multiple LSTM models, which were varied by varying the timestep between 80, 120, 160 but visually showed no improvements.

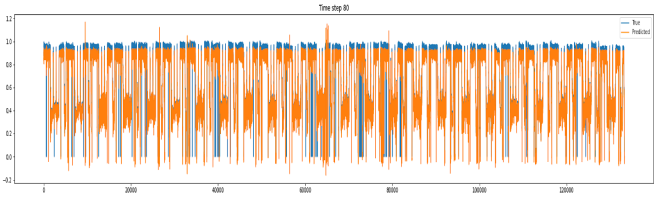

Figure 2(a) : LSTM model for timestep 80

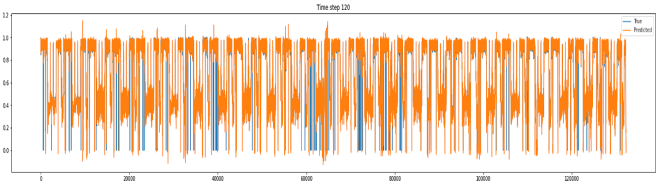

Figure 2(b) : LSTM model for timestep 120

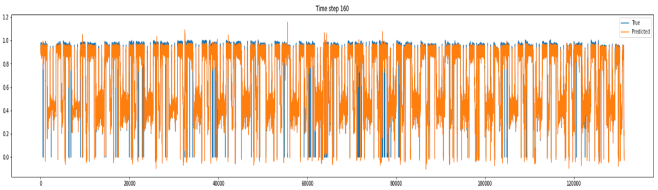

Figure 2(c) : LSTM model for timestep 160

On the calculation of the RMSE (Root Mean Squared Error) and MSE, it is indicated that timestep 160 is the best model having had the lowest errors as seen from Table 1.

Timestep	RMSE	MSE
80	0.0437	0.0023
120	0.0375	0.0030
160	0.0344	0.0021

Table 1: RMSE and MSE of LSTM model for different timesteps.

The anomalies were detected based on the difference between the predicted speed value and the true speed value. Keeping a threshold of 0.045, if the absolute value of the difference is greater than the threshold then it was marked as an anomaly. Hence, the anomaly detection is highly dependent on how well the anomaly threshold is set, which can be tuned to different

values based on tolerable error for the particular use-case.

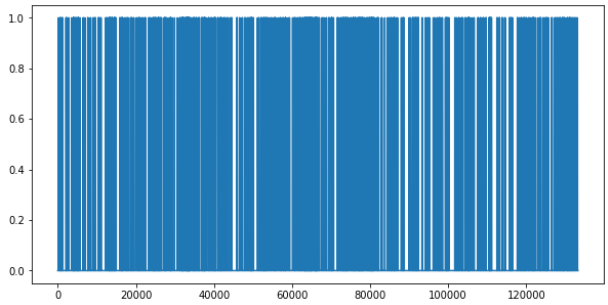

Figure 3: Anomaly Result

2 Distributed Training on Pytorch

As part of the distributed training, 4 experiments were conducted in the form of the following training setups :

1. Single-node CPU
2. Single-node GPU
3. Multi-node CPU
4. Multi-node GPU

The LSTM models with timestep 80 and 160 were used for each of the above setups. The models were trained on AWS EC2 p2.xlarge instances (each with 1 Tesla k80 GPU) to conduct experiments using the multi-GPU resource. The CPU versions were trained only for a single epoch while the GPU versions were trained on 50 epochs. Hence, the training time values in Table 2 show total training time for 1 epoch (in case of CPU) and 50 epoch (in case of GPU). The time is given in seconds. For multi node setups, we show the total training time for both nodes. The multi-node GPU models show the lowest MSE/RMSE values and we don't see much difference in MSE/RMSE between the multi-node GPU models trained 80 timesteps vs 160 timesteps. Both versions have an MSE value of 0.0011, although the model on 160 timesteps shows a slightly lower RMSE value of 0.0261.

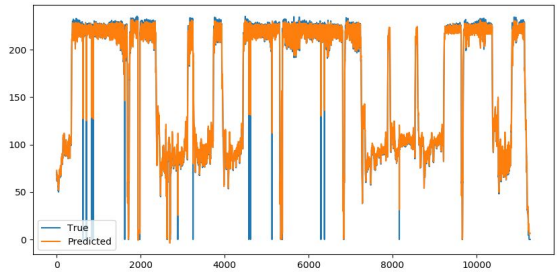

Figure 4: True vs Predicted Speed on 80 timesteps trained on 2 nodes with GPUs

Timestep	Metric	Single node CPU (1 epoch)	Single node GPU (50 epochs)	Multi node CPU (1 epoch)	Multi node GPU (50 epochs)
80	**Training time**	223.3	679.33	123.97/ 126.68	401.44 /399.8
80	**RMSE**	0.0621	0.0303	0.066	0.027
80	**MSE**	0.004	0.0013	0.0051	0.0011
160	**Training time**	799.43	1296.02	383.4/ 388.39	760.31/ 758.9
160	**RMSE**	0.0727	0.0339	0.0669	0.0261
160	**MSE**	0.0058	0.0016	0.0052	0.0011

