**Project link**

[**waymo-open-dataset/tutorial\_motion.ipynb at master · waymo-research/waymo-open-dataset · GitHub**](https://github.com/waymo-research/waymo-open-dataset/blob/master/tutorial/tutorial_motion.ipynb)

[**tfrecord data format**](https://waymo.com/intl/en_us/open/data/motion/)

[**Problem Statement & Results**](https://waymo.com/intl/en_us/open/challenges/2022/motion-prediction/)

[**tf record variable declaration**](https://waymo.com/intl/en_us/open/data/motion/tfexample)

**Multi-layer Perceptron (MLP)**

**Long Short-Term Memory(LSTM)**

**roadgraph\_features = {**

**"roadgraph\_samples/dir": tf.io.FixedLenFeature(**

**[20000, 3], tf.float32, default\_value=None**

**),**

**"roadgraph\_samples/id": tf.io.FixedLenFeature(**

**[20000, 1], tf.int64, default\_value=None**

**),**

**"roadgraph\_samples/type": tf.io.FixedLenFeature(**

**[20000, 1], tf.int64, default\_value=None**

**),**

**"roadgraph\_samples/valid": tf.io.FixedLenFeature(**

**[20000, 1], tf.int64, default\_value=None**

**),**

**"roadgraph\_samples/xyz": tf.io.FixedLenFeature(**

**[20000, 3], tf.float32, default\_value=None**

**),**

**}**

This is a dictionary called roadgraph\_features with keys representing features of a road network. Each feature is defined using tf.io.FixedLenFeature(), which specifies the **data type, shape, and default value of the feature.**

The features include:

* "roadgraph\_samples/dir": A **3-dimensional float32** tensor with shape [20000, 3] representing the direction of each road segment in the road network. The default value is None.
* "roadgraph\_samples/id": A **1-dimensional int64** tensor with shape [20000, 1] representing the unique identifier of each road segment in the road network. The default value is None.
* "roadgraph\_samples/type": A **1-dimensional int64** tensor with shape [20000, 1] representing the type of each road segment in the road network. The default value is None.
* "roadgraph\_samples/valid": A **1-dimensional int64** tensor with shape [20000, 1] representing whether each road segment in the road network is valid or not. The default value is None.
* "roadgraph\_samples/xyz": A **3-dimensional float32** tensor with shape [20000, 3] representing the xyz coordinates of each road segment in the road network. The default value is None.

**# Features of other agents.**

**state\_features = {**

**"state/id": tf.io.FixedLenFeature([128], tf.float32, default\_value=None),**

**"state/type": tf.io.FixedLenFeature([128], tf.float32, default\_value=None),**

**"state/is\_sdc": tf.io.FixedLenFeature([128], tf.int64, default\_value=None),**

**"state/tracks\_to\_predict": tf.io.FixedLenFeature(**

**[128], tf.int64, default\_value=None**

**),**

**"state/current/bbox\_yaw": tf.io.FixedLenFeature(**

**[128, 1], tf.float32, default\_value=None**

**),**

**"state/current/height": tf.io.FixedLenFeature(**

**[128, 1], tf.float32, default\_value=None**

**),**

**"state/current/length": tf.io.FixedLenFeature(**

**[128, 1], tf.float32, default\_value=None**

**),**

**"state/current/timestamp\_micros": tf.io.FixedLenFeature(**

**[128, 1], tf.int64, default\_value=None**

**),**

**"state/current/valid": tf.io.FixedLenFeature(**

**[128, 1], tf.int64, default\_value=None**

**),**

**"state/current/vel\_yaw": tf.io.FixedLenFeature(**

**[128, 1], tf.float32, default\_value=None**

**),**

**"state/current/velocity\_x": tf.io.FixedLenFeature(**

**[128, 1], tf.float32, default\_value=None**

**),**

**"state/current/velocity\_y": tf.io.FixedLenFeature(**

**[128, 1], tf.float32, default\_value=None**

**),**

**"state/current/speed": tf.io.FixedLenFeature(**

**[128, 1], tf.float32, default\_value=None**

**),**

**"state/current/width": tf.io.FixedLenFeature(**

**[128, 1], tf.float32, default\_value=None**

**),**

**"state/current/x": tf.io.FixedLenFeature([128, 1], tf.float32, default\_value=None),**

**"state/current/y": tf.io.FixedLenFeature([128, 1], tf.float32, default\_value=None),**

**"state/current/z": tf.io.FixedLenFeature([128, 1], tf.float32, default\_value=None),**

**"state/future/bbox\_yaw": tf.io.FixedLenFeature(**

**[128, 80], tf.float32, default\_value=None**

**),**

**"state/future/height": tf.io.FixedLenFeature(**

**[128, 80], tf.float32, default\_value=None**

**),**

**"state/future/length": tf.io.FixedLenFeature(**

**[128, 80], tf.float32, default\_value=None**

**),**

**"state/future/timestamp\_micros": tf.io.FixedLenFeature(**

**[128, 80], tf.int64, default\_value=None**

**),**

**"state/future/valid": tf.io.FixedLenFeature(**

**[128, 80], tf.int64, default\_value=None**

**),**

**"state/future/vel\_yaw": tf.io.FixedLenFeature(**

**[128, 80], tf.float32, default\_value=None**

**),**

**"state/future/velocity\_x": tf.io.FixedLenFeature(**

**[128, 80], tf.float32, default\_value=None**

**),**

**"state/future/velocity\_y": tf.io.FixedLenFeature(**

**[128, 80], tf.float32, default\_value=None**

**),**

**"state/future/width": tf.io.FixedLenFeature(**

**[128, 80], tf.float32, default\_value=None**

**),**

**"state/future/x": tf.io.FixedLenFeature([128, 80], tf.float32, default\_value=None),**

**"state/future/y": tf.io.FixedLenFeature([128, 80], tf.float32, default\_value=None),**

**"state/future/z": tf.io.FixedLenFeature([128, 80], tf.float32, default\_value=None),**

**"state/past/bbox\_yaw": tf.io.FixedLenFeature(**

**[128, 10], tf.float32, default\_value=None**

**),**

**"state/past/height": tf.io.FixedLenFeature(**

**[128, 10], tf.float32, default\_value=None**

**),**

**"state/past/length": tf.io.FixedLenFeature(**

**[128, 10], tf.float32, default\_value=None**

**),**

**"state/past/timestamp\_micros": tf.io.FixedLenFeature(**

**[128, 10], tf.int64, default\_value=None**

**),**

**"state/past/valid": tf.io.FixedLenFeature([128, 10], tf.int64, default\_value=None),**

**"state/past/vel\_yaw": tf.io.FixedLenFeature(**

**[128, 10], tf.float32, default\_value=None**

**),**

**"state/past/velocity\_x": tf.io.FixedLenFeature(**

**[128, 10], tf.float32, default\_value=None**

**),**

**"state/past/velocity\_y": tf.io.FixedLenFeature(**

**[128, 10], tf.float32, default\_value=None**

**),**

**"state/past/speed": tf.io.FixedLenFeature(**

**[128, 10], tf.float32, default\_value=None**

**),**

**"state/past/width": tf.io.FixedLenFeature(**

**[128, 10], tf.float32, default\_value=None**

**),**

**"state/past/x": tf.io.FixedLenFeature([128, 10], tf.float32, default\_value=None),**

**"state/past/y": tf.io.FixedLenFeature([128, 10], tf.float32, default\_value=None),**

**"state/past/z": tf.io.FixedLenFeature([128, 10], tf.float32, default\_value=None),**

**"scenario/id": tf.io.FixedLenFeature([1], tf.string, default\_value=None),**

**}**

<https://waymo.com/intl/en_us/open/data/motion/> →Samples

Current →1 Samples

Future →80 future Samples

Past →10 history Samples

**All coordinates in the dataset are in a global frame with X as East, Y as North and Z as up**

These are the features of an agent in a self-driving car system. The **features** can be grouped into three main categories: **current state, past state, and future state**.

The current state features include the ID and type of the agent, whether the agent is a self-driving car, the tracks the agent is predicted to travel on, the agent's current bounding box yaw, height, length, timestamp, validity, velocity yaw, velocity x, velocity y, speed, width, and position in the x, y, and z axes.

The past state features include the agent's bounding box yaw, height, length, timestamp, validity, velocity yaw, velocity x, velocity y, speed, width, and position in the x, y, and z axes for the past 10 time steps.

The future state features include the agent's bounding box yaw, height, length, timestamp, validity, velocity yaw, velocity x, velocity y, width, and position in the x, y, and z axes for the next 80 time steps.

**traffic\_light\_features = {**

**"traffic\_light\_state/current/state": tf.io.FixedLenFeature(**

**[1, 16], tf.int64, default\_value=None**

**),**

**"traffic\_light\_state/current/valid": tf.io.FixedLenFeature(**

**[1, 16], tf.int64, default\_value=None**

**),**

**"traffic\_light\_state/current/id": tf.io.FixedLenFeature(**

**[1, 16], tf.int64, default\_value=None**

**),**

**"traffic\_light\_state/current/x": tf.io.FixedLenFeature(**

**[1, 16], tf.float32, default\_value=None**

**),**

**"traffic\_light\_state/current/y": tf.io.FixedLenFeature(**

**[1, 16], tf.float32, default\_value=None**

**),**

**"traffic\_light\_state/current/z": tf.io.FixedLenFeature(**

**[1, 16], tf.float32, default\_value=None**

**),**

**"traffic\_light\_state/past/state": tf.io.FixedLenFeature(**

**[10, 16], tf.int64, default\_value=None**

**),**

**"traffic\_light\_state/past/valid": tf.io.FixedLenFeature(**

**[10, 16], tf.int64, default\_value=None**

**),**

**# "traffic\_light\_state/past/id":**

**# tf.io.FixedLenFeature([1, 16], tf.int64, default\_value=None),**

**"traffic\_light\_state/past/x": tf.io.FixedLenFeature(**

**[10, 16], tf.float32, default\_value=None**

**),**

**"traffic\_light\_state/past/y": tf.io.FixedLenFeature(**

**[10, 16], tf.float32, default\_value=None**

**),**

**"traffic\_light\_state/past/z": tf.io.FixedLenFeature(**

**[10, 16], tf.float32, default\_value=None**

**),**

**}**

**features\_description = {}**

**features\_description.update(roadgraph\_features)**

**features\_description.update(state\_features)**

**features\_description.update(traffic\_light\_features)**

This code appears to define a dictionary called traffic\_light\_features that contains tf.io.FixedLenFeature objects with various shapes and data types. These features likely describe properties of traffic lights, such as their state (e.g., red, green, yellow), their location (x, y, z coordinates), and their past state history.

The features\_description dictionary appears to be a combination of roadgraph\_features, state\_features, and traffic\_light\_features. It's possible that these dictionaries are used as inputs to a TensorFlow model.

**def \_parse\_model\_ready(value):**

**"""**

**This function parses 1 scenario (where we max can have 128 different objects) out of many in 1 .tf file**

**into its basic elements. It returns all the necessary information about the scenario such as velocity**

**and position of each object for all the time points, which of the objects do currently or have/will exist**

**decoded\_example = tf.io.parse\_single\_example(value, features\_description)**

**Inputs:**

**value: 1 tf file that is going to be parsed**

**Returns:**

**inputs: a dictionary that contains all the necessary information to do prediction about the future.**

**"""**

**decoded\_example = tf.io.parse\_single\_example(value, features\_description)**

**#past\_states includes all relevant information about pasts states that does not need to be normalized**

**past\_states = tf.stack([**

**#Only includes valid values for each of the features by masking with valid vector. Currently sets all invalid enties to zero, but can easily be changed**

**tf.where(decoded\_example['state/past/valid'] > 0, decoded\_example['state/past/speed'], tf.zeros\_like(decoded\_example['state/past/speed'])),**

**tf.where(decoded\_example['state/past/valid'] > 0, decoded\_example['state/past/bbox\_yaw'], tf.zeros\_like(decoded\_example['state/past/bbox\_yaw'])),**

**tf.where(decoded\_example['state/past/valid'] > 0, decoded\_example['state/past/velocity\_x'], tf.zeros\_like(decoded\_example['state/past/velocity\_x'])),**

**tf.where(decoded\_example['state/past/valid'] > 0, decoded\_example['state/past/velocity\_y'], tf.zeros\_like(decoded\_example['state/past/velocity\_y'])),**

**tf.where(decoded\_example['state/past/valid'] > 0, decoded\_example['state/past/vel\_yaw'], tf.zeros\_like(decoded\_example['state/past/vel\_yaw']))**

**], 1) # shape (128,5,10) --> 128 = #of objects in the scenario, 5 properties (speed,bbox\_yaw, ... vel\_yaw), 10 steps in the past**

**#past\_xyz includes all relevant information about past states that needs to be normalized**

**past\_xyz = tf.stack([**

**decoded\_example['state/past/x'],**

**decoded\_example['state/past/y'],**

**decoded\_example['state/past/z'],**

**], 1) # shape (128,3,10)**

**#Same as for pasts\_states but for current time point, see comment above**

**cur\_states = tf.stack([**

**#Only includes valid values for each of the features by masking with valid vector. Currently sets all invalid entries to zero, but can easily be changed**

**tf.where(decoded\_example['state/current/valid'] > 0, decoded\_example['state/current/speed'], tf.zeros\_like(decoded\_example['state/current/speed'])),**

**tf.where(decoded\_example['state/current/valid'] > 0, decoded\_example['state/current/bbox\_yaw'], tf.zeros\_like(decoded\_example['state/current/bbox\_yaw'])),**

**tf.where(decoded\_example['state/current/valid'] > 0, decoded\_example['state/current/velocity\_x'], tf.zeros\_like(decoded\_example['state/current/velocity\_x'])),**

**tf.where(decoded\_example['state/current/valid'] > 0, decoded\_example['state/current/velocity\_y'], tf.zeros\_like(decoded\_example['state/current/velocity\_y'])),**

**tf.where(decoded\_example['state/current/valid'] > 0, decoded\_example['state/current/vel\_yaw'], tf.zeros\_like(decoded\_example['state/current/vel\_yaw']))**

