Project Final Report

<https://github.com/ouriquco/MotionCNN-Waymo-Open-Motion-Dataset>

<https://github.com/ouriquco/MTR>

<https://github.com/jshon88/MTR>

CMPE 249 - Intelligent Autonomous Systems

Howell DeGuzman - 013978029

Rahul Kandekar - 014315366

Cody Ourique 010892622

Jay Shon - 017553289

Team 5

# Table of Contents

[**Table of Contents 2**](#_9cu0awm5xbfx)

[**Application/Algorithm/Model + ROS (Rahul/Howell/Cody/Jay) 3**](#_ot1r4ccr2hj2)

[Application: Online Inference Engine for Self Driving 3](#_5tj8ag3fzh1b)

[Algorithm/Model: MTR (Motion Transformer) & MotionCNN 3](#_rf5ty396gn)

[ROS Implementation (Rahul/Howel) 4](#_fkhywpdj535m)

[**Overall Architecture 4**](#_qk4djgiqcmlo)

[Motion Transformer & Motion CNN (Cody/Jay) 4](#_oyvjkikja7h8)

[ROS (Rahul/Howell) 5](#_hb9lwkk1kejl)

[**Key Techniques 6**](#_qe728bkvi54p)

[Comparative Analysis (Cody/Jay) 6](#_x7vjf61qobhn)

[Positional Embedding Experiment 1 (Cody) 6](#_idiyj611d7pf)

[Positional Embedding Experiment 2 (Jay) 6](#_c9xh0ebk8ghn)

[Deployment to ROS (Rahul/Howell) 6](#_x7igcauy2pqs)

[Replication Steps 7](#_s96uul2m84ha)

[Inference Optimizations (Rahul/Howell) 8](#_5j8iudenkhcd)

[**Future Work (Rahul/Howell/Cody/Jay) 9**](#_u1ykdhhl8c61)

[**Task Distributions (Rahul/Howell/Cody/Jay) 9**](#_9peax6m38p0j)

[**Key References 11**](#_ikkdoonfotyh)

# Application/Algorithm/Model + ROS (Rahul/Howell/Cody/Jay)

### Application: Online Inference Engine for Self Driving

The overall goal of this project was to create an online inference engine that was able to process data for self driving applications in real time. This engine would be able to run inference on models for motion prediction by taking data from sensors. Then, it would be able to provide meaningful outputs that can be utilized by autonomous systems. The main purpose for this is so that self-driving vehicles have the capability to perform their own decision making and operate within certain constraints so that safe navigation is ensured. The main challenges for this application is being able to produce high accuracy inference while maintaining low latency, as well as integrating with different pipelines in relation to autonomous driving.

### Algorithm/Model: MTR (Motion Transformer) & MotionCNN

The models that we have decided to use for our project are MTR (Motion Transformer) and MotionCNN.

MTR is a model that leverages the transformer architecture by taking temporal data as input and predicting future trajectories of objects in relation to motion prediction tasks. It is able to model the problem by performing joint optimization on high level global intention and local movements in order to enhance the overall accuracy of predictions. Additionally, it introduces motion query pairs to stabilize training and eliminate dependencies on dense goal candidates. Within these queries there are the static intention query, which retrieves the agent’s motion static aspects, and the dynamic searching query, which retrieves the dynamic movements that an object makes based on interactions with other objects or factors. As far as MTR’s architectural components, it has the motion encoder which takes trajectories and map context data and encodes them to be utilized later by downstream modules. It also contains a dense future prediction module which predicts a single trajectory for each agent, and a dynamic map collection module which collects map elements along each predicted trajectory. Lastly, it also has a motion decoder, which takes what was encoded from the encoder step and converts it into trajectory points that are time-stamped. It then takes these points and integrates global and local motion refinements to create its final predictions.

MotionCNN is a convolutional neural network utilized for tasks in motion prediction. It performs well with prediction of future trajectories for objects that move in continuous space and was ranked 3rd for performance in motion prediction for the Waymo Open Dataset Challenge. As far as its overall architecture, it consists of a ResNet-34 backbone. This backbone is what processes input data using feature extraction so that features are represented well with efficiency. It is able to take map data and capture both high and low level spatial features. MotionCNN also has a fully connected layer that takes the features that have been extracted from the backbone and produces corresponding trajectory predictions with confidence scores. Additionally, it also utilizes a Negative Log Likelihood Loss function for 2D Gaussian Distribution so that predictions are further verified to be accurate with adjusted confidence estimate scores. These probabilistic outputs allow for predictions of future trajectories for each object in the continuous space and are paired with confidence scores that tell us how likely that trajectory will occur.