Table 2: Results of Distributed Training

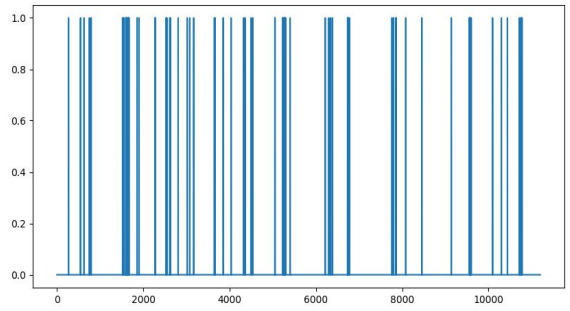

Figure 5: Anomaly Detection on 160 timesteps trained on 2 nodes with GPUs

3 Apache Storm for Real-time Streaming

Another important task for the real-time streaming of anomaly detection. For this, we used Apache Storm. We visualize the speed on the y-axis and timestep on the x-axis. We show anomaly scores for each prediction as a bar. If the anomaly score is higher than a threshold (220

in our case), then that is visualized as red, otherwise, it is visualized as blue. The model that is trained on multi-node GPU setup on 80 timesteps is used for inference since it had the lowest MSE value.

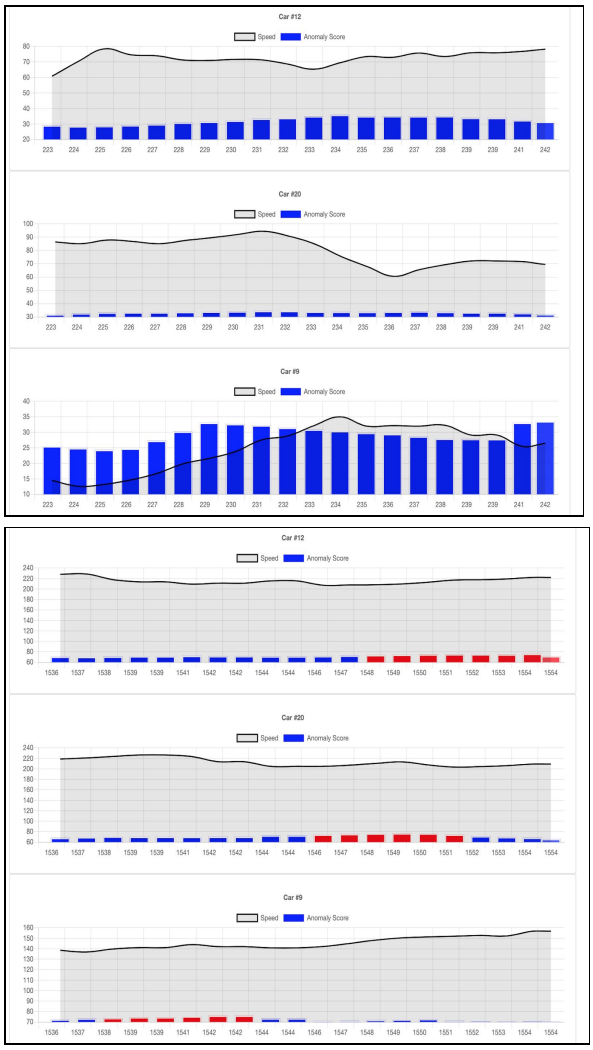

Figure 6: Real-time anomalies on Storm (red)

CONCLUSION

Real-world data is noisy and complex and efficient distributed training is crucial for deploying real-time machine learning models. This research has been a good foundation for fast and accurate anomaly detection but there are multiple things that can be done further to enhance the implementation. Telemetry data can be combined with the video streaming data to create better anomaly detection models. Using millisecond data instead of per second data may prove to be beneficial, for the detection may then be more accurate. For faster training, research can also explore a multi-machine-multi-GPU setup and try to advance, along the lines of integrating distributed training and inference into a single real-time data streaming pipeline. Furthermore, the data could be labeled manually to create an anomaly classification system as well, after an anomaly is detected.

In conclusion, efficient distributed training and improvements in real-time machine learning models are crucial and can be extremely beneficial for various industry applications.

ACKNOWLEDGEMENTS

We would like to thank our professor, Dr. Judy Fox, for imparting knowledge about deep learning and high-performance computing. We appreciate our associate instructor, Selahattin Akkas for providing us with the resources and guidance we needed for the ongoing project. Our project mentor, Jiayu Li was also helpful when reviewing our work for better approaches to the problem.

REFERENCES

[1]Hochreiter, Sepp, and Jürgen Schmidhuber. "Long short-term memory." *Neural computation* 9.8 (1997): 1735-1780.

[2]https://storm.apache.org/

[3]https://leimao.github.io/blog/PyTorch-Distributed-Training/

[4]https://www.indycar.com/Fan-Info/INDYCAR-01/What-Is-INDYCAR

[5]Sherstinsky, Alex. "Fundamentals of recurrent neural network (rnn) and long short-term memory (lstm) network." *Physica D: Nonlinear Phenomena* 404 (2020): 132306.

[6]Li, Shen, et al. "Pytorch distributed: Experiences on accelerating data parallel training." *arXiv preprint arXiv:2006.15704* (2020).

[7]https://pytorch.org/tutorials/beginner/dist_overview.html

[8]Peng, Bo, et al. "Rank Position Forecasting in Car Racing." *arXiv preprint arXiv:2010.01707* (2020).

[9]Pang, Guansong, et al. "Deep learning for anomaly detection: A review." *arXiv preprint arXiv:2007.02500* (2020).

[10]Braei, Mohammad, and Sebastian Wagner. "Anomaly Detection in Univariate Time-series: A Survey on the State-of-the-Art." *arXiv preprint arXiv:2004.00433* (2020).