**], 1) # shape (128,5,1)**

**cur\_xyz = tf.stack([**

**decoded\_example['state/current/x'],**

**decoded\_example['state/current/y'],**

**decoded\_example['state/current/z'],**

**], 1) # shape (128,3,1)**

**#Same as for pasts\_states Note, future states are currently not being used and might be able to be removed.**

**future\_states = tf.stack([**

**# If we use future states speed should probably be included but currently throws an error.**

**#decoded\_example['state/future/speed'],**

**decoded\_example['state/future/bbox\_yaw'],**

**decoded\_example['state/future/velocity\_x'],**

**decoded\_example['state/future/velocity\_y'],**

**decoded\_example['state/future/vel\_yaw']**

**], 1) # Will have shape (128,5,80)**

**#Same as for past\_xyz but for future time points**

**future\_xyz = tf.stack([**

**decoded\_example['state/future/x'],**

**decoded\_example['state/future/y'],**

**decoded\_example['state/future/z'],**

**], 1) # Will have shape (128,3,80)**

**past\_is\_valid = decoded\_example['state/past/valid'] > 0**

**current\_is\_valid = decoded\_example['state/current/valid'] > 0**

**future\_is\_valid = decoded\_example['state/future/valid'] > 0**

**is\_valid = tf.concat([past\_is\_valid, current\_is\_valid], axis = 1)**

**gt\_future\_is\_valid = tf.concat([past\_is\_valid, current\_is\_valid, future\_is\_valid], 1)**

**# If a sample was not seen at all in the past, we declare the sample as invalid.**

**sample\_is\_valid = tf.reduce\_any(tf.concat([past\_is\_valid, current\_is\_valid], 1), 1)**

**# Gets the valid object types and set all invalid object types to 0.**

**object\_type = tf.where(sample\_is\_valid,decoded\_example['state/type'], tf.zeros\_like(decoded\_example['state/type']))**

**# Get all valid roadgraph types and roadgraph directions and setting all others to 0.**

**roadgraph\_type = tf.where(decoded\_example['roadgraph\_samples/valid'] > 0,decoded\_example['roadgraph\_samples/type'], tf.zeros\_like(decoded\_example['roadgraph\_samples/type']))**

**roadgraph\_mask = tf.concat([decoded\_example['roadgraph\_samples/valid'] > 0, decoded\_example['roadgraph\_samples/valid'] > 0, decoded\_example['roadgraph\_samples/valid'] > 0],1)**

**roadgraph\_dir = tf.where(roadgraph\_mask, decoded\_example['roadgraph\_samples/dir'], tf.zeros\_like(decoded\_example['roadgraph\_samples/dir']))**

**inputs = {**

**'past\_states': past\_states,**

**'past\_xyz': past\_xyz,**

**'past\_valid': past\_is\_valid,**

**'current\_states': cur\_states,**

**'current\_xyz': cur\_xyz,**

**'current\_valid': decoded\_example['state/current/valid'] > 0,**

**'future\_states': future\_states,**

**'future\_xyz': future\_xyz,**

**'future\_valid': decoded\_example['state/future/valid'] > 0,**

**'object\_type': decoded\_example['state/type'],**

**'is\_valid': is\_valid,**

**'tracks\_to\_predict': decoded\_example['state/tracks\_to\_predict'] > 0,**

**'sample\_is\_valid': sample\_is\_valid,**

**'roadgraph\_samples/dir' : roadgraph\_dir,**

**'roadgraph\_samples/id' : decoded\_example['roadgraph\_samples/id'],**

**'roadgraph\_samples/type' : roadgraph\_type,**

**'roadgraph\_samples/valid' : decoded\_example['roadgraph\_samples/valid'],**

**'roadgraph\_samples/xyz' : decoded\_example['roadgraph\_samples/xyz'],**

**# For finding the gt in training and validation set**

**'final\_x': tf.reshape(decoded\_example['state/future/x'][:,-1], (-1,1)),**

**'final\_y': tf.reshape(decoded\_example['state/future/y'][:,-1], (-1,1))**

**}**

**return inputs**

→<https://waymo.com/intl/en_us/open/data/motion/tfexample> →To declare all variables of .tfrecord dataset

This function takes in a TensorFlow file and returns a dictionary containing all the necessary information to do prediction about the future. The **TensorFlow fil**e contains information about the **past, current, and future states of multiple objects in a scenario**.

The function first parses the TensorFlow file into its basic elements using tf.io.parse\_single\_example. It then extracts information about the past states, past x-y-z positions, current states, current x-y-z positions, future states, and future x-y-z positions of each object in the scenario.

The **past and current states** each consist of **five properties** for each of the 128 objects: **speed, bounding box yaw, velocity in x and y directions, and velocity yaw.**

The **future states** only include **four** properties: **bounding box yaw, velocity in x and y directions, and velocity yaw.**

The x-y-z positions for all three time points are each represented by a tensor of shape (128, 3, T) where T is the number of time points for that particular set of positions.

The function also creates a boolean tensor is\_valid which concatenates the valid property of the past and current states, and a boolean tensor gt\_future\_is\_valid which concatenates the valid property of the past, current, and future states.

Finally, the function **sets all invalid** object types to **0** and returns a dictionary containing all the extracted information.

**def get\_dataset(path, num\_used\_files, bs, alt\_cizgi = 0):**

**data\_files = os.listdir(path)**

**data\_files = data\_files[alt\_cizgi: alt\_cizgi + num\_used\_files]**

**dataset\_plot = tf.data.TFRecordDataset([os.path.join(path, f) for f in data\_files], num\_parallel\_reads=4)**

**dataset = dataset\_plot.map(\_parse\_model\_ready)**

**dataset = dataset.batch(bs)**

**return dataset, dataset\_plot**

This is a Python function that takes in four arguments: path, num\_used\_files, bs, and alt\_cizgi. It returns two variables, dataset and dataset\_plot.

The purpose of the function is to read a number of TFRecord files from a given directory, parse them using the \_parse\_model\_ready function (which is assumed to be defined elsewhere), and batch them into a TensorFlow dataset with a specified batch size.

Here is a breakdown of the function:

1. os.listdir(path) returns a list of all the files in the specified directory (path).
2. data\_files = data\_files[alt\_cizgi: alt\_cizgi + num\_used\_files] selects a slice of the list, starting from index alt\_cizgi and containing num\_used\_files elements. This allows the function to select only a subset of the files in the directory.
3. [os.path.join(path, f) for f in data\_files] creates a list of full file paths by concatenating the directory path and each file name.
4. tf.data.TFRecordDataset creates a TensorFlow dataset by reading the specified files in parallel (num\_parallel\_reads=4).
5. .map(\_parse\_model\_ready) applies the \_parse\_model\_ready function to each element in the dataset. This function is assumed to take a raw TFRecord as input and return a parsed example that can be fed to a machine learning model.
6. .batch(bs) groups the parsed examples into batches of size bs.
7. The function returns the **batched dataset** (dataset) and the **original dataset** before batching (dataset\_plot). The latter can be useful for debugging purposes, or if you need to access the original examples later on.

**def const\_train\_set(inputs, gt\_func):**

**#Normalizes the input that needs to be normalized**

**normalized\_inputs = normalize(inputs)**

**bs, num\_agents, num\_features, past\_time\_steps = normalized\_inputs['past\_states'].shape**

**time\_steps = past\_time\_steps + 1**

**bs, road\_num, \_ = inputs['roadgraph\_samples/id'].shape**

**# Adds past and current states to input vector**

**X\_past = tf.reshape(normalized\_inputs['past\_states'], [bs, num\_agents \* num\_features, past\_time\_steps])**

**X\_current = tf.reshape(normalized\_inputs['current\_states'], [bs, num\_agents \* num\_features, 1])**

**X\_time = tf.concat([X\_past, X\_current], axis = 2)**

**# Adds object types to input vector**

**A = tf.broadcast\_to(tf.expand\_dims(inputs['object\_type'], axis = 2), shape=[bs, num\_agents, time\_steps])**

**X\_time = tf.concat([X\_time, A], axis=1)**

**A = tf.broadcast\_to(tf.expand\_dims(tf.cast(inputs['tracks\_to\_predict'], dtype = 'float32'), axis = 2), shape=[bs, num\_agents, time\_steps])**

**X\_time = tf.concat([X\_time, A], axis=1)**

**# Adds roadgraph types (e.g. LaneCenter-Freeway, Roadline-SolidSingleYellow)**

**X\_spatial = tf.cast(inputs['roadgraph\_samples/type'], dtype = 'float32')**

**# Adds the direction of the roadgraph**

**A = tf.cast(tf.reshape(inputs['roadgraph\_samples/dir'][:,:,0], (bs, 20000,1)), dtype = 'float32')**

**X\_spatial = tf.concat([X\_spatial, A], axis=1)**

**A = tf.cast(tf.reshape(inputs['roadgraph\_samples/dir'][:,:,1], (bs, 20000,1)), dtype = 'float32')**

**X\_spatial = tf.concat([X\_spatial, A], axis=1)**

**A = tf.cast(tf.reshape(inputs['roadgraph\_samples/dir'][:,:,2], (bs, 20000,1)), dtype = 'float32')**

**X\_spatial = tf.concat([X\_spatial, A], axis=1)**

**Y = gt\_func(inputs, normalized\_inputs)**

**return X\_time, X\_spatial, Y, normalized\_inputs['center\_x'], normalized\_inputs['center\_y']**

This function takes in the inputs and gt\_func as inputs and returns X\_time, X\_spatial, Y, center\_x and center\_y.

inputs represent the input data to the model, which contains the past and current states of the agents, **object types, roadgraph types, and the direction of the roadgraph.**

gt\_func is a function that generates the ground truth values for the model.

normalize function normalizes the input data.

The function first reshapes normalized\_inputs['past\_states'] and normalized\_inputs['current\_states'] to combine the num\_agents and num\_features dimensions. Then, tf.concat() is used to add the current states to the past states along the time dimension.

inputs['object\_type'] is broadcast to have the same shape as X\_time and is concatenated along the feature dimension using tf.concat().

Similarly, inputs['tracks\_to\_predict'] is cast to float32, broadcast to have the same shape as X\_time, and concatenated along the feature dimension using tf.concat().

X\_spatial is initialized with inputs['roadgraph\_samples/type'] and then the direction of the roadgraph is added using tf.concat().

Finally, Y is generated by calling gt\_func on inputs and normalized\_inputs. center\_x and center\_y are also returned from the function.

Overall, this function preprocesses the input data to be used by the model and generates the ground truth values for the model.

**def normalize(inputs):**

**# This function moves all scense to origin**

**# Finds the center values for each scene in the batch. shape of center\_? = (128,)**

**center\_x, center\_y, center\_z = find\_center(inputs['roadgraph\_samples/xyz'], inputs['roadgraph\_samples/valid'])**

**# Normalizes the roadgraph by subtracting the center value.**

**# The transposes are there to make the broadcasting work properly in the correct dimension.**

**# Also extracts the valid values and sets all other values to zero.**

**bs = inputs['roadgraph\_samples/xyz'].shape[0]**

**roadgraph\_x\_norm = tf.transpose(tf.transpose(inputs['roadgraph\_samples/xyz'][:,:,0]) - center\_x)**

**roadgraph\_x\_norm\_valid = tf.where(tf.reshape(inputs['roadgraph\_samples/valid']>0, (bs,-1)), roadgraph\_x\_norm, tf.zeros\_like(roadgraph\_x\_norm))**

**roadgraph\_y\_norm = tf.transpose(tf.transpose(inputs['roadgraph\_samples/xyz'][:,:,1]) - center\_x)**

**roadgraph\_y\_norm\_valid = tf.where(tf.reshape(inputs['roadgraph\_samples/valid']>0, (bs,-1)), roadgraph\_y\_norm, tf.zeros\_like(roadgraph\_y\_norm))**

**roadgraph\_z\_norm = tf.transpose(tf.transpose(inputs['roadgraph\_samples/xyz'][:,:,2]) - center\_x)**

**roadgraph\_z\_norm\_valid = tf.where(tf.reshape(inputs['roadgraph\_samples/valid']>0, (bs,-1)), roadgraph\_z\_norm, tf.zeros\_like(roadgraph\_z\_norm))**

**# Normalizes the past coordinates by subtracting the center value.**

**# The transposes are there to make the broadcasting work properly in the correct dimension.**

**# Also extracts the valid values and sets all other values to zero.**

**bs, num\_agents, \_, past\_timesteps = inputs['past\_xyz'].shape**

**past\_states\_x\_norm = inputs['past\_xyz'][:,:,0,:] - tf.transpose(tf.broadcast\_to(tf.broadcast\_to(center\_x,(num\_agents,bs)), (past\_timesteps,num\_agents,bs)), (2,1,0))**

**past\_states\_x\_norm\_valid = tf.where(inputs['past\_valid'], past\_states\_x\_norm, tf.zeros\_like(past\_states\_x\_norm))**

**past\_states\_y\_norm = inputs['past\_xyz'][:,:,1,:] - tf.transpose(tf.broadcast\_to(tf.broadcast\_to(center\_y,(num\_agents,bs)), (past\_timesteps,num\_agents,bs)), (2,1,0))**

**past\_states\_y\_norm\_valid = tf.where(inputs['past\_valid'], past\_states\_y\_norm, tf.zeros\_like(past\_states\_y\_norm))**

**past\_states\_z\_norm = inputs['past\_xyz'][:,:,2,:] - tf.transpose(tf.broadcast\_to(tf.broadcast\_to(center\_z,(num\_agents,bs)), (past\_timesteps,num\_agents,bs)), (2,1,0))**

**past\_states\_z\_norm\_valid = tf.where(inputs['past\_valid'], past\_states\_z\_norm, tf.zeros\_like(past\_states\_z\_norm))**

**past\_xyz = tf.stack([past\_states\_x\_norm\_valid, past\_states\_y\_norm\_valid, past\_states\_z\_norm\_valid], 2)**

**past\_states = tf.concat([past\_xyz, inputs['past\_states']], 2)**

**# Normalizes the current coordinates by subtracting the center value.**

**# The transposes are there to make the broadcasting work properly in the correct dimension.**

**# Also extracts the valid values and sets all other values to zero.**

**bs, num\_agents, \_, current\_timesteps = inputs['current\_xyz'].shape**

**curr\_states\_x\_norm = inputs['current\_xyz'][:,:,0,:] - tf.transpose(tf.broadcast\_to(tf.broadcast\_to(center\_x,(num\_agents,bs)), (current\_timesteps,num\_agents,bs)), (2,1,0))**