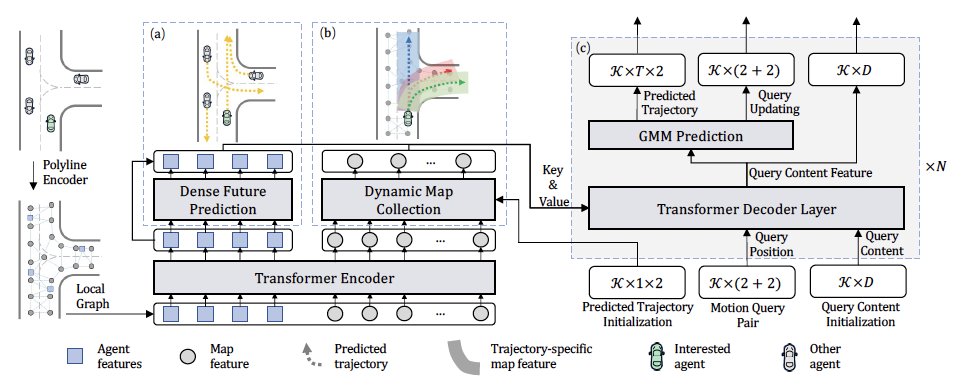
### ROS Implementation (Rahul/Howel)

To implement ROS, we used RoboStack ROS. RoboStack is a tool that allows integration of ROS2 within Python and Conda and provides pre-built ROS2 packages to simplify the setup process for the ROS2 environment. Within its implementation we can have different nodes that each serve as a component in its setup. We first have the publisher nodes which publish and stream the input data from sensors to the processing nodes downstream. Additionally, it has inference nodes with implements models that have been trained for inference and can be more optimized with additional tools like ONNX to improve latency. Lastly, it also has subscriber nodes that take the outputs that come from the motion prediction and transfer them over to other subsystems related to autonomous driving. Overall, ROS helps in deploying motion prediction models and integrating with other models to serve autonomous driving applications.

# Overall Architecture

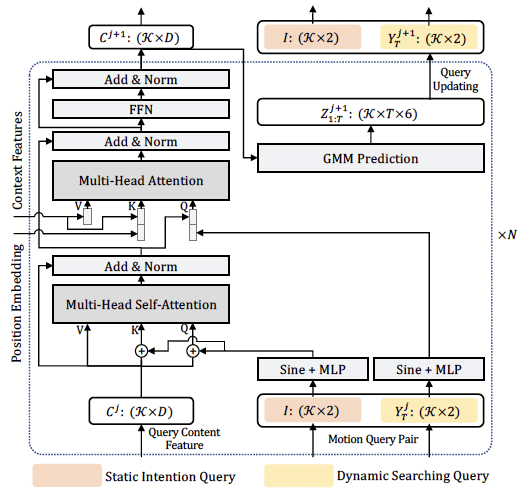
### Motion Transformer & Motion CNN (Cody/Jay)

#### Motion Transformer



*Figure 1: The architecture of MTR framework*

In this figure above, we can see the overall architecture and model work flow of MTR. Motion dataset including dynamic agents, map elements is preprocessed and represented as polyline through polyline encoder as MTR follows the vectorized representation to organize both input trajectories and road map as polylines. Transformer encoder, a.k.a motion encoder, looks at only local neighborhoods which reduces computational complexity while still preserving local relationships because locality structure is important for encoding scene context, especially for the road map. MTR adds spatial information to the features using sinusoidal encoding while the latest position and center of the polyline are used for agents and map elements respectively. Dense Future Prediction module predicts both future trajectories and velocities of all agents by adding a simple MLP that returns future position and velocity of each agent at time step i. This provides additional future context information to the decoder network, facilitating the model to predict more scene-compliant future trajectories for the ego vehicle. Output from the Dense Future Prediction module is encoded and combined with historical states embeddings. Dynamic Map Collection Module collects map elements along each predicted trajectory. Finally, Transformer decoder, a.k.a motion decoder, takes motion query pairs as a placeholder for motion prediction which include Static Intention Query and Dynamic Searching Query and provides final predictions.

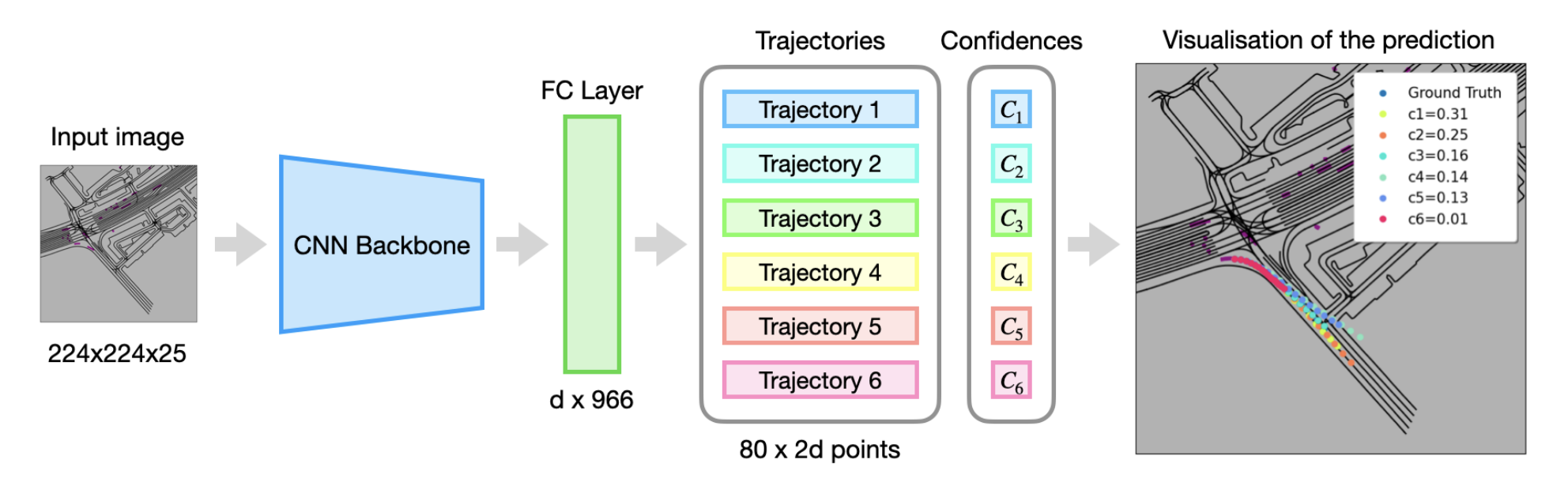
**

*Figure 2: The network structure of motion decoder network with motion query pair*

Motion decoder utilizes motion query pairs which consist of Static Intention Query and Dynamic Searching Query. Static Intention Query is a learnable positional embedding representing a specific motion mode such as left, right, straight, and it is used for global intention localization. Dynamic Searching Query is initialized similarly but dynamically updated to refine the trajectory by retrieving local features. K in the motion decoder network is the number of motion query pairs and also represents the number of clusters of different motion intentions. The predicted trajectories, motion query paris, and query content features are the outputs from the last decoder layer and will be taken as input to the next decoder layer.

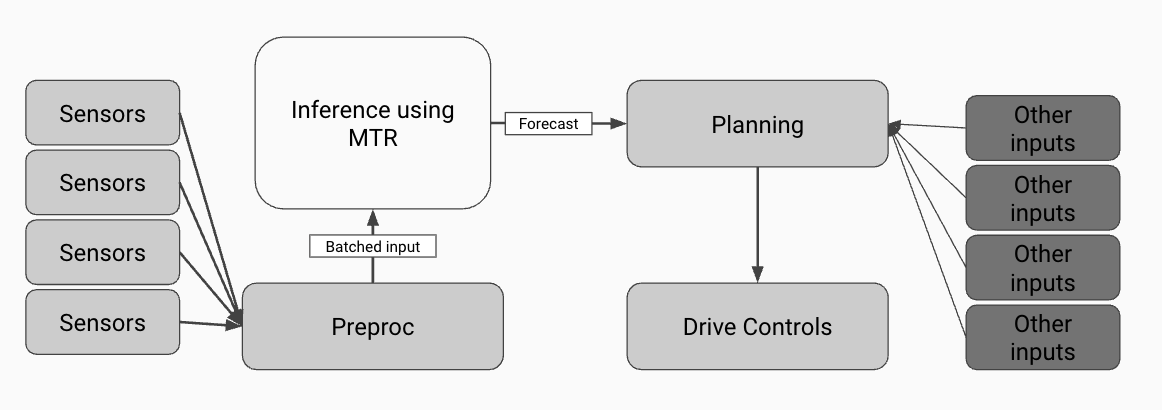
#### Motion CNN

Motion CNN uses a ResNet-18 backbone to extract features from images of size 224x224x25. The backbone is pretrained on the ImageNet dataset. The activation map, produced by the CNN, is then used as input to a fully connected layer which outputs a set of trajectory predictions and their corresponding confidence scores which are normalized using the softmax operator. A 2D Gaussian-based Negative Loss Likelihood is used to measure the overall performance of the model. We can observe the model’s architecture in figure 3.