**curr\_states\_x\_norm\_valid = tf.where(inputs['current\_valid'], curr\_states\_x\_norm, tf.zeros\_like(curr\_states\_x\_norm))**

**curr\_states\_y\_norm = inputs['current\_xyz'][:,:,1,:] - tf.transpose(tf.broadcast\_to(tf.broadcast\_to(center\_y,(num\_agents,bs)), (current\_timesteps,num\_agents,bs)), (2,1,0))**

**curr\_states\_y\_norm\_valid = tf.where(inputs['current\_valid'], curr\_states\_y\_norm, tf.zeros\_like(curr\_states\_y\_norm))**

**curr\_states\_z\_norm = inputs['current\_xyz'][:,:,2,:] - tf.transpose(tf.broadcast\_to(tf.broadcast\_to(center\_z,(num\_agents,bs)), (current\_timesteps,num\_agents,bs)), (2,1,0))**

**curr\_states\_z\_norm\_valid = tf.where(inputs['current\_valid'], curr\_states\_z\_norm, tf.zeros\_like(curr\_states\_z\_norm))**

**curr\_xyz = tf.stack([curr\_states\_x\_norm\_valid, curr\_states\_y\_norm\_valid, curr\_states\_z\_norm\_valid], 2)**

**curr\_states = tf.concat([curr\_xyz, inputs['current\_states']], 2)**

**# Normalizes the future coordinates by subtracting the center value.**

**# The transposes are there to make the broadcasting work properly in the correct dimension.**

**# Also extracts the valid values and sets all other values to zero.**

**bs, num\_agents, \_, future\_timesteps = inputs['future\_xyz'].shape**

**future\_states\_x\_norm = inputs['future\_xyz'][:,:,0,:] - tf.transpose(tf.broadcast\_to(tf.broadcast\_to(center\_x,(num\_agents,bs)), (future\_timesteps,num\_agents,bs)), (2,1,0))**

**future\_states\_x\_norm\_valid = tf.where(inputs['future\_valid'], future\_states\_x\_norm, tf.zeros\_like(future\_states\_x\_norm))**

**future\_states\_y\_norm = inputs['future\_xyz'][:,:,1,:] - tf.transpose(tf.broadcast\_to(tf.broadcast\_to(center\_y,(num\_agents,bs)), (future\_timesteps,num\_agents,bs)), (2,1,0))**

**future\_states\_y\_norm\_valid = tf.where(inputs['future\_valid'], future\_states\_y\_norm, tf.zeros\_like(future\_states\_y\_norm))**

**future\_states\_z\_norm = inputs['future\_xyz'][:,:,2,:] - tf.transpose(tf.broadcast\_to(tf.broadcast\_to(center\_z,(num\_agents,bs)), (future\_timesteps,num\_agents,bs)), (2,1,0))**

**future\_states\_z\_norm\_valid = tf.where(inputs['future\_valid'], future\_states\_z\_norm, tf.zeros\_like(future\_states\_z\_norm))**

**future\_xyz = tf.stack([future\_states\_x\_norm\_valid, future\_states\_y\_norm\_valid, future\_states\_z\_norm\_valid], 2)**

**future\_states = tf.concat([future\_xyz, inputs['future\_states']], 2)**

**normalized\_inputs = {**

**'past\_states': past\_states,**

**'current\_states': curr\_states,**

**'future\_states': future\_states,**

**'roadgraph\_samples/x': tf.reshape(roadgraph\_x\_norm\_valid, (bs,-1,1)),**

**'roadgraph\_samples/y': tf.reshape(roadgraph\_y\_norm\_valid, (bs,-1,1)),**

**'roadgraph\_samples/z': tf.reshape(roadgraph\_z\_norm\_valid, (bs,-1,1)),**

**# For finding the gt in training and validation set**

**'final\_x': tf.reshape(future\_states\_x\_norm\_valid[:,:,-1], (bs, num\_agents, 1)),**

**'final\_y': tf.reshape(future\_states\_y\_norm\_valid[:,:,-1], (bs, num\_agents, 1)),**

**'center\_x': center\_x,**

**'center\_y': center\_y**

**}**

**return normalized\_inputs**

It looks like the code you shared is a function called "normalize" that takes in some input data, and it applies some normalization operations to different parts of the data. Specifically, it finds the center values of each scene in the batch, and then normalizes the past, present, and future coordinates by subtracting the center value.

The input data is provided as a dictionary, where the keys correspond to different types of data. For example, "roadgraph\_samples/xyz" is the key for the xyz coordinates of the road network graph. "past\_xyz" and "current\_xyz" are the keys for the past and current coordinates of the agents, respectively.

The function uses TensorFlow to perform these operations, and it makes use of several TensorFlow functions such as "tf.transpose", "tf.broadcast\_to", and "tf.where". These operations are performed on tensors that represent the data, and the resulting tensors are returned as the output of the function.

**def find\_center(roadgraph\_xyz, roadgraph\_valid):**

**#finds the center of the map by finding the largest and the smallest value of the roads, and average these values.**

**#Finds the max for each scene in the batch by taking the max over the second dimension (the road values).**

**#Invalid data is filled with large negative value in order to not be the maximum value and thus influence the result.**

**#The same approach is used for finding the minimum value but with a large positive value.**

**max\_x = tf.reduce\_max(tf.where(roadgraph\_valid[:,:,0]>0, roadgraph\_xyz[:,:,0], tf.ones\_like(roadgraph\_xyz[:,:,0])\*-10000000), 1)**

**min\_x = tf.reduce\_min(tf.where(roadgraph\_valid[:,:,0]>0, roadgraph\_xyz[:,:,0], tf.ones\_like(roadgraph\_xyz[:,:,0])\*10000000), 1)**

**max\_y = tf.reduce\_max(tf.where(roadgraph\_valid[:,:,0]>0, roadgraph\_xyz[:,:,1], tf.ones\_like(roadgraph\_xyz[:,:,1])\*-10000000), 1)**

**min\_y = tf.reduce\_min(tf.where(roadgraph\_valid[:,:,0]>0, roadgraph\_xyz[:,:,1], tf.ones\_like(roadgraph\_xyz[:,:,1])\*10000000), 1)**

**max\_z = tf.reduce\_max(tf.where(roadgraph\_valid[:,:,0]>0, roadgraph\_xyz[:,:,2], tf.ones\_like(roadgraph\_xyz[:,:,2])\*-10000000), 1)**

**min\_z = tf.reduce\_min(tf.where(roadgraph\_valid[:,:,0]>0, roadgraph\_xyz[:,:,2], tf.ones\_like(roadgraph\_xyz[:,:,2])\*10000000), 1)**

**center\_x = (max\_x + min\_x)/2**

**center\_y = (max\_y + min\_y)/2**

**center\_z = (max\_z + min\_z)/2**

**return center\_x, center\_y, center\_z**

This is a function that takes in two arguments, roadgraph\_xyz and roadgraph\_valid. The roadgraph\_xyz argument is a tensor containing the xyz coordinates of **road segments in a map**, while roadgraph\_valid is a tensor that specifies which **road segments are valid**.

The function calculates the **center of the map** by finding the **maximum and minimum values of the x, y, and z coordinates of the valid road segments. Invalid segments are represented by a large negative value for x and y, and a large positive value for z.**

The **maximum and minimum values** of each coordinate are found by using the tf.reduce\_max and tf.reduce\_min functions, respectively. tf.where is used to **mask out invalid segments by replacing their values with the specified large values.** The center of the map is then calculated by averaging the maximum and minimum values of each coordinate.

The function returns three tensors representing the x, y, and z coordinates of the center of the map.

## **Construct the LSTM model for end point prediction**

**def GoalPredModel(num\_states\_steps, latent\_dim, num\_agents\_to\_predict, num\_end\_points):**

**"""Goal Prediction model**

**param num\_states\_steps: 10 past + 1 current scenario**

**param lated\_dim: the dimension of the latent diemsions between the LSTM blocks**

**param num\_agents\_to\_predict: Number of agents that the system needs to predict**

**param num\_end\_points: Number of end points the system needs to predict = 2 (x and y coordinate for 1 point)**

**"""**

**#The length of the large input vector. Depends on how many features we include.**

**input\_vec\_length = 1280**

**num\_static\_features = 20000\*4**

**time\_inputs = Input(shape=(input\_vec\_length, num\_states\_steps))**

**static\_inputs = Input(shape=(num\_static\_features,1))**

**a0 = Input(shape=(latent\_dim,), name='a0')**

**c0 = Input(shape=(latent\_dim,), name='c0')**

**a = a0**

**c = c0**

**# Create the encoder**

**for t in range(num\_states\_steps):**

**# Select the "t"th time step vector from enc\_input.**

**x\_time = Lambda(lambda z: z[:, :, t])(time\_inputs)**

**# Reshape x**

**x\_time = Reshape((1, input\_vec\_length))(x\_time)**

**x\_time = BatchNormalization()(x\_time)**

**# LSTM\_cell**

**a, \_, c = LSTM(latent\_dim, kernel\_initializer=GlorotUniform(), return\_state = True)(x\_time, initial\_state=[a, c])**

**#Convolve spatial features**

**x\_spatial = Conv1D(4, 5, strides = 2, padding = 'same', kernel\_initializer=GlorotUniform())(static\_inputs)**

**x\_spatial = MaxPool1D(pool\_size=2, strides=2)(x\_spatial)**

**x\_spatial = Conv1D(8, 5, strides = 2, padding = 'same', kernel\_initializer=GlorotUniform())(x\_spatial)**

**x\_spatial = MaxPool1D(pool\_size=2, strides=2)(x\_spatial)**

**x\_spatial = Conv1D(8, 5, strides = 2, padding = 'same', kernel\_initializer=GlorotUniform())(x\_spatial)**

**x\_spatial = MaxPool1D(pool\_size=2, strides=2)(x\_spatial)**

**x\_spatial = Conv1D(16, 5, strides = 2, padding = 'same', kernel\_initializer=GlorotUniform())(x\_spatial)**

**x\_spatial = MaxPool1D(pool\_size=2, strides=2)(x\_spatial)**

**x\_spatial = tf.keras.layers.Flatten()(x\_spatial)**

**x\_spatial = Dropout(rate=DROPOUT\_RATE\_SPATIAL)(x\_spatial)**

**a = tf.keras.layers.Flatten()(a)**

**x = tf.concat([a, x\_spatial], axis = 1)**

**# To hold the outputs**

**outputs = []**

**out = Dense(num\_agents\_to\_predict\*num\_end\_points, kernel\_initializer=GlorotUniform())(x)**

**outputs.append(out)**

**model = Model(inputs=[time\_inputs, static\_inputs, a0, c0], outputs=outputs)**

**return model**

This code defines a function GoalPredModel which creates a neural network model for predicting the goals of agents in a scenario.

The model takes in four parameters:

* num\_states\_steps: an integer representing the number of previous time steps (past states) plus the current state that the model should consider for each agent in a scenario.
* latent\_dim: an integer representing the dimension of the latent space between the LSTM blocks in the model.
* num\_agents\_to\_predict: an integer representing the number of agents the system needs to predict the goals for.
* num\_end\_points: an integer representing the number of end points that the system needs to predict for each agent (in this case, 2, the x and y coordinates of a point).

The function first creates three inputs: time\_inputs for the temporal input (i.e., the sequence of states), static\_inputs for the spatial input (i.e., the static features), a0 and c0 for the initial state of the LSTM cells.

Then, the function creates an encoder using a for loop to process each time step. The Lambda layer selects the tth time step vector from time\_inputs and Reshape changes its shape to (1, input\_vec\_length). The LSTM layer processes this vector and updates the LSTM cell states a and c.

After the encoder, the function creates a spatial convolutional neural network using Conv1D and MaxPool1D layers to convolve the spatial features. Then, these features are flattened and concatenated with the LSTM cell state a.

Finally, the function creates an output layer using Dense that outputs a tensor of shape (num\_agents\_to\_predict\*num\_end\_points,) for each agent in the scenario.

The function returns a Keras Model object that takes the four inputs (time\_inputs, static\_inputs, a0, and c0) and outputs a list of tensors (outputs).

**x\_spatial = Conv1D(4, 5, strides = 2, padding = 'same', kernel\_initializer=GlorotUniform())(static\_inputs)**

This line of code applies a 1D convolution operation with **4 filters of size 5** on the static\_inputs tensor. The convolution has a stride of 2 and the output has the same shape as the input due to the padding='same' parameter. **The weights of the kernel are initialized using the Glorot uniform initializer.**

In this model, static\_inputs is a tensor of shape (num\_static\_features, 1), where num\_static\_features is the number of static features in the input. The output of this convolutional layer will have shape (num\_static\_features, 4) which is the result of applying the convolution operation to each of the 4 filters.

→**LSTM**

LSTM stands for **Long Short-Term Memory** and is a **type of recurrent neural network (RNN)** architecture that is designed to handle the vanishing and exploding gradient problems that are common in traditional RNNs.

LSTM models are commonly used in motion prediction tasks for autonomous vehicles, robotics, and other applications where it is necessary to predict the future motion of objects. The LSTM model can process sequential data, such as time-series data or video frames, and learn the patterns in the data to predict the future behavior of objects.

In the context of motion prediction, the **LSTM model takes as input the past trajectory of an object, such as the position and velocity at previous time steps, and uses this information to predict the future trajectory of the object**. The model can also take into account other factors such as the current speed and direction of the object, the behavior of other objects in the scene, and environmental factors such as road conditions or obstacles.

LSTM models have shown promising results in motion prediction tasks, with some studies reporting high accuracy in predicting the future trajectories of objects. However, these models also have some limitations, such as their sensitivity to changes in the input data and the need for large amounts of training data to learn the complex patterns in the data.

Despite these limitations, LSTM models remain a popular choice for motion prediction tasks, and ongoing research continues to explore ways to improve the accuracy and efficiency of these models.