*Figure 3: Overview of the architecture of MotionCNN*

### ROS (Rahul/Howell)



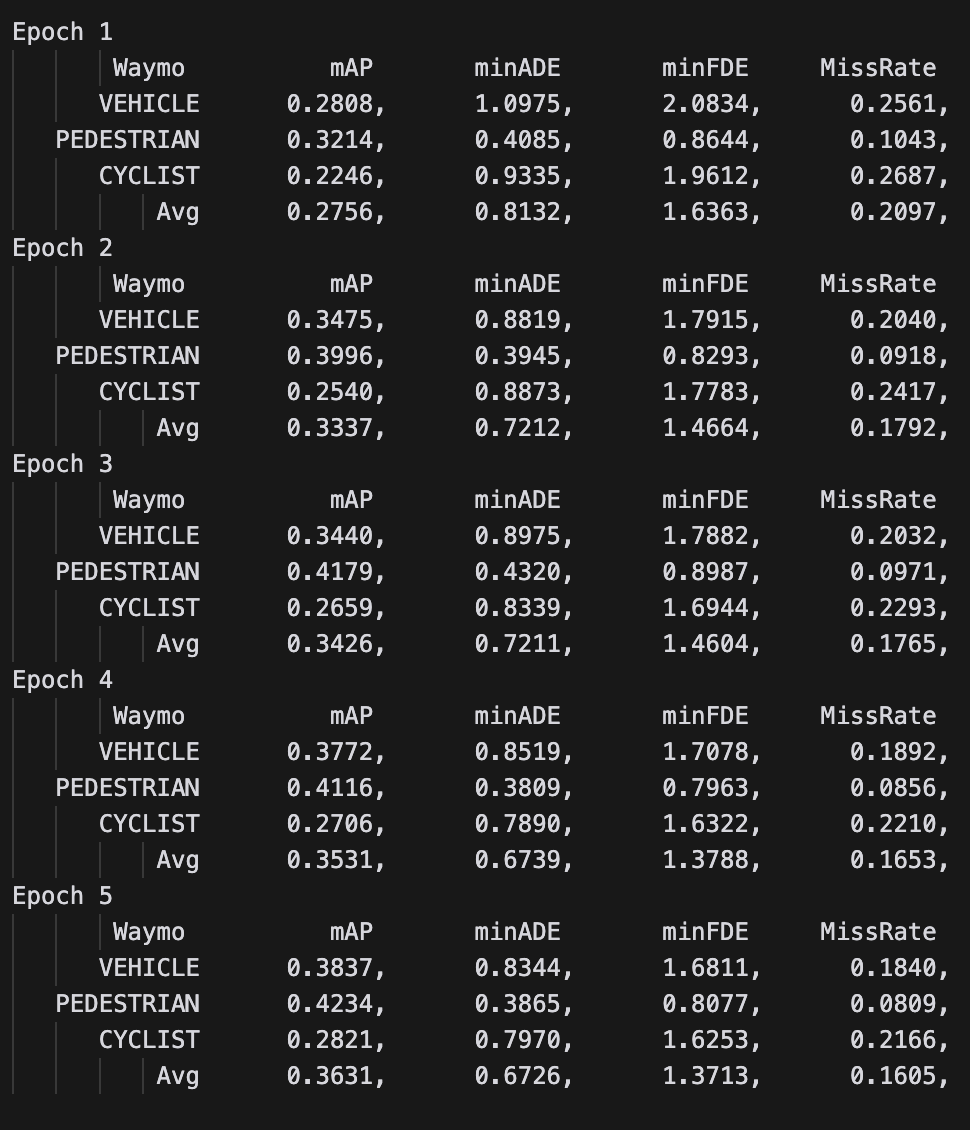
*Deployment with ROS Flow*

In the figure above, we can see the overall architecture and flow for ROS. Essentially, the architecture consists of numerous nodes that have specific tasks and interact with each other. These nodes are able to interact with each other through ROS’s publisher-subscriber capabilities. Through this publisher-subscriber mechanism, the publisher nodes are what produce data and publish it to a communication channel, and the subscriber nodes are what retrieve this data from the channel and subscribe to it. Within this channel, the data is transferred in the form of messages of different types.

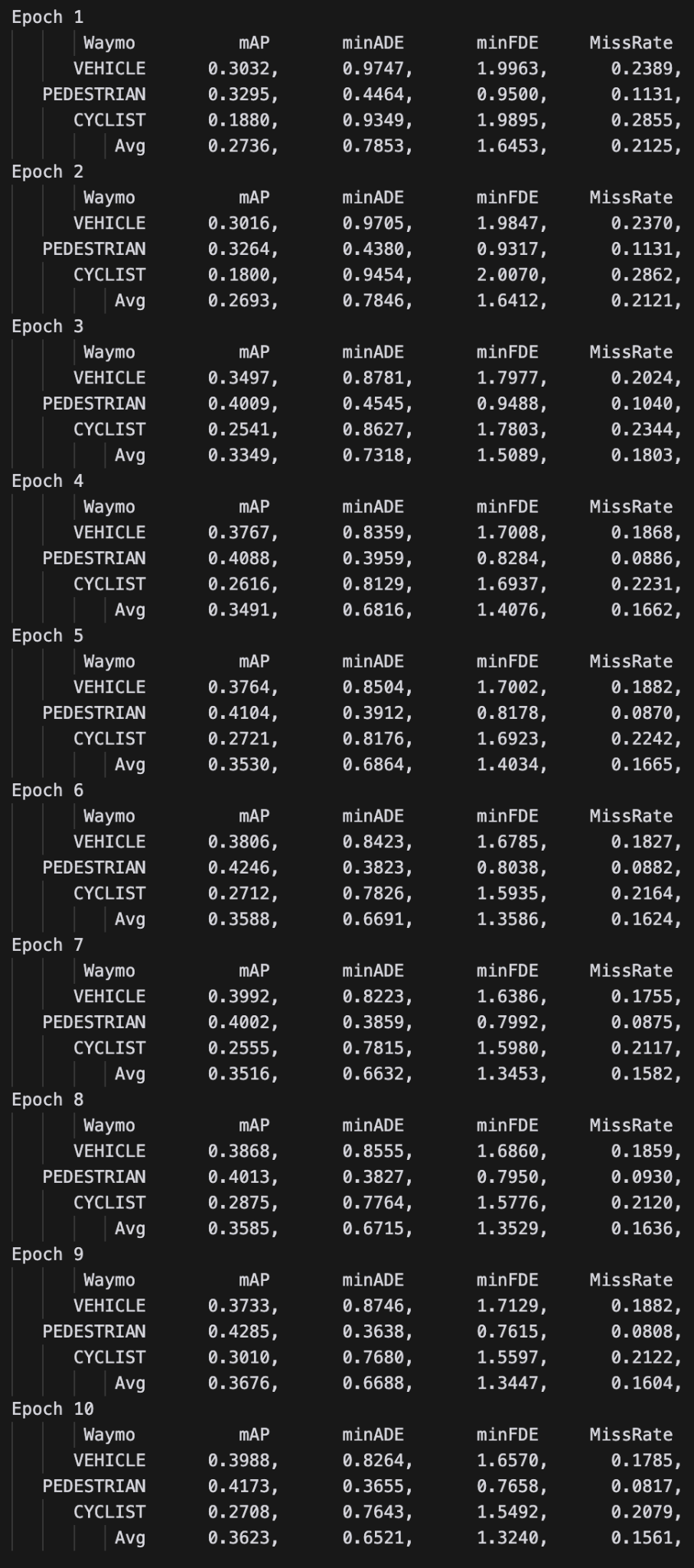
The system starts with multiple sensors, which can be cameras, LiDAR, and radar, that provide positions of objects, map features, and more. The preprocessing node starts by subscribing to this data where the data is taken as input and processed into a format that will be utilized by the model for inference. This processed data is packaged into a batched input by normalizing the sensor data and extracting key features from the map and trajectories. This node then takes samples from the dataset and publishes these samples in the correct format that is suitable for the inference node to utilize. We then have the inference node, where the idea here is that we have a single node that performs inference on the data and runs the model. It starts by subscribing to the communication channel with the preprocessed batched input and then performs motion prediction for trajectories utilizing the trained MTR model. The node then publishes these predictions to the ROS topic (channel) so it can be used by the display nodes. We then have a display node that subscribes to the inference node’s trajectory predictions and visualizes its results by printing and displaying its information. The information from this display node can then be used by a planning module that utilizes the predictions from the model to create a safe navigation route/plan. Then lastly, this route/plan will be passed to a drive controls module for autonomous systems that can display signals for a vehicle that suggest actions it needs to take to follow this path. The system can also take other inputs for its planning module to help improve its navigation route.

# Key Techniques

### Comparative Analysis (Cody/Jay)



*Figure: modified MTR model evaluation*

**

*Figure: MTR model evaluation*

The figures presented above illustrate the performance evaluations of our models based on the Waymo benchmarking metrics. Specifically, the evaluations were conducted using the validation dataset over differing numbers of training epochs: five epochs for the decoder-modified MTR model and ten epochs for the original MTR model. Although the performance disparity between the two models is minimal, the data indicates that the modified version consistently exhibits marginally superior performance compared to the original model within the initial five epochs.

Delving deeper into the results, the original MTR model achieved its peak performance at the ninth epoch out of ten, whereas the modified MTR model reached its optimal performance at the fifth epoch out of five. Notably, despite undergoing fewer training epochs, the modified MTR model demonstrated performance on par with the original MTR model at the ninth epoch. This suggests that the modifications introduced may enhance the model’s efficiency, enabling it to attain comparable performance levels with fewer training iterations.

However, it is important to acknowledge that both models have not yet been trained to convergence over a more extended number of epochs. Consequently, it remains premature to definitively assert the superiority of one model over the other based solely on the current results. To address this, our future work will involve training both the original and modified MTR models for additional epochs until they reach saturation points in terms of loss reduction. Achieving this will allow for a more comprehensive and conclusive comparison of their performance metrics.

### Positional Embedding Experiment 1 (Cody)

The purpose of this experiment was to apply a different technique to apply the sinusoidal positional embeddings. Typically, the embeddings are added to the input embeddings and the model will learn the meaning of these positional embeddings through training. However, this method does not guarantee that the positional embeddings will be distinct to the model. One method that ensures distinct, orthogonal, positional embedding representations is to concatenate the embeddings to the end of the positional embeddings. This method would increase the model parameter size and computational complexity but it would ensure that represented positions were distinct.

### Positional Embedding Experiment 2 (Jay)

This experiment intended to replicate the relative positional encodings that are discussed in MTR++ which is the newest iteration of the MTR model. The motivation behind the relative positional encodings is that for each individual token in the sequence, the token will have access to positional embedding for all other tokens with some values that represent the relative position. This type of positional encodings are extremely useful when dealing with graph or image structures. In the case of MTR++, there exist relative positional descriptors that are captured in the scene elements (e.g. relative distance and orientation) which are then transformed into Fourier features and passed through a Multi-Layer Perceptron to produce relative positional embeddings. These embeddings are then injected into the Keys and Values of the attention mechanism which allows the model to account for the spatial and temporal relationships.

### Deployment to ROS (Rahul/Howell)

The first step we did to deploy to ROS was to follow the existing RoboStack documentation for installation and ROS tutorials. We were able to develop simple publisher and subscriber nodes using Python in order to help us familiarize with creating nodes that pass string message information over to a ROS topic (communication channel). We also followed additional tutorials like creating custom msg and srv files with Python in order to learn how to define our own messages to be utilized by our publisher and subscriber nodes. Additionally, our research included looking into tools like roscore, rosnode, and rostopic to gain the basic skills that would help us implement a ROS system.

The next step we took was attempting to build ROS on the HPC. However, this attempt did not work because the code binary files for ROS are not available for the HPC computer. The ROS binary files are more platform-specific, and we found that the HPC environment lacked support for this. So we shifted to try and do a different approach.