→**Get the ground truth data**

**def get\_gt\_final(inputs, normalized\_inputs):**

**"""Returns a concatenated tensor of shape (bs,256,1), where 256=128\*2 (for x and y coordinate)**

**The only values that are non-zero in the return tensor are the ones that we actually are interested in.**

**Observe the first 8 of the indicies represent the x and the last 8 represents the y coordinate**

**"""**

**Yx = normalized\_inputs['final\_x']**

**Yy = normalized\_inputs['final\_y']**

**tracks = inputs['tracks\_to\_predict']**

**Yx = tf.math.multiply(Yx, tf.expand\_dims(tf.cast(tracks, dtype = 'float32'), axis = 2))**

**Yx = Yx[:,0:8,:]**

**Yy = tf.math.multiply(Yy, tf.expand\_dims(tf.cast(tracks, dtype = 'float32'), axis = 2))**

**Yy = Yy[:,0:8,:]**

**Y = tf.concat([Yx, Yy], axis = 1)**

**return Y**

The get\_gt\_final function takes two inputs, inputs and normalized\_inputs. inputs is a dictionary that contains the tracks to predict, while normalized\_inputs contains the final x and y coordinates of the tracks after normalization. The function returns a concatenated tensor of shape (bs, 256, 1) where bs represents the batch size and 256=128\*2 represents the x and y coordinates of the final position.

In the function, **first, the final x and y coordinates are multiplied with the tracks to predict using element-wise multiplication to get the coordinates of only the tracks that are to be predicted.** Then, the first eight elements of the resulting Yx tensor represent the x coordinates of the final positions, while the last eight elements of the Yy tensor represent the y coordinates. Finally, these tensors are concatenated along the second dimension (i.e., the x and y coordinates) to get the final concatenated tensor Y.

The returned **tensor Y contains only the coordinates of the tracks that are to be predicted, and the remaining values are set to zero.** This function is commonly used in motion prediction tasks to extract the ground truth labels for training the model.

**def create\_figure\_and\_axes(size\_pixels):**

**"""Initializes a unique figure and axes for plotting."""**

**fig, ax = plt.subplots(1, 1, num=uuid.uuid4())**

**# Sets output image to pixel resolution.**

**dpi = 100**

**size\_inches = size\_pixels / dpi**

**fig.set\_size\_inches([size\_inches, size\_inches])**

**fig.set\_dpi(dpi)**

**fig.set\_facecolor('white')**

**ax.set\_facecolor('white')**

**ax.xaxis.label.set\_color('black')**

**ax.tick\_params(axis='x', colors='black')**

**ax.yaxis.label.set\_color('black')**

**ax.tick\_params(axis='y', colors='black')**

**fig.set\_tight\_layout(True)**

**ax.grid(False)**

**return fig, ax**

The create\_figure\_and\_axes function initializes a new figure and axes for plotting. The function takes one input, size\_pixels, which represents the size of the output image in pixels.

First, the function creates a new figure and axes using the subplots function from the matplotlib.pyplot library. The num argument is set to a random UUID value to ensure that each figure is unique.

Next, the function sets the output image resolution to dpi (dots per inch) and the figure size in inches based on the size\_pixels input. The background color of the figure and axes are set to white, and the tick and label colors for both x and y axes are set to black.

The function then **sets the tight layout to ensure that all elements of the plot are visible and removes the grid lines**. Finally, the function returns the figure and axes objects for further use in plotting.

This function is useful when plotting multiple figures to avoid overlap and ensure each figure is clearly visible.

**UUID** →Universally Unique Identifier. It is a 128-bit value used to identify information in computer systems.

**def fig\_canvas\_image(fig):**

**"""Returns a [H, W, 3] uint8 np.array image from fig.canvas.tostring\_rgb()."""**

**# Just enough margin in the figure to display xticks and yticks.**

**fig.subplots\_adjust(**

**left=0.08, bottom=0.08, right=0.98, top=0.98, wspace=0.0, hspace=0.0)**

**fig.canvas.draw()**

**data = np.frombuffer(fig.canvas.tostring\_rgb(), dtype=np.uint8)**

**return data.reshape(fig.canvas.get\_width\_height()[::-1] + (3,))**

The fig\_canvas\_image function takes one input, fig, which is a matplotlib figure object. The function returns a 3D NumPy array representing the RGB image of the figure.

To create the image, the function first adjusts the margins of the figure to ensure that the x and y ticks are visible. It then calls the canvas.draw() method to render the figure onto the canvas. The canvas.tostring\_rgb() method returns a byte string representation of the RGB image. This byte string is then converted into a 1D NumPy array of unsigned integers with the np.frombuffer() method. Finally, the 1D array is reshaped into a 3D array of shape (height, width, 3) using the reshape() method, where the last dimension represents the RGB color channels.

The resulting NumPy array represents the RGB image of the figure and can be saved or displayed using any method that supports 3D arrays. This function is useful when working with matplotlib figures in applications where the output needs to be saved or displayed as an image.

**def get\_colormap(num\_agents):**

**"""Compute a color map array of shape [num\_agents, 4]."""**

**colors = cm.get\_cmap('jet', num\_agents)**

**colors = colors(range(num\_agents))**

**np.random.shuffle(colors)**

**colors[0] = [0.,0.,0.5,1.,]**

**colors[1] = [0.5,0.,0.,1.,]**

**colors[2] = [0.,0.5,0.,1.,]**

**colors[3] = [0.5,0.5,0.,1.,]**

**colors[4] = [0.5,0.,0.5,1.,]**

**colors[5] = [0.,0.5,0.5,1.,]**

**colors[6] = [0.2,0.7,0.1,1.,]**

**colors[7] = [0.5,0.5,0.5,1.,]**

**return colors**

The get\_colormap function takes one input, num\_agents, which is an integer representing the number of agents to be plotted. The function returns a NumPy array of shape [num\_agents, 4] representing the colormap for the agents.

The function first calls cm.get\_cmap('jet', num\_agents) to generate a colormap with num\_agents colors. The range() method is then called on the colormap object to generate an array of evenly spaced values between 0 and 1. The colors are shuffled randomly using np.random.shuffle().

The first eight colors are set to specific RGB values, presumably to ensure that the colors are distinguishable and visually appealing. The returned array has four columns representing the RGBA values for each color. The **first three columns** represent the **RGB** values, and the **fourth column** represents the **opacity (alpha) value**.

**def get\_viewport(all\_states, all\_states\_mask):**

**"""Gets the region containing the data.**

**Args:**

**all\_states: states of agents as an array of shape [num\_agents, num\_steps,**

**2].**

**all\_states\_mask: binary mask of shape [num\_agents, num\_steps] for**

**`all\_states`.**

**Returns:**

**center\_y: float. y coordinate for center of data.**

**center\_x: float. x coordinate for center of data.**

**width: float. Width of data.**

**"""**

**valid\_states = all\_states[all\_states\_mask]**

**all\_y = valid\_states[..., 1]**

**all\_x = valid\_states[..., 0]**

**center\_y = (np.max(all\_y) + np.min(all\_y)) / 2**

**center\_x = (np.max(all\_x) + np.min(all\_x)) / 2**

**range\_y = np.ptp(all\_y)**

**range\_x = np.ptp(all\_x)**

**width = max(range\_y, range\_x)**

**return center\_y, center\_x, width**

This is a function that calculates the center and width of a viewport for a given set of agent states and their binary mask. The agent states are given as an array of shape [num\_agents, num\_steps, 2], where each row corresponds to an agent and each column corresponds to a time step. The first and second columns represent the x and y coordinates of the agent, respectively. The binary mask is also an array of shape [num\_agents, num\_steps], where each element is either 0 or 1 depending on whether the corresponding agent state is valid or not.

The function first selects only the valid agent states based on the binary mask and then calculates the center and width of a rectangle that encompasses all the valid agent states. The center of the rectangle is calculated as the midpoint between the maximum and minimum y and x coordinates of the valid agent states. The width of the rectangle is then determined as the maximum of the ranges of the y and x coordinates. Finally, the function returns the calculated center\_y, center\_x, and width values as output. These values can be used to set the limits of the x and y axes of a plot.

**def visualize\_one\_step(states,**

**mask,**

**roadgraph,**

**title,**

**center\_y,**

**center\_x,**

**width,**

**color\_map,**

**size\_pixels=1000):**

**"""Generate visualization for a single step."""**

**# Create figure and axes.**

**fig, ax = create\_figure\_and\_axes(size\_pixels=size\_pixels)**

**# Plot roadgraph.**

**rg\_pts = roadgraph[:, :2].T**

**ax.plot(rg\_pts[0, :], rg\_pts[1, :], 'k.', alpha=1, ms=2)**

**masked\_x = states[:, 0][mask]**

**masked\_y = states[:, 1][mask]**

**colors = color\_map[mask]**

**# Plot agent current position.**

**ax.scatter(**

**masked\_x,**

**masked\_y,**

**marker='o',**

**linewidths=3,**

**color=colors,**

**)**

**# Title.**

**ax.set\_title(title)**

**# Set axes. Should be at least 10m on a side and cover 160% of agents.**

**size = max(10, width \* 1.0)**

**ax.axis([**

**-size / 2 + center\_x, size / 2 + center\_x, -size / 2 + center\_y,**

**size / 2 + center\_y**

**])**

**ax.set\_aspect('equal')**

**image = fig\_canvas\_image(fig)**

**plt.close(fig)**

**return image**

This function generates a visualization of a single step of an agent trajectory on a roadgraph. It takes the following arguments:

* states: a numpy array of shape [num\_agents, num\_steps, 2] representing the positions of the agents at each step. The second dimension is ignored, since we are only visualizing a single step.
* mask: a boolean array of shape [num\_agents] representing which agents to include in the visualization.
* roadgraph: a numpy array of shape [num\_edges, 4] representing the edges of the roadgraph. The first two columns represent the x and y coordinates of the starting point of the edge, and the last two columns represent the x and y coordinates of the ending point of the edge.
* title: a string representing the title of the visualization.
* center\_y, center\_x, and width: floats representing the center y and x coordinates and the width of the visualization region, respectively. These are obtained from the get\_viewport function.
* color\_map: a numpy array of shape [num\_agents, 4] representing the color to use for each agent.
* size\_pixels: an integer representing the pixel size of the output image.

The function first creates a figure and axes using the create\_figure\_and\_axes function. It then plots the roadgraph using black dots, and plots the positions of the selected agents using circles with a color based on the color\_map argument. It sets the title of the plot, and sets the x and y limits of the axes based on the center\_y, center\_x, and width arguments. Finally, it generates a PNG image of the plot using the fig\_canvas\_image function and returns it.

**def visualize\_all\_agents\_smooth(decoded\_example,**

**size\_pixels=1000,**

**):**

**"""Visualizes all agent predicted trajectories in a series of images.**

**Args:**

**decoded\_example: Dictionary containing agent info about all modeled agents.**

**size\_pixels: The size in pixels of the output image.**

**Returns:**

**T of [H, W, 3] uint8 np.arrays of the drawn matplotlib's figure canvas.**

**"""**

**# [num\_agents, num\_past\_steps, 2] float32.**

**past\_states = tf.stack(**

**[decoded\_example['state/past/x'], decoded\_example['state/past/y']],**

**-1).numpy()**

**num\_agents, num\_past\_steps, \_ = past\_states.shape**

**past\_states\_mask = decoded\_example['state/past/valid'].numpy() > 0.0**

**# [num\_agents, 1, 2] float32.**

**current\_states = tf.stack(**

**[decoded\_example['state/current/x'], decoded\_example['state/current/y']],**

**-1).numpy()**

**current\_states\_mask = decoded\_example['state/current/valid'].numpy() > 0.0**

**# [num\_agents, num\_future\_steps, 2] float32.**

**future\_states = tf.stack(**

**[decoded\_example['state/future/x'], decoded\_example['state/future/y']],**

**-1).numpy()**

**num\_future\_steps = future\_states.shape[1]**

**future\_states\_mask = decoded\_example['state/future/valid'].numpy() > 0.0**

**# [num\_points, 3] float32.**

**roadgraph\_xyz = decoded\_example['roadgraph\_samples/xyz'].numpy()**

**color\_map = get\_colormap(num\_agents)**

**# [num\_agens, num\_past\_steps + 1 + num\_future\_steps, depth] float32.**

**all\_states = np.concatenate([past\_states, current\_states, future\_states], 1)**

**# [num\_agens, num\_past\_steps + 1 + num\_future\_steps] float32.**

**all\_states\_mask = np.concatenate(**

**[past\_states\_mask, current\_states\_mask, future\_states\_mask], 1)**

**center\_y, center\_x, width = get\_viewport(all\_states, all\_states\_mask)**

**images = []**

**# Generate images from past time steps.**

**for i, (s, m) in enumerate(**

**zip(**

**np.split(past\_states, num\_past\_steps, 1),**

**np.split(past\_states\_mask, num\_past\_steps, 1))):**

**im = visualize\_one\_step(s[:, 0], m[:, 0], roadgraph\_xyz,**

**'past: %d' % (num\_past\_steps - i), center\_y,**

**center\_x, width, color\_map, size\_pixels)**

**images.append(im)**

**# Generate one image for the current time step.**

**s = current\_states**

**m = current\_states\_mask**

**im = visualize\_one\_step(s[:, 0], m[:, 0], roadgraph\_xyz, 'current', center\_y,**

**center\_x, width, color\_map, size\_pixels)**

**images.append(im)**

**predict\_agent\_mask = decoded\_example['state/tracks\_to\_predict'].numpy() > 0.0**

**predict\_agent\_mask = tf.broadcast\_to(tf.expand\_dims(predict\_agent\_mask, axis=1), [num\_agents, num\_future\_steps])**

**future\_states\_mask = np.logical\_and(future\_states\_mask, predict\_agent\_mask)**

**print(future\_states\_mask[0:10,-1])**

**# Generate images from future time steps.**

**for i, (s, m) in enumerate(**

**zip(**

**np.split(future\_states, num\_future\_steps, 1),**

**np.split(future\_states\_mask, num\_future\_steps, 1))):**

**im = visualize\_one\_step(s[:, 0], m[:, 0], roadgraph\_xyz,**

**'future: %d' % (i + 1), center\_y, center\_x, width,**

**color\_map, size\_pixels)**

**images.append(im)**

**return images**

This function visualizes all agent predicted trajectories in a series of images. It takes in a dictionary containing agent info about all modeled agents and the size in pixels of the output image.