Our next approach was to attempt to build ROS on HPC using Docker. Initially, we thought that Docker would help us get over the issue of binary incompatibility to try and run ROS on the HPC. However, this proved to be unsuccessful since we found that you aren’t able to access the GPU from Docker. Essentially, if the ROS process is running on Docker, then the ROS processes can’t access the GPU resources.

So our next step was to then transition to RoboStack and run ROS on the HPC which proved to be successful and allows access to the GPU. This proved to be successful since RoboStack is compatible with HPC environments and allows for efficient computations for models and inference. This then allowed us to integrate MTR with ROS, where on a high level, we first cloned the MTR repository and built it using its instructions for model setup. This build step included installing necessary dependencies for MTR like PyTorch. Then, in the ROS Python codebase, we imported and utilized the MTR model by modifying the ROS inference node for prediction of trajectories. We then also used and loaded our pretrained weights for the MTR model to carry out our tasks for inference.

#### Replication Steps

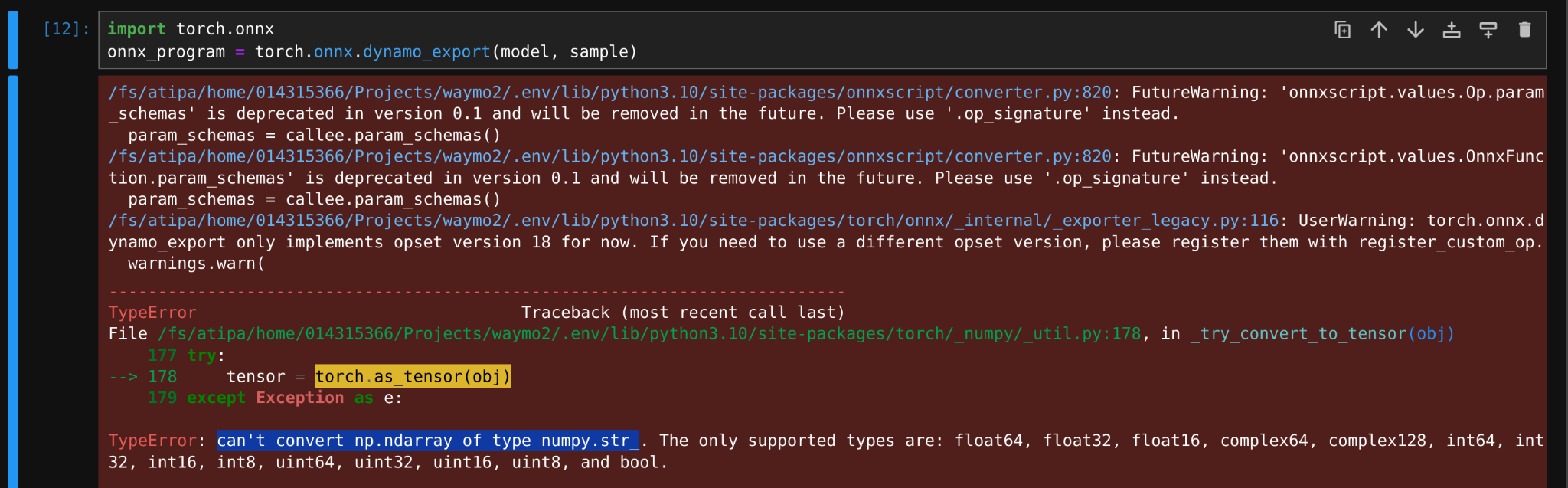
Asciinema video demo of the ROS implementations <https://asciinema.org/a/VjJwLBx43PAmfVuggtRF4Yh4Z>

The following are the steps to replicate the ros set up we implemented. These steps include, creating a virtual environment with micromamba (conda/mamba are also expected to work), installing robostack-ros, setting up the MTR project, setting up our live inference demonstration project, running our project as a ros node which uses the MTR project with GPU resource. For convenience this demonstration will use the waymo dataset that we have already downloaded and prepossessed. We will also extract our ros project from an archive we have saved in the /data directory.

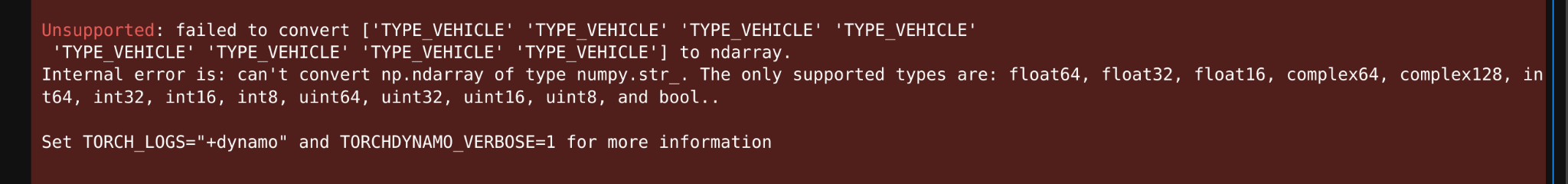
1. ssh into HPC
2. Make and cd into project directory
3. Make the virtual environment with RoboStack
   1. micromamba create -p ./env1 python=3.10 -c robostack-staging ros-humble-desktop compilers cmake pkg-config make ninja colcon-common-extensions catkin\_tools rosdep
   2. Give it a while to install all dependencies
4. Activate the environment then install cuda-toolkit
   1. micromamba install cuda-toolkit
   2. Do not try to merge this step with the previous step. cuda-toolkit needs to be installed separately.
5. Clone the MTR project in the project directory
   1. git clone git@github.com:sshaoshuai/MTR.git
   2. git clone https://github.com/sshaoshuai/MTR.git
6. Change directory into the MTR project
7. Install the MTR dependencies
   1. pip install -r requirements.txt
   2. Give it a while to install all the dependencies
8. Request a gpu node
   1. srun -p gpu --pty /bin/bash
9. Still in the MTR project directory, build the cuda modules defined in the MTR project
   1. python setup.py develop
10. Back in the project directory copy /data/cmpe258-sp24/014315366/print\_sample.py into the project directory and use it to verify MTR is properly set up
    1. python print\_sample.py
    2. Expected behavior is that it should print a dictionary of tensors
11. Make a src/ directory and extract the ros\_mtr project into it
    1. tar xvf /data/cmpe258-sp24/014315366/ros\_mtr.tar.gz --directory=src/
12. Build the ros\_mtr package
    1. colcon build --packages-select ros\_mtr
    2. It may look like there are a lot of errors, but let them pass. In the end you’ll see a display that shows ros\_mtr is built
13. Install the built project
    1. source install/local\_setup.bash
14. Print a single sample using ros console script
    1. ros2 run ros\_mtr print\_sample
15. Run inference node for continuous inference
    1. ros2 run ros\_mtr run\_inference\_node
    2. You should see a continuous live inference process

### Inference Optimizations (Rahul/Howell)

For this technique, one of our goals was to perform inference optimization by accelerating the inference for MTR. We wanted to do this by exporting the MTR model to the ONNX format. ONNX is an open source format framework that allows different machine learning frameworks to exchange information and make use of it. In our case, it allows models to be shared and deployed to be utilized more efficiently (i.e. MTR). In order to export MTR to ONNX, we explored 2 main approaches: Dynax and TorchScript. Dynax is a tool that is used for dynamic model analysis and export. However, with this approach, we ran into some compatibility issues since there were some operations in MTR that were not supported by ONNX. This then prompted us to try the TorchScript approach, however we also ran into the same limitations in the ONNX format with MTR.

**

The main challenge that we faced here was that the MTR consists of certain operations on numpy string arrays. However Pytorch does not support string tensors. This prevents us from exporting the model to the ONNX format. On further investigation we found that the numpy strings are used to represent scenario ids. Based on this we tried replacing the original alphanumeric scenario\_ids with numeric scenario ids. However we discovered even more operations internal to MTR that are defined in terms of strings. For example, the waymo dataset categorizes entities like vehicles, pedestrians, cycles as strings. So the MTR project also continues to represent them with strings internally.



Our exploration revealed that although the fix might be forward, finding and replacing every string operation might take too time consuming so we leave it as future work.

# Future Work (Rahul/Howell/Cody/Jay)

In the ongoing development and refinement of our motion prediction models, specifically MTR and Motion CNN, there remains substantial potential for enhancements and the exploration of alternative methodologies. Our current training regimen utilizes a single GPU node on a High-Performance Computing (HPC) system, which has resulted in prolonged training durations. For instance, each epoch for the MTR model requires approximately 11 hours on an NVIDIA A40 GPU. This extended training time poses a significant bottleneck, limiting our ability to iterate rapidly and explore a broader range of hyperparameter configurations.

To address this limitation, we intend to optimize our training framework by configuring Slurm scripts to facilitate distributed training across multiple GPU nodes. By leveraging a distributed computing environment, we can significantly reduce the time required per epoch, thereby enabling the training of a greater number of epochs within a feasible time frame. This is particularly important for the MTR model, which, as per the original paper, was trained for 30 epochs. Extending the number of epochs through distributed training will likely enhance the model’s performance and generalization capabilities.

In addition to scaling our computational resources, we plan to integrate advanced techniques from recent advancements in the field. Specifically, we aim to implement query-centric relative positional encoding as described in the MTR++ paper. Currently, the MTR++ methodology lacks publicly available source code, presenting an opportunity for us to contribute by developing and incorporating this feature into our existing MTR framework. Given that MTR++ represents an evolved version of the original MTR model, incorporating its core modifications is anticipated to yield significant performance improvements, enhancing both accuracy and robustness in motion prediction tasks.

Furthermore, to ensure a comprehensive and unbiased evaluation of our models, we will implement the Waymo evaluation metrics for the Motion CNN model. Although we have successfully trained Motion CNN on the Waymo motion dataset, our evaluation to date has been limited by the absence of standardized performance metrics. By adopting the Waymo evaluation framework, we can facilitate a fair and direct comparison between Motion CNN and the MTR model. This will enable us to quantitatively assess the strengths and weaknesses of each model, providing clear insights into their respective performance profiles.

For ONNX optimization, one area of future work would be to make changes and new configurations for the MTR model so that it does not rely on string-based operations. This is primarily because these string-based operations are not supported by ONNX, so we would have to replace them with alternatives that would be compatible. By having this replacement within the model, we would be able to utilize ONNX capabilities and perform faster inference with this export of MTR.