First, it extracts past, current, and future states and their respective masks from the input dictionary. It also extracts the road graph, which is the path taken by the agents.

Next, it concatenates all the states and their masks into one array each. Then it gets the center coordinates and the width of the viewport for the road graph.

It then generates images for past time steps, current time step, and future time steps. For each time step, it calls visualize\_one\_step function, which generates the visualization for a single step. It passes the masked x and y coordinates of the agents in that time step, the road graph, the title of the image, the center and width of the viewport, and the color map to that function. It then appends the generated image to a list of images.

Finally, it returns the list of images.

**def visualize\_all\_agents\_smooth\_w\_np(decoded\_example, prediction\_file\_x, prediction\_file\_y,**

**size\_pixels=1000,):**

**"""Visualizes all agent predicted trajectories in a series of images.**

**Args:**

**decoded\_example: Dictionary containing agent info about all modeled agents.**

**size\_pixels: The size in pixels of the output image.**

**Returns:**

**T of [H, W, 3] uint8 np.arrays of the drawn matplotlib's figure canvas.**

**"""**

**# [num\_agents, num\_past\_steps, 2] float32.**

**past\_states = tf.stack(**

**[decoded\_example['state/past/x'], decoded\_example['state/past/y']],**

**-1).numpy()**

**num\_agents, num\_past\_steps, \_ = past\_states.shape**

**past\_states\_mask = decoded\_example['state/past/valid'].numpy() > 0.0**

**# [num\_agents, 1, 2] float32.**

**current\_states = tf.stack(**

**[decoded\_example['state/current/x'], decoded\_example['state/current/y']],**

**-1).numpy()**

**current\_states\_mask = decoded\_example['state/current/valid'].numpy() > 0.0**

**# [num\_agents, num\_future\_steps, 2] float32.**

**future\_states = np.stack(**

**[prediction\_file\_x, prediction\_file\_y],**

**-1)**

**num\_future\_steps = future\_states.shape[1]**

**future\_states\_mask = decoded\_example['state/future/valid'].numpy() > 0.0**

**# [num\_points, 3] float32.**

**roadgraph\_xyz = decoded\_example['roadgraph\_samples/xyz'].numpy()**

**color\_map = get\_colormap(num\_agents)**

**# [num\_agens, num\_past\_steps + 1 + num\_future\_steps, depth] float32.**

**all\_states = np.concatenate([past\_states, current\_states, future\_states], 1)**

**# [num\_agens, num\_past\_steps + 1 + num\_future\_steps] float32.**

**all\_states\_mask = np.concatenate(**

**[past\_states\_mask, current\_states\_mask, future\_states\_mask], 1)**

**center\_y, center\_x, width = get\_viewport(all\_states, all\_states\_mask)**

**images = []**

**# Generate images from past time steps.**

**for i, (s, m) in enumerate(**

**zip(**

**np.split(past\_states, num\_past\_steps, 1),**

**np.split(past\_states\_mask, num\_past\_steps, 1))):**

**im = visualize\_one\_step(s[:, 0], m[:, 0], roadgraph\_xyz,**

**'past: %d' % (num\_past\_steps - i), center\_y,**

**center\_x, width, color\_map, size\_pixels)**

**images.append(im)**

**# Generate one image for the current time step.**

**s = current\_states**

**m = current\_states\_mask**

**im = visualize\_one\_step(s[:, 0], m[:, 0], roadgraph\_xyz, 'current', center\_y,**

**center\_x, width, color\_map, size\_pixels)**

**images.append(im)**

**predict\_agent\_mask = decoded\_example['state/tracks\_to\_predict'].numpy() > 0.0**

**predict\_agent\_mask = tf.broadcast\_to(tf.expand\_dims(predict\_agent\_mask, axis=1), [num\_agents, num\_future\_steps])**

**future\_states\_mask = np.logical\_and(future\_states\_mask, predict\_agent\_mask)**

**#print(future\_states\_mask[0:10,-1])**

**# Generate images from future time steps.**

**for i, (s, m) in enumerate(**

**zip(**

**np.split(future\_states, num\_future\_steps, 1),**

**np.split(future\_states\_mask, num\_future\_steps, 1))):**

**im = visualize\_one\_step(s[:, 0], m[:, 0], roadgraph\_xyz,**

**'future: %d' % (i + 1), center\_y, center\_x, width,**

**color\_map, size\_pixels)**

**images.append(im)**

**return images**

**This function takes in information about all the modeled agents, including their past and future trajectories, and creates a series of images that visualize the predicted trajectories of all the agents.** The function first extracts the past, current, and future states of all agents and their respective masks. It then concatenates these states and masks to create a single tensor for each. The function then determines the center and width of the viewport for the visualization using the concatenated tensor. Next, the function generates images for each time step in the past and future by calling the visualize\_one\_step function for each step. The resulting images are returned as a list.

**def calculate\_plots(visualization\_file, predicted\_values\_x, predicted\_values\_y, gt\_values\_x, gt\_values\_y):**

**# Plot the end point for the agents we are interested in. First for the GT case (not the predicted one).**

**#images = visualize\_all\_agents\_smooth(visualization\_file)**

**#plt.rcParams['figure.figsize'] = [10, 10]**

**#plt.imshow(images[-1])**

**#plt.show()**

**# Plot the end point for the agents we are interested in. Second for the predicted one.**

**gt\_file\_x = visualization\_file['state/future/x'].numpy()**

**gt\_file\_x[:,-1][0:8] = gt\_values\_x**

**gt\_file\_y = visualization\_file['state/future/y'].numpy()**

**gt\_file\_y[:,-1][0:8] = gt\_values\_y**

**images = visualize\_all\_agents\_smooth\_w\_np(visualization\_file, gt\_file\_x, gt\_file\_y)**

**plt.rcParams['figure.figsize'] = [10, 10]**

**plt.imshow(images[-1])**

**plt.show()**

**# Plot the end point for the agents we are interested in. Second for the predicted one.**

**prediction\_file\_x = visualization\_file['state/future/x'].numpy()**

**prediction\_file\_x[:,-1][0:8] = predicted\_values\_x**

**prediction\_file\_y = visualization\_file['state/future/y'].numpy()**

**prediction\_file\_y[:,-1][0:8] = predicted\_values\_y**

**images = visualize\_all\_agents\_smooth\_w\_np(visualization\_file, prediction\_file\_x, prediction\_file\_y)**

**plt.rcParams['figure.figsize'] = [10, 10]**

**plt.imshow(images[-1])**

**plt.show()**

This function seems to take a **visualization file and two sets of predicted values and ground truth values for the x and y coordinates of certain agents,** and then it **generates two images showing the predicted trajectories and ground truth trajectories for these agents.**

The function first modifies the visualization file to replace the last position for the specified agents with the predicted or ground truth values. Then, it uses the modified file to generate the two images by calling the visualize\_all\_agents\_smooth\_w\_np function. Finally, it plots the images using matplotlib.

It is worth noting that the visualize\_all\_agents\_smooth\_w\_np function seems to take a lot of arguments and perform some complex operations to generate the visualizations.

**def train\_1\_epoch(epoch, train\_dataset):**

**# Set the batch losses to be an empty list to only contain losses from this epochs batches.**

**logits\_visualize, center\_x\_visualize, center\_y\_visualize, y\_gt\_visualize = 0, 0, 0, 0**

**train\_batch\_losses = []**

**# Iterate over the batches of the dataset.**

**for step, batch in enumerate(train\_dataset):**

**# Send in 1 batch to the function train\_step batch.**

**# 'batch' is a DICTIONARY containing the past position, velocity, current position,**

**# velocity etc. for all the objects in the scenario for all the time indices for**

**# all batches.**

**train\_batch\_time, train\_batch\_static, y\_batch\_train, center\_x, center\_y = const\_train\_set(batch, get\_gt\_final)**

**bs = train\_batch\_time.shape[0]**

**a0 = np.zeros((bs, LATENT\_DIM))**

**c0 = np.zeros((bs, LATENT\_DIM))**

**with tf.GradientTape() as tape:**

**# Run the forward pass**

**logits = model([train\_batch\_time, train\_batch\_static ,a0,c0], training=True)**

**y\_batch\_train = tf.reshape(y\_batch\_train, (y\_batch\_train.shape[0:2]))**

**# Saves gt and predicted values to be able to visualize the results**

**if step == STEP\_TO\_VISUALIZE and epoch % EPOCH\_TO\_VISUALIZE == 0:# and epoch != 0:**

**y\_gt\_visualize = y\_batch\_train**

**logits\_visualize = logits**

**center\_x\_visualize = center\_x**

**center\_y\_visualize = center\_y**

**# Compute the loss value**

**weights = batch['tracks\_to\_predict'][:,:8]**

**weights = tf.concat([weights, weights], axis = 1)**

**valid\_logits = tf.boolean\_mask(logits, weights)**

**valid\_gt = tf.boolean\_mask(y\_batch\_train, weights)**

**loss\_value = loss\_fn(valid\_gt, valid\_logits)**

**# Use the gradient tape to automatically retrieve**

**# the gradients of the trainable variables with respect to the loss.**

**grads = tape.gradient(loss\_value, model.trainable\_weights)**

**# Run one step of gradient descent by updating**

**# the value of the variables to minimize the loss**

**optimizer.apply\_gradients(zip(grads, model.trainable\_weights))**

**train\_batch\_losses.append(loss\_value)**

**# Prints the training loss after each epoch**

**print("The mean training loss after %d epochs: %.4f"% (epoch, float(sum(train\_batch\_losses)/len(train\_batch\_losses))))**

**train\_epoch\_losses.append(float(sum(train\_batch\_losses)/len(train\_batch\_losses)))**

**return logits\_visualize, center\_x\_visualize, center\_y\_visualize, y\_gt\_visualize**

This function train\_1\_epoch trains the model for one epoch on the training dataset. It takes two arguments: epoch and train\_dataset.

Within the function, it initializes the train\_batch\_losses list to collect the losses from each batch of the training dataset. It then iterates over the batches of the train\_dataset, retrieves the data for that batch, and runs the forward pass of the model to get the predicted values.

After that, it computes the loss value using the loss function and the ground truth data for that batch. The loss is then appended to train\_batch\_losses. It uses the GradientTape to calculate the gradients of the trainable weights with respect to the loss value. Then it updates the trainable weights by applying the optimizer.

Finally, **it calculates the mean loss of the current epoch, saves the losses in the train\_epoch\_losses list, and returns the values needed for visualization.**

**def calc\_val\_loss(valid\_dataset):**

**valid\_batch\_losses = []**

**logits\_visualize\_valid, center\_x\_valid\_visualize, center\_y\_valid\_visualize, y\_gt\_visualize\_valid = 0,0,0,0**

**for step, batch in enumerate(valid\_dataset):**

**valid\_time\_batch, valid\_static\_batch,y\_batch\_valid, center\_x\_valid, center\_y\_valid = const\_train\_set(batch, get\_gt\_final)**

**bs = valid\_time\_batch.shape[0]**

**a0 = np.zeros((bs, LATENT\_DIM))**

**c0 = np.zeros((bs, LATENT\_DIM))**

**with tf.GradientTape() as tape:**

**# Run the forward pass of the layer.**

**logits = model([valid\_time\_batch,valid\_static\_batch,a0,c0], training=False)**

**y\_batch\_valid = tf.reshape(y\_batch\_valid, (y\_batch\_valid.shape[0:2]))**

**# Saves gt and predicted values to be able to visualize the results**

**if step == STEP\_TO\_VISUALIZE and epoch % EPOCH\_TO\_VISUALIZE == 0 and epoch != 0:**

**y\_gt\_visualize\_valid = y\_batch\_valid**

**logits\_visualize\_valid = logits**

**center\_x\_valid\_visualize = center\_x\_valid**

**center\_y\_valid\_visualize = center\_y\_valid**

**# Compute the loss value for this minibatch.**

**loss\_value = loss\_fn(y\_batch\_valid, logits)**

**valid\_batch\_losses.append(loss\_value)**

**print("The mean validation loss after %d epochs: %.4f"% (epoch, float(sum(valid\_batch\_losses)/len(valid\_batch\_losses))))**

**valid\_epoch\_losses.append(float(sum(valid\_batch\_losses)/len(valid\_batch\_losses)))**

**return logits\_visualize\_valid, center\_x\_valid\_visualize, center\_y\_valid\_visualize, y\_gt\_visualize\_valid**

**This function calculates the validation loss for the model on the validation dataset.** The validation dataset is passed as the argument valid\_dataset.

In each iteration of the loop, a batch is retrieved from the validation dataset and is processed through the model. The output logits are then compared with the ground truth y\_batch\_valid to calculate the loss value using the loss\_fn function.

After all batches have been processed, the mean validation loss for the epoch is computed by taking the average of all the batch losses. The mean validation loss is then printed to the console.

Additionally, if the STEP\_TO\_VISUALIZE-th batch is reached and the current epoch is a multiple of EPOCH\_TO\_VISUALIZE, the ground truth values, predicted logits, and centers are saved to be able to visualize the results later. These saved values are returned by the function.

**model = GoalPredModel(11, LATENT\_DIM, 8, 2)**

**lr = START\_LR**

**#optimizer = Adam(learning\_rate=lr, beta\_1=0.9, beta\_2=0.999)**

**optimizer = tf.keras.optimizers.Adam(learning\_rate=lr, beta\_1=0.9, beta\_2=0.999)**

**loss\_fn = tf.keras.losses.MeanSquaredError()**

**train\_epoch\_losses = []**

**valid\_epoch\_losses = []**

Here, model is instantiated using the GoalPredModel class, which takes in the following arguments:

* 11: the number of features in the time series data
* LATENT\_DIM: the dimension of the latent state vectors
* 8: the number of objects in the scenario
* 2: the number of timesteps to predict into the future

lr is set to the **starting learning rate**, and the optimizer is set to an instance of the Adam optimizer with the given learning rate and hyperparameters for the exponential moving average of the gradient and the squared gradient.

loss\_fn is set to an instance of the mean squared error loss function provided by TensorFlow.

train\_epoch\_losses and valid\_epoch\_losses are empty lists that will be used to store the training and validation losses over the epochs of training.