Our last area of focus for future work would be with the ROS system improvements. With the current state of our project, the data loading process is tightly coupled with the inference node which limits its overall flexibility. For the future, we could decouple these 2 components by separating the data publisher from the inference node. By separating these components, inference would then only be performed on data that is published and subscribed to improve modularity. Another area of future work for ROS would be to integrate simulated data into the prediction node. We could feed in data for simulation so that we can see how the system performs in realistic scenarios and evaluate them for autonomous driving applications.

# Task Distributions (Rahul/Howell/Cody/Jay)

Cody:

* Downloaded Waymo motion dataset
* Preprocessed Waymo motion dataset v\_1\_3\_0
  + Preprocessed in Google Colab and moved the data into a Google Cloud bucket where it could be accessed by HPC
* Preprocessed Waymo motion dataset v\_1\_0\_0
  + Preprocessed in HPC
* Completed setting up environment for MTR
  + Created virtual environment with python 3.10
  + Installed all necessary dependencies
  + Resolved all the errors and issues related to MTR
* Completed setting up environment for MotionCNN
  + Created virtual environment with python 3.8
  + Installed all necessary dependencies
  + Resolved all the errors and issues related to MotionCNN
* Trained MTR with 20% data for 10 epochs
  + Each epoch took around 12-20 hours to complete
* Trained MotionCNN with 100% data for 60 epochs (still processing, currently at epoch 32)
  + Each epoch takes between 15-24 hours to complete
* Attempted Architecture changes
  + Concatenated positional sinusoidal encodings
  + Adding relative positional encodings

Jay:

* Downloaded Waymo motion dataset
* Preprocessed Waymo motion dataset
  + Preprocessed in AWS EC2 as installing waymo API at the time did not work well on HPC
  + Transferred preprocessed waymo dataset from EC2 to HPC
* Completed setting up environment for MTR
  + Created virtual environment with Python 3.10
  + Installed all necessary dependencies
  + Resolved all the errors and issues related to MTR
* Implemented MTR
* Trained MTR for 10 epochs
  + Each epoch takes 20 hours on P100 and 11 hours on A40
* Created animation for one sample motion scene
* Changed Motion decoder and trained
  + Trained for 5 epochs
* Performed model evaluation on Waymo metrics through Waymo API on Google Colab
  + Had an issue with Waymo open dataset installation on HPC
* Resolved Waymo open dataset installation issue on HPC
* Performed model evaluation again on HPC
* Compared the performance between original MTR with modified version MTR
  + Up until 5 epochs, modified version has slightly better evaluation results
* Attempted to change the size of motion query pairs
* Programming and Documentation

Rahul:

* Attempt to build ROS on HPC
* Attempt to build ROS on HPC using Docker
* Implemented ROS tutorial demos
  + Examined and learned publisher-subscriber pipeline for ROS project
  + Created publisher node
  + Created subscriber node
  + Performed testing
* Implemented ROS project
  + Used RoboStack to run ROS on HPC
  + Import and use MTR in ROS project
    - Developed full ROS pipeline using MTR
    - Cloned and built MTR repository
    - Modified ROS inference node to utilize MTR model
    - Used pretrained weights in inference process in ROS node
    - Created demo to visualize publisher and subscriber node intractability
* Attempt to implement ONNX optimization
  + Dynax attempt for export
  + PyTorch attempt for export
  + Investigate requirements for ONNX optimization
* Pair programming with Howell
  + Documenting when Howell is programming
  + Programming when Howell is Documenting

Howell:

* Attempt to build ROS on HPC
* Attempt to build ROS on HPC using Docker
* Implemented ROS tutorial demos
  + Examined and learned publisher-subscriber pipeline for ROS project
  + Created publisher node
  + Created subscriber node
  + Performed testing
* Implemented ROS project
  + Used RoboStack to run ROS on HPC
  + Import and use MTR in ROS project
    - Developed full ROS pipeline using MTR
    - Cloned and built MTR repository
    - Modified ROS inference node to utilize MTR model
    - Used pretrained weights in inference process in ROS node
    - Created demo to visualize publisher and subscriber node intractability
* Attempt to implement ONNX optimization
  + Dynax attempt for export
* PyTorch attempt for export
* Pair programming with Rahul
  + Documenting when Rahul is programming
  + Programming when Rahul is Documenting

# Key References

<https://github.com/sshaoshuai/MTR?tab=readme-ov-file>

<https://arxiv.org/abs/2209.13508>

<https://arxiv.org/abs/2306.17770>

<https://github.com/sshaoshuai/MTR/tree/master>

<https://robostack.github.io/GettingStarted.html#__tabbed_1_2>

<https://docs.ros.org/en/humble/Tutorials/Beginner-CLI-Tools/Introducing-Turtlesim/Introducing-Turtlesim.html>

<https://docs.ros.org/en/humble/Tutorials/Beginner-Client-Libraries/Writing-A-Simple-Py-Publisher-And-Subscriber.html>

<https://docs.ros.org/en/humble/Tutorials/Beginner-Client-Libraries/Custom-ROS2-Interfaces.html>

<https://deepdatamininglearning.readthedocs.io/en/latest/source/HPC2.html#introduction-of-hpc>

<https://slurm.schedmd.com/gres.html#GPU_Management>