**dataset\_for\_visualization = tf.data.TFRecordDataset(FILENAME, compression\_type='') # parse the tf record**

**dataset\_for\_visualization\_valid = tf.data.TFRecordDataset(VALIDATION\_FILENAME, compression\_type='') # parse the tf record**

**for step, batch in enumerate(dataset\_for\_visualization.batch(BS)):**

**if step == STEP\_TO\_VISUALIZE:**

**for example in batch:**

**visualization\_file = tf.io.parse\_single\_example(example, features\_description)**

**break**

**for step, batch in enumerate(dataset\_for\_visualization\_valid.batch(BSV)):**

**if step == STEP\_TO\_VISUALIZE:**

**for example in batch:**

**visualization\_file\_valid = tf.io.parse\_single\_example(example, features\_description)**

**break**

**visualization\_file\_for\_prediction = dataset\_for\_visualization.map(\_parse\_model\_ready)**

**visualization\_file\_for\_prediction\_valid = dataset\_for\_visualization\_valid.map(\_parse\_model\_ready)**

It seems like the code is setting up a data pipeline for visualization purposes.

* dataset\_for\_visualization and dataset\_for\_visualization\_valid are TensorFlow TFRecordDataset objects that represent the data stored in TFRecord format. They are initialized with the file paths to the training and validation datasets respectively.
* visualization\_file and visualization\_file\_valid are dictionaries that contain a single example from the training and validation datasets respectively. These examples are obtained from the first batch of the datasets using dataset\_for\_visualization.batch(BS) and dataset\_for\_visualization\_valid.batch(BSV) respectively.
* visualization\_file\_for\_prediction and visualization\_file\_for\_prediction\_valid are TensorFlow datasets that are created by mapping the function \_parse\_model\_ready to dataset\_for\_visualization and dataset\_for\_visualization\_valid respectively. \_parse\_model\_ready is a function that preprocesses the raw data from the TFRecord file to the format expected by the model.

**for epoch in range(EPOCHS):**

**print('\nStart of epoch %d' % (epoch,))**

**# Run 1 epoch on the training dataset and update model parameters**

**logits\_visualize, center\_x\_visualize, center\_y\_visualize, y\_gt\_visualize = train\_1\_epoch(epoch, train\_dataset)**

**# Calculate and print the validation loss**

**logits\_visualize\_valid, center\_x\_valid\_visualize, center\_y\_valid\_visualize, y\_gt\_visualize\_valid = calc\_val\_loss(valid\_dataset)**

**# Visualize the results**

**if epoch % EPOCH\_TO\_VISUALIZE == 0 and epoch != 0:**

**calculate\_plots(visualization\_file, logits\_visualize[0][0:8]+center\_x\_visualize[0], logits\_visualize[0][8:]+center\_y\_visualize[0], y\_gt\_visualize[0][0:8]+center\_x\_visualize[0], y\_gt\_visualize[0][8:]+center\_y\_visualize[0]) #adding the center values to the logits to make the visualization in the original coordinate system**

**calculate\_plots(visualization\_file\_valid, logits\_visualize\_valid[0][0:8]+center\_x\_valid\_visualize[0], logits\_visualize\_valid[0][8:]+center\_y\_valid\_visualize[0], y\_gt\_visualize\_valid[0][0:8]+center\_x\_valid\_visualize[0], y\_gt\_visualize\_valid[0][8:]+center\_y\_valid\_visualize[0]) #adding the center values to the logits to make the visualization in the original coordinate system**

This code is training a neural network model on the Waymo motion prediction dataset. It uses the train\_1\_epoch function to run one epoch of training on the train\_dataset, and then uses the calc\_val\_loss function to calculate the validation loss on the valid\_dataset. The validation loss is printed and the results are visualized every EPOCH\_TO\_VISUALIZE epochs using the calculate\_plots function.

During each epoch of training, the model parameters are updated based on the training data. The logits\_visualize, center\_x\_visualize, center\_y\_visualize, and y\_gt\_visualize variables are used to visualize the results of the training data, while the logits\_visualize\_valid, center\_x\_valid\_visualize, center\_y\_valid\_visualize, and y\_gt\_visualize\_valid variables are used to visualize the results of the validation data.

The train\_1\_epoch function and calc\_val\_loss function likely contain the bulk of the training and validation code, respectively, and the calculate\_plots function is used to visualize the results. The get\_dataset function is used to get the train, validation, and test datasets from the corresponding file paths.

## **Calculate the GT for future steps**

**def get\_trajectory\_gt(inputs, normalized\_inputs):**

**"""Returns a concatenated tensor of shape (bs,1280,1), where 1280=8\*2\*80 (for x and y coordinate)**

**The only values that are non-zero in the return tensor are the ones that we actually are interested in.**

**"""**

**Yx = normalized\_inputs['future\_states'][:,:,0,:] # is of shape (bs, #agents, time steps in future (80)) -- x coordinate**

**Yy = normalized\_inputs['future\_states'][:,:,1,:] # is of shape (bs, #agents, time steps in future (80)) -- y coordinate**

**tracks = inputs['tracks\_to\_predict']**

**Yx = tf.math.multiply(Yx, tf.expand\_dims(tf.cast(tracks, dtype = 'float32'), axis = 2))**

**Yx = Yx[:,0:8,:]**

**Yy = tf.math.multiply(Yy, tf.expand\_dims(tf.cast(tracks, dtype = 'float32'), axis = 2))**

**Yy = Yx[:,0:8,:]**

**Y = tf.concat([Yx, Yy], axis = 2)**

**Y\_final = tf.expand\_dims(Y[:,0,:], axis=0)**

**for i in range(7):**

**Y\_final = tf.concat([Y\_final, tf.expand\_dims(Y[:,i,:], axis=0)], axis = 0)**

**return Y\_final**

The function get\_trajectory\_gt takes in two inputs, inputs and normalized\_inputs, and returns a concatenated tensor of shape (bs, 1280, 1) where **1280 = 8 \* 2 \* 80 (for the x and y coordinates of 8 agents over 80 timesteps in the future).**

The function extracts the x and y coordinates of the future states from the normalized\_inputs tensor, and then uses the tracks\_to\_predict field of the inputs tensor to filter out the coordinates of agents that are not being predicted.

The filtered coordinates are then concatenated into a single tensor, and returned as Y\_final, where the first dimension represents the batch size.

## **LSTM model for the trajectory prediction part**

**def encoder\_trajPred(num\_states\_steps, latent\_dim):**

**#The length of the large input vector. Depends on how many features we include.**

**input\_vec\_length = 1280**

**num\_static\_features = 20000\*4**

**time\_inputs = Input(shape=(input\_vec\_length, num\_states\_steps))**

**final\_points = Input(shape = (16,1))**

**static\_inputs = Input(shape=(num\_static\_features,1))**

**a0 = Input(shape=(latent\_dim,))**

**c0 = Input(shape=(latent\_dim,))**

**a = a0**

**c = c0**

**# Create the encoder**

**for t in range(num\_states\_steps):**

**# Select the "t"th time step vector from enc\_input.**

**x\_time = Lambda(lambda z: z[:, :, t])(time\_inputs)**

**# Reshape x**

**x\_time = Reshape((1, input\_vec\_length))(x\_time)**

**reshaped\_fp = Reshape((1,16))(final\_points)**

**x\_time = tf.concat([reshaped\_fp, x\_time], axis = 2)**

**x\_time = BatchNormalization()(x\_time)**

**# LSTM\_cell**

**a, \_, c = LSTM(latent\_dim, kernel\_initializer=GlorotUniform(), return\_state = True)(x\_time, initial\_state=[a, c])**

**#Convolve spatial features**

**x\_spatial = Conv1D(4, 5, strides = 2, padding = 'same', kernel\_initializer=GlorotUniform())(static\_inputs)**

**x\_spatial = MaxPool1D(pool\_size=2, strides=2)(x\_spatial)**

**x\_spatial = Conv1D(8, 5, strides = 2, padding = 'same', kernel\_initializer=GlorotUniform())(x\_spatial)**

**x\_spatial = MaxPool1D(pool\_size=2, strides=2)(x\_spatial)**

**x\_spatial = Conv1D(8, 5, strides = 2, padding = 'same', kernel\_initializer=GlorotUniform())(x\_spatial)**

**x\_spatial = MaxPool1D(pool\_size=2, strides=2)(x\_spatial)**

**x\_spatial = Conv1D(16, 5, strides = 2, padding = 'same', kernel\_initializer=GlorotUniform())(x\_spatial)**

**x\_spatial = MaxPool1D(pool\_size=2, strides=2)(x\_spatial)**

**x\_spatial = tf.keras.layers.Flatten()(x\_spatial)**

**x\_spatial = Dropout(rate=DROPOUT\_RATE\_SPATIAL)(x\_spatial)**

**a = tf.keras.layers.Flatten()(a)**

**x = tf.concat([a, x\_spatial], axis = 1)**

**output\_encoder = Dense(LATENT\_DIM, kernel\_initializer=GlorotUniform())(x)**

**return time\_inputs, static\_inputs, final\_points, a0, c0, output\_encoder**

This function defines the encoder architecture for the trajectory prediction model. It takes two inputs: num\_states\_steps which is the number of steps in the future to predict, and latent\_dim which is the dimension of the latent space. It returns six inputs and one output.

The function first defines the input shapes for the three types of inputs: the time inputs (time\_inputs), the static features inputs (static\_inputs), and the final points (final\_points). It also defines the initial states for the LSTM cell (a0 and c0).

It then creates the encoder by iterating over the num\_states\_steps and processing each time step vector using an LSTM cell. The function also applies batch normalization to the input before processing it through the LSTM cell. After processing all the time steps, it applies 1D convolution on the spatial features of static\_inputs and flattens the output. The final latent vector is obtained by concatenating the flattened LSTM output and the flattened convolution output and passing it through a dense layer with latent\_dim units.

The function returns the inputs (time\_inputs, static\_inputs, final\_points, a0, and c0) and the output (output\_encoder) of the encoder.

**def decoder\_trajPred(output\_encoder, outputs, num\_future\_steps, latent\_dim):**

**a0\_dec = output\_encoder**

**c0\_dec = Input(shape=(latent\_dim,))**

**a = a0\_dec**

**c = c0\_dec**

**zero\_input = Input(shape=(128, 2, 2))**

**for t in range(int(num\_future\_steps/10)):**

**zi\_t = Lambda(lambda z: z[:, t, :, :])(zero\_input)**

**zi\_t = Reshape((1, 2\*2))(zi\_t)**

**a, \_, c = LSTM(latent\_dim, kernel\_initializer=GlorotUniform(), return\_state = True)(zi\_t, initial\_state=[a, c])**

**out\_t = Dense(160, kernel\_initializer=GlorotUniform())(a)**

**outputs.append(out\_t)**

**return c0\_dec, zero\_input**

**→doubt about the “outputs”**

**def TrajPredModel(num\_states\_steps, latent\_dim, num\_future\_steps):**

**time\_inputs, static\_inputs, final\_points, a0, c0, output\_encoder = encoder\_trajPred(num\_states\_steps, latent\_dim)**

**outputs = []**

**c0\_dec, zero\_input = decoder\_trajPred(output\_encoder, outputs, num\_future\_steps, latent\_dim)**

**model = Model(inputs=[time\_inputs, static\_inputs, final\_points, a0, c0, c0\_dec, zero\_input], outputs=outputs)**

**return model**

This function defines a Trajectory Prediction Model using an encoder-decoder architecture with an LSTM-based encoder and a feed-forward decoder. It takes three parameters:

* num\_states\_steps: an integer indicating the number of time steps in the input sequence.
* latent\_dim: an integer indicating the dimension of the LSTM hidden state.
* num\_future\_steps: an integer indicating the number of future steps to predict.

The function first calls the encoder\_trajPred() function to create the encoder network, which takes as input time series data (time\_inputs), static data (static\_inputs), and the final points of the input sequence (final\_points). It returns the encoded representation (output\_encoder) of the input sequence.

Then, the function calls the decoder\_trajPred() function to create the decoder network, which takes as input the encoded representation (output\_encoder), the number of future steps to predict (num\_future\_steps), and the LSTM hidden state (latent\_dim). It returns the LSTM hidden state (c0\_dec) and an input tensor (zero\_input) for the decoder network.

Finally, the function creates the Trajectory Prediction Model by defining the input and output tensors for the model, and passing them to the Model constructor. The input tensors include the time series data (time\_inputs), the static data (static\_inputs), the final points of the input sequence (final\_points), the initial LSTM hidden state (a0 and c0), the LSTM hidden state for the decoder (c0\_dec), and the input tensor for the decoder (zero\_input). The output tensors are the predicted future points (outputs).

**def train\_1\_epoch\_trajPred(epoch, train\_dataset):**

**logits\_visualize, center\_x\_visualize, center\_y\_visualize, y\_gt\_visualize = 0, 0, 0, 0**

**# Set the batch losses to be an empty list to only contain losses from this epochs batches.**

**train\_batch\_losses = []**

**# Iterate over the batches of the dataset.**

**for step, batch in enumerate(train\_dataset):**

**# Send in 1 batch to the function train\_step batch.**

**# 'batch' is a DICTIONARY containing the past position, velocity, current position,**

**# velocity etc. etc for all the objects in the scenario for all the time indices for**

**# all batches.**

**train\_batch\_time, train\_batch\_static, y\_batch\_train, center\_x, center\_y = const\_train\_set(batch, get\_trajectory\_gt)**

**bs = train\_batch\_time.shape[0]**

**a0 = np.zeros((bs, LATENT\_DIM))**

**c0 = np.zeros((bs, LATENT\_DIM))**

**c0\_dec = np.zeros((bs, LATENT\_DIM))**

**zero\_input = np.zeros((bs, 128, 2, 2))**

**with tf.GradientTape() as tape:**

**# Run the forward pass of the layer.**

**final\_points = get\_gt\_final(batch, normalize(batch))**

**logits = traj\_model([train\_batch\_time, train\_batch\_static, final\_points, a0, c0, c0\_dec, zero\_input], training=True)**

**# Compute the loss value for this minibatch.**

**loss\_value = loss\_fn(y\_batch\_train, logits)**

**# Use the gradient tape to automatically retrieve**

**# the gradients of the trainable variables with respect to the loss.**

**grads = tape.gradient(loss\_value, traj\_model.trainable\_weights)**

**# Run one step of gradient descent by updating**

**# the value of the variables to minimize the loss**

**optimizer.apply\_gradients(zip(grads, traj\_model.trainable\_weights))**

**train\_batch\_losses.append(loss\_value)**

**# Prints the training loss after each epoch**

**if len(train\_batch\_losses) != 0:**

**print("The mean training loss after %d epochs: %.4f"% (epoch, float(sum(train\_batch\_losses)/len(train\_batch\_losses))))**

**traj\_train\_epoch\_losses.append(float(sum(train\_batch\_losses)/len(train\_batch\_losses)))**

This code appears to train a trajectory prediction model for object motion.

The train\_1\_epoch\_trajPred function trains the model for one epoch using the provided train\_dataset, which contains past and current object states.

During each iteration of the loop, the function extracts the necessary inputs from the batch and initializes the hidden states of the LSTM layers. Then, it uses tf.GradientTape to compute the gradients of the loss with respect to the model weights, and applies them using the optimizer.

Finally, the mean training loss for the epoch is computed and printed.

**def calculate\_validation\_loss(valid\_dataset):**

**logits\_visualize\_valid, center\_x\_valid\_visualize, center\_y\_valid\_visualize, y\_gt\_visualize\_valid = 0,0,0,0**

**valid\_batch\_losses = []**

**# Calculates and prints the validation loss**

**for step, batch in enumerate(valid\_dataset):**

**valid\_time\_batch, valid\_static\_batch, y\_batch\_valid, center\_x\_valid, center\_y\_valid = const\_train\_set(batch, get\_trajectory\_gt)**

**bs = valid\_time\_batch.shape[0]**

**a0 = np.zeros((bs, LATENT\_DIM))**

**c0 = np.zeros((bs, LATENT\_DIM))**

**c0\_dec = np.zeros((bs, LATENT\_DIM))**

**zero\_input = np.zeros((bs, 128, 2, 2))**

**with tf.GradientTape() as tape:**

**# Run the forward pass of the layer.**

**final\_points = get\_gt\_final(batch, normalize(batch))**

**logits = traj\_model([valid\_time\_batch, valid\_static\_batch, final\_points, a0, c0, c0\_dec, zero\_input], training=False)**

**# Compute the loss value for this minibatch.**

**loss\_value = loss\_fn(y\_batch\_valid, logits)**

**valid\_batch\_losses.append(loss\_value)**

**if len(valid\_batch\_losses) != 0:**

**print("The mean validation loss after %d epochs: %.4f"% (epoch, float(sum(valid\_batch\_losses)/len(valid\_batch\_losses))))**

**traj\_valid\_epoch\_losses.append(float(sum(valid\_batch\_losses)/len(valid\_batch\_losses)))**

→From ChatGPT **def calculate\_validation\_loss(epoch, valid\_dataset):** this line changed and we need to check again.

→The above finding the validation loss.

**epochs = EPOCHS**

**traj\_train\_epoch\_losses = []**

**traj\_valid\_epoch\_losses = []**

**traj\_model = TrajPredModel(11, LATENT\_DIM, 80)**

This code initializes some variables and the TrajPredModel with some arguments. The TrajPredModel is a class that defines the architecture of the model used for trajectory prediction. The first argument is 11, which corresponds to the number of features used for each time step in the input data. The second argument is LATENT\_DIM, which is a hyperparameter that specifies the dimensionality of the hidden state of the LSTM layer. The third argument is 80, which is the number of timesteps used for trajectory prediction.

The traj\_train\_epoch\_losses and traj\_valid\_epoch\_losses lists are used to store the training and validation loss values for each epoch, respectively. The epochs variable specifies the number of epochs for training.

**lr = START\_LR\_TRAJ**

**#optimizer = Adam(learning\_rate=lr, beta\_1=0.9, beta\_2=0.999)**

**optimizer = tf.keras.optimizers.Adam(learning\_rate=lr, beta\_1=0.9, beta\_2=0.999)**

**loss\_fn = tf.keras.losses.MeanSquaredError()**

These lines define the optimizer and loss function to be used during training.

The optimizer is an instance of the tf.keras.optimizers.Adam class, which is an adaptive optimization algorithm based on the stochastic gradient descent algorithm. It adjusts the learning rate adaptively for each weight parameter based on the estimate of the first and second moments of the gradients. The learning\_rate parameter specifies the step size at each iteration.

The loss function is an instance of the tf.keras.losses.MeanSquaredError class, which computes the mean squared error between the predicted and true values. This is a commonly used loss function for regression problems.

**for epoch in range(epochs):**

**print('\nStart of epoch %d' % (epoch,))**

**train\_1\_epoch\_trajPred(epoch, train\_dataset)**

**calculate\_validation\_loss(valid\_dataset)**

**→we need to change from the chatGPT →**

**calculate\_validation\_loss(epoch, valid\_dataset)**

**def testing(test\_dataset):**

**test\_batch\_ade = []**

**test\_batch\_fde = []**

**for step, batch in enumerate(test\_dataset):**

**test\_time\_batch, test\_static\_batch, y\_batch\_test, center\_x\_test, center\_y\_test = const\_train\_set(batch, get\_trajectory\_gt)**

**\_, \_, y\_batch\_test\_end\_point, \_, \_ = const\_train\_set(batch, get\_gt\_final)**

**bs = test\_time\_batch.shape[0]**

**a0 = np.zeros((bs, LATENT\_DIM))**

**c0 = np.zeros((bs, LATENT\_DIM))**

**c0\_dec = np.zeros((bs, LATENT\_DIM))**

**zero\_input = np.zeros((bs, 128, 2, 2))**

**with tf.GradientTape() as tape:**

**predicted\_final\_points = model([test\_time\_batch, test\_static\_batch, a0, c0], training=False)**

**predicted\_trajectories = traj\_model([test\_time\_batch, test\_static\_batch, predicted\_final\_points, a0, c0, c0\_dec, zero\_input], training=False)**

**# Compute the loss value for this minibatch.**

**ade = np.sqrt(loss\_fn(y\_batch\_test, predicted\_trajectories))**

**test\_batch\_ade.append(ade)**

**fde = np.sqrt(loss\_fn(y\_batch\_test\_end\_point, predicted\_final\_points))**

**test\_batch\_fde.append(fde)**

**return test\_batch\_ade, test\_batch\_fde**

This function seems to be calculating the average displacement error (ADE) and final displacement error (FDE) for the test dataset. The inputs to the function are the test dataset. The function then iterates over each batch in the test dataset and computes the predictions for the batch using the trained models.

The function then calculates the ADE and FDE for the batch by comparing the ground truth trajectories and the predicted trajectories. **The ADE is computed as the Euclidean distance between the predicted and ground truth trajectories averaged over all timesteps in the trajectory. The FDE is computed as the Euclidean distance between the predicted and ground truth final points.**

Finally, the function returns the ADE and FDE for all batches in the test dataset as lists.

#test\_batch\_ade, test\_batch\_fde = testing(valid\_dataset)

#print('Mean ade:', sum(test\_batch\_ade)/(len(test\_batch\_ade)))

#print('Mean fde:', sum(test\_batch\_fde)/(len(test\_batch\_fde)))

**test\_batch\_ade, test\_batch\_fde = testing(test\_dataset)**

**#print(sum(test\_batch\_ade))**

**if len(test\_batch\_ade) > 0:**

**print('Mean ade:', sum(test\_batch\_ade)/(len(test\_batch\_ade)))**

**if len(test\_batch\_fde) > 0:**

**print('Mean fde:', sum(test\_batch\_fde)/(len(test\_batch\_fde)))**

It looks like you are testing your trained models on the test dataset. The ADE and FDE metrics have been computed, and their mean values have been printed.

-------------------------------------------------------------------------------------

**def create\_animation(images):**

**""" Creates a Matplotlib animation of the given images.**

**Args:**

**images: A list of numpy arrays representing the images.**

**Returns:**

**A matplotlib.animation.Animation.**

**Usage:**

**anim = create\_animation(images)**

**anim.save('/tmp/animation.avi')**

**HTML(anim.to\_html5\_video())**

**"""**

**plt.ioff()**

**fig, ax = plt.subplots()**

**dpi = 100**

**size\_inches = 1000 / dpi**

**fig.set\_size\_inches([size\_inches, size\_inches])**

**plt.ion()**

**def animate\_func(i):**

**ax.imshow(images[i])**

**ax.set\_xticks([])**

**ax.set\_yticks([])**

**ax.grid('off')**

**anim = animation.FuncAnimation(**

**fig, animate\_func, frames=len(images) // 2, interval=100)**

**plt.close(fig)**

**return anim**

**anim = create\_animation(images[::5])**

**HTML(anim.to\_html5\_video())**

The create\_animation function takes a list of numpy arrays representing images and creates a Matplotlib animation of those images. The animation shows the images in a sequence with a given interval between them. The function returns a matplotlib.animation.Animation object that can be saved as a video file or displayed as an HTML5 video.

The animation is created using the matplotlib.animation.FuncAnimation class, which takes a figure object, a function to be called for each frame, and the number of frames to display. In this case, the animate\_func function is called for each frame and sets the current image as the plot image. The interval parameter specifies the delay between frames in milliseconds.

The resulting animation can be displayed as an HTML5 video using the to\_html5\_video method of the Animation object and the HTML function from IPython.display.

**def \_parse(value):**

**decoded\_example = tf.io.parse\_single\_example(value, features\_description)**

**past\_states = tf.stack([**

**decoded\_example['state/past/x'], decoded\_example['state/past/y'],**

**decoded\_example['state/past/length'], decoded\_example['state/past/width'],**

**decoded\_example['state/past/bbox\_yaw'],**

**decoded\_example['state/past/velocity\_x'],**

**decoded\_example['state/past/velocity\_y']**

**], -1)**

**cur\_states = tf.stack([**

**decoded\_example['state/current/x'], decoded\_example['state/current/y'],**

**decoded\_example['state/current/length'],**

**decoded\_example['state/current/width'],**

**decoded\_example['state/current/bbox\_yaw'],**

**decoded\_example['state/current/velocity\_x'],**

**decoded\_example['state/current/velocity\_y']**

**], -1)**

**input\_states = tf.concat([past\_states, cur\_states], 1)[..., :2]**

**future\_states = tf.stack([**

**decoded\_example['state/future/x'], decoded\_example['state/future/y'],**

**decoded\_example['state/future/length'],**

**decoded\_example['state/future/width'],**

**decoded\_example['state/future/bbox\_yaw'],**

**decoded\_example['state/future/velocity\_x'],**

**decoded\_example['state/future/velocity\_y']**

**], -1)**

**gt\_future\_states = tf.concat([past\_states, cur\_states, future\_states], 1)**

**past\_is\_valid = decoded\_example['state/past/valid'] > 0**

**current\_is\_valid = decoded\_example['state/current/valid'] > 0**

**future\_is\_valid = decoded\_example['state/future/valid'] > 0**

**gt\_future\_is\_valid = tf.concat(**

**[past\_is\_valid, current\_is\_valid, future\_is\_valid], 1)**

**# If a sample was not seen at all in the past, we declare the sample as**

**# invalid.**

**sample\_is\_valid = tf.reduce\_any(**

**tf.concat([past\_is\_valid, current\_is\_valid], 1), 1)**

**inputs = {**

**'input\_states': input\_states,**

**'gt\_future\_states': gt\_future\_states,**

**'gt\_future\_is\_valid': gt\_future\_is\_valid,**

**'object\_type': decoded\_example['state/type'],**

**'tracks\_to\_predict': decoded\_example['state/tracks\_to\_predict'] > 0,**

**'sample\_is\_valid': sample\_is\_valid,**

**}**

**return inputs**

This is a private function \_parse that is used for parsing a single example from a TensorFlow dataset. The input argument value is a single example from the dataset, encoded as a string, and the output is a dictionary of tensors that represent various fields of the example.

The function first calls tf.io.parse\_single\_example to decode the example according to a specified protocol buffer format called features\_description. The format defines the fields and data types of the example.

The function then extracts the past, current, and future states of the object, along with various other fields like object type, tracks to predict, and sample validity, from the decoded example. The states are represented as tensors of shape (batch\_size, num\_steps, 7) where batch\_size is the number of examples in the batch and num\_steps is the number of time steps in the sequence (past, current, and future). The seven dimensions of the states are the x, y positions, length, width, yaw angle, and x, y velocities.

The function finally returns a dictionary inputs that contains all the extracted fields as tensors.

**def \_default\_metrics\_config():**

**config = motion\_metrics\_pb2.MotionMetricsConfig()**

**config\_text = """**

**track\_steps\_per\_second: 10**

**prediction\_steps\_per\_second: 2**

**track\_history\_samples: 10**

**track\_future\_samples: 80**

**speed\_lower\_bound: 1.4**

**speed\_upper\_bound: 11.0**

**speed\_scale\_lower: 0.5**

**speed\_scale\_upper: 1.0**

**step\_configurations {**

**measurement\_step: 5**

**lateral\_miss\_threshold: 1.0**

**longitudinal\_miss\_threshold: 2.0**

**}**

**step\_configurations {**

**measurement\_step: 9**

**lateral\_miss\_threshold: 1.8**

**longitudinal\_miss\_threshold: 3.6**

**}**

**step\_configurations {**

**measurement\_step: 15**

**lateral\_miss\_threshold: 3.0**

**longitudinal\_miss\_threshold: 6.0**

**}**

**max\_predictions: 6**

**"""**

**text\_format.Parse(config\_text, config)**

**return config**

This function returns a default configuration for the motion metrics used to evaluate the performance of a motion prediction model. The configuration is defined using a Protocol Buffer, a mechanism for serializing structured data in a compact and efficient format.

The default configuration includes various settings, such as the number of samples per second for the track and prediction steps, the number of past and future samples to use for predictions, speed bounds and scaling factors, and threshold values for measuring the accuracy of the predictions. It also specifies the maximum number of predictions to make for each track.

The configuration is defined using a text string, which is parsed into a motion\_metrics\_pb2.MotionMetricsConfig object using the text\_format.Parse function from the google.protobuf.text\_format module.

**class SimpleModel(tf.keras.Model):**

**"""A simple one-layer regressor."""**

**def \_\_init\_\_(self, num\_agents\_per\_scenario, num\_states\_steps,**

**num\_future\_steps):**

**super(SimpleModel, self).\_\_init\_\_()**

**self.\_num\_agents\_per\_scenario = num\_agents\_per\_scenario**

**self.\_num\_states\_steps = num\_states\_steps**

**self.\_num\_future\_steps = num\_future\_steps**

**self.regressor = tf.keras.layers.Dense(num\_future\_steps \* 2)**

**def call(self, states):**

**states = tf.reshape(states, (-1, self.\_num\_states\_steps \* 2))**

**pred = self.regressor(states)**

**pred = tf.reshape(**

**pred, [-1, self.\_num\_agents\_per\_scenario, self.\_num\_future\_steps, 2])**

**return pred**

This is a simple implementation of a regression model using a single dense layer. The model takes in states as input, which is a tensor of shape (batch\_size, num\_agents\_per\_scenario, num\_states\_steps, 2).

num\_agents\_per\_scenario refers to the number of agents (e.g. vehicles) in each scenario, num\_states\_steps refers to the number of steps in the past trajectory of each agent that are used as input to the model, and num\_future\_steps refers to the number of future steps to be predicted for each agent.

The call() method first reshapes the input tensor to be of shape (batch\_size \* num\_agents\_per\_scenario, num\_states\_steps \* 2). It then applies the regressor layer, which is a Dense layer with num\_future\_steps \* 2 output units, and reshapes the output back to the original shape of (batch\_size, num\_agents\_per\_scenario, num\_future\_steps, 2), where the last dimension represents the x and y coordinates of each predicted step.

In summary, the model takes in past trajectory information for each agent and outputs predicted future trajectories.

**class MotionMetrics(tf.keras.metrics.Metric):**

**"""Wrapper for motion metrics computation."""**

**def \_\_init\_\_(self, config):**

**super().\_\_init\_\_()**

**self.\_prediction\_trajectory = []**

**self.\_prediction\_score = []**

**self.\_ground\_truth\_trajectory = []**

**self.\_ground\_truth\_is\_valid = []**

**self.\_prediction\_ground\_truth\_indices = []**

**self.\_prediction\_ground\_truth\_indices\_mask = []**

**self.\_object\_type = []**

**self.\_metrics\_config = config**

**def reset\_state():**

**self.\_prediction\_trajectory = []**

**self.\_prediction\_score = []**

**self.\_ground\_truth\_trajectory = []**

**self.\_ground\_truth\_is\_valid = []**

**self.\_prediction\_ground\_truth\_indices = []**

**self.\_prediction\_ground\_truth\_indices\_mask = []**

**self.\_object\_type = []**

**def update\_state(self, prediction\_trajectory, prediction\_score,**

**ground\_truth\_trajectory, ground\_truth\_is\_valid,**

**prediction\_ground\_truth\_indices,**

**prediction\_ground\_truth\_indices\_mask, object\_type):**

**self.\_prediction\_trajectory.append(prediction\_trajectory)**

**self.\_prediction\_score.append(prediction\_score)**

**self.\_ground\_truth\_trajectory.append(ground\_truth\_trajectory)**

**self.\_ground\_truth\_is\_valid.append(ground\_truth\_is\_valid)**

**self.\_prediction\_ground\_truth\_indices.append(**

**prediction\_ground\_truth\_indices)**

**self.\_prediction\_ground\_truth\_indices\_mask.append(**

**prediction\_ground\_truth\_indices\_mask)**

**self.\_object\_type.append(object\_type)**

**def result(self):**

**# [batch\_size, num\_preds, 1, 1, steps, 2].**

**# The ones indicate top\_k = 1, num\_agents\_per\_joint\_prediction = 1.**

**prediction\_trajectory = tf.concat(self.\_prediction\_trajectory, 0)**

**# [batch\_size, num\_preds, 1].**

**prediction\_score = tf.concat(self.\_prediction\_score, 0)**

**# [batch\_size, num\_agents, gt\_steps, 7].**

**ground\_truth\_trajectory = tf.concat(self.\_ground\_truth\_trajectory, 0)**

**# [batch\_size, num\_agents, gt\_steps].**

**ground\_truth\_is\_valid = tf.concat(self.\_ground\_truth\_is\_valid, 0)**

**# [batch\_size, num\_preds, 1].**

**prediction\_ground\_truth\_indices = tf.concat(**

**self.\_prediction\_ground\_truth\_indices, 0)**

**# [batch\_size, num\_preds, 1].**

**prediction\_ground\_truth\_indices\_mask = tf.concat(**

**self.\_prediction\_ground\_truth\_indices\_mask, 0)**

**# [batch\_size, num\_agents].**

**object\_type = tf.cast(tf.concat(self.\_object\_type, 0), tf.int64)**

**# We are predicting more steps than needed by the eval code. Subsample.**

**interval = (**

**self.\_metrics\_config.track\_steps\_per\_second //**

**self.\_metrics\_config.prediction\_steps\_per\_second)**

**prediction\_trajectory = prediction\_trajectory[...,**

**(interval - 1)::interval, :]**

**return py\_metrics\_ops.motion\_metrics(**

**config=self.\_metrics\_config.SerializeToString(),**

**prediction\_trajectory=prediction\_trajectory,**

**prediction\_score=prediction\_score,**

**ground\_truth\_trajectory=ground\_truth\_trajectory,**

**ground\_truth\_is\_valid=ground\_truth\_is\_valid,**

**prediction\_ground\_truth\_indices=prediction\_ground\_truth\_indices,**

**prediction\_ground\_truth\_indices\_mask=prediction\_ground\_truth\_indices\_mask,**

**object\_type=object\_type)**

This is a class definition for a custom TensorFlow Keras metric named MotionMetrics. It is a wrapper around motion metrics computation.

The class has several instance variables, including lists for storing the prediction trajectory, prediction score, ground truth trajectory, ground truth is valid, prediction ground truth indices, prediction ground truth indices mask, and object type. These variables are updated using the update\_state method and reset using the reset\_state method.

The result method computes the motion metrics using the stored variables and returns the results. The motion metrics computation is performed using a TensorFlow custom op called py\_metrics\_ops.motion\_metrics, which takes several inputs including the motion metrics configuration, the prediction trajectory, the prediction score, the ground truth trajectory, the ground truth is valid, the prediction ground truth indices, the prediction ground truth indices mask, and the object type.

Overall, this class provides a convenient way to compute motion metrics during training and evaluation of a motion prediction model

**model = SimpleModel(128, 11, 80)**

**optimizer = tf.keras.optimizers.Adam(learning\_rate=1e-3)**

**loss\_fn = tf.keras.losses.MeanSquaredError()**

**metrics\_config = \_default\_metrics\_config()**

**motion\_metrics = MotionMetrics(metrics\_config)**

**metric\_names = config\_util.get\_breakdown\_names\_from\_motion\_config(**

**metrics\_config)**

It looks like you are defining a model, optimizer, loss function, and metrics for a motion prediction task.

* model = SimpleModel(128, 11, 80) creates an instance of the SimpleModel class with 128 agents per scenario, 11 state steps, and 80 future steps.
* optimizer = tf.keras.optimizers.Adam(learning\_rate=1e-3) creates an Adam optimizer with a learning rate of 0.001.
* loss\_fn = tf.keras.losses.MeanSquaredError() creates a mean squared error loss function.
* metrics\_config = \_default\_metrics\_config() loads the default motion metrics configuration.
* motion\_metrics = MotionMetrics(metrics\_config) creates an instance of the MotionMetrics class with the given metrics configuration.
* metric\_names = config\_util.get\_breakdown\_names\_from\_motion\_config(metrics\_config) extracts the breakdown names from the given metrics configuration. These names are used to compute more specific metrics for each breakdown category.

**def train\_step(inputs):**

**with tf.GradientTape() as tape:**

**# [batch\_size, num\_agents, D]**

**states = inputs['input\_states']**

**# Predict. [batch\_size, num\_agents, steps, 2].**

**pred\_trajectory = model(states, training=True)**

**# Set training target.**

**prediction\_start = metrics\_config.track\_history\_samples + 1**

**# [batch\_size, num\_agents, steps, 7]**

**gt\_trajectory = inputs['gt\_future\_states']**

**gt\_targets = gt\_trajectory[..., prediction\_start:, :2]**

**# [batch\_size, num\_agents, steps]**

**gt\_is\_valid = inputs['gt\_future\_is\_valid']**

**# [batch\_size, num\_agents, steps]**

**weights = (**

**tf.cast(inputs['gt\_future\_is\_valid'][..., prediction\_start:],**

**tf.float32) \***

**tf.cast(inputs['tracks\_to\_predict'][..., tf.newaxis], tf.float32))**

**loss\_value = loss\_fn(gt\_targets, pred\_trajectory, sample\_weight=weights)**

**grads = tape.gradient(loss\_value, model.trainable\_weights)**

**optimizer.apply\_gradients(zip(grads, model.trainable\_weights))**

**# [batch\_size, num\_agents, steps, 2] ->**

**# [batch\_size, num\_agents, 1, 1, steps, 2].**

**# The added dimensions are top\_k = 1, num\_agents\_per\_joint\_prediction = 1.**

**pred\_trajectory = pred\_trajectory[:, :, tf.newaxis, tf.newaxis]**

**# Fake the score since this model does not generate any score per predicted**

**# trajectory.**

**pred\_score = tf.ones(shape=tf.shape(pred\_trajectory)[:3])**

**# [batch\_size, num\_agents].**

**object\_type = inputs['object\_type']**

**# [batch\_size, num\_agents].**

**batch\_size = tf.shape(inputs['tracks\_to\_predict'])[0]**

**num\_samples = tf.shape(inputs['tracks\_to\_predict'])[1]**

**pred\_gt\_indices = tf.range(num\_samples, dtype=tf.int64)**

**# [batch\_size, num\_agents, 1].**

**pred\_gt\_indices = tf.tile(pred\_gt\_indices[tf.newaxis, :, tf.newaxis],**

**(batch\_size, 1, 1))**

**# [batch\_size, num\_agents, 1].**

**pred\_gt\_indices\_mask = inputs['tracks\_to\_predict'][..., tf.newaxis]**

**motion\_metrics.update\_state(pred\_trajectory, pred\_score, gt\_trajectory,**

**gt\_is\_valid, pred\_gt\_indices,**

**pred\_gt\_indices\_mask, object\_type)**

**return loss\_value**

This function defines a single training step for a machine learning model. It takes in inputs as a dictionary, which contains the input states, ground truth future states, ground truth future is valid flags, tracks to predict, and object types. It uses these inputs to make a prediction using the model, and calculates the loss\_value using the loss\_fn and weights. It then calculates the gradients of the loss with respect to the model's trainable weights using a tf.GradientTape, applies the gradients to the optimizer, and updates the motion\_metrics using the predicted trajectory, predicted score, ground truth trajectory, ground truth is valid flags, predicted ground truth indices, predicted ground truth indices mask, and object type. Finally, it returns the loss value.

Note that the motion\_metrics object is likely a custom metrics object used to track various evaluation metrics during training. The specific metrics being tracked may depend on the configuration specified by metrics\_config and metric\_names.

**dataset = tf.data.TFRecordDataset(FILENAME)**

**dataset = dataset.map(\_parse)**

**dataset = dataset.batch(32)**

These lines of code create a TensorFlow dataset from a TFRecord file. The TFRecordDataset class reads records from one or more TFRecord files and creates a dataset. The \_parse function is applied to each record to decode the data. Finally, the batch method is used to combine multiple records into batches of size 32.

Here's a breakdown of what each line does:

1. dataset = tf.data.TFRecordDataset(FILENAME): Create a dataset that reads records from the TFRecord file FILENAME.
2. dataset = dataset.map(\_parse): Apply the \_parse function to each record in the dataset to decode the data.
3. dataset = dataset.batch(32): Combine multiple records into batches of size 32.

**epochs = 10**

**num\_batches\_per\_epoch = 10**

**for epoch in range(epochs):**

**print('\nStart of epoch %d' % (epoch,))**

**start\_time = time.time()**

**# Iterate over the batches of the dataset.**

**for step, batch in enumerate(dataset):**

**loss\_value = train\_step(batch)**

**# Log every 10 batches.**

**if step % 10 == 0:**

**print('Training loss (for one batch) at step %d: %.4f' %**

**(step, float(loss\_value)))**

**print('Seen so far: %d samples' % ((step + 1) \* 64))**

**if step >= num\_batches\_per\_epoch:**

**break**

**# Display metrics at the end of each epoch.**

**train\_metric\_values = motion\_metrics.result()**

**for i, m in enumerate(**

**['min\_ade', 'min\_fde', 'miss\_rate', 'overlap\_rate', 'map']):**

**for j, n in enumerate(metric\_names):**

**print('{}/{}: {}'.format(m, n, train\_metric\_values[i, j]))**

This code is training a model using a TensorFlow dataset. It trains for a specified number of epochs (in this case, 10), with each epoch consisting of a specified number of batches (in this case, 10).

For each batch, it calls the train\_step function, which computes the loss on the batch and applies backpropagation to update the model's parameters. The loss is computed using the mean squared error loss function.

Every 10 batches, the training loss for that batch is printed, along with the total number of samples seen so far.

After each epoch, the current metrics for the model are printed using the motion\_metrics object. This object computes various motion-related metrics (e.g. minimum average displacement error, minimum final displacement error, miss rate, overlap rate, etc.) based on the predicted trajectories and the ground truth trajectories. The metric\_names object is used to specify the names of the different breakdowns for which these metrics should be computed (e.g. "all", "car", "pedestrian", etc.).