Code explanation:

Let’s start by understanding the basic architecture. The idea is to have a neural network that will extract features from the masked image. Instead of passing the raw image with too much information, the idea is to just pass the masked version which provides a higher level representation of the image. So, you can have a raw image of a road with several cars, buildings and other things in it. And then, after applying DeepLab (resnet18) on it, you can get a new image where different parts of the environment are classified pixel-wise. So, for example, all pixels belonging to the road are colored pink. All pixels belonging to a car could be colored blue. And so on. This allows us to get our work cut out for us.

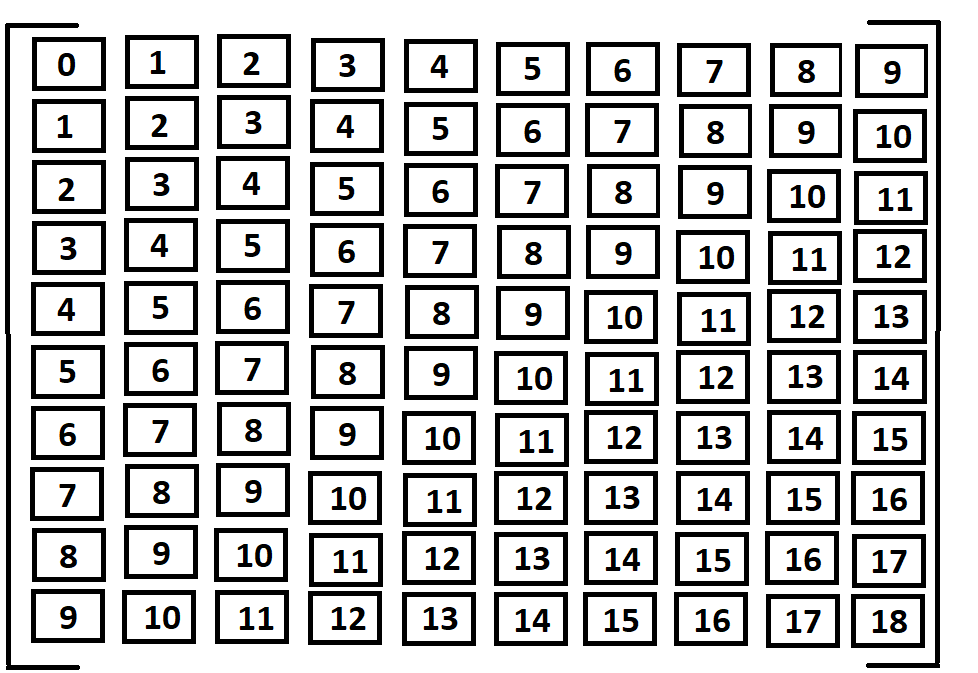
But of course, the raw image still has useful information. So, the masked version of the same image shows dynamic obstacles labeled pixel-wise with the same color. We may still be able to infer something about the orientation and size of the dynamic obstacle but we need more information. In this case, a good idea is to go back to the raw image to extract this information.

So, as of now, we have a neural network to extract features from a masked version of the full raw image. As of now, we are extracting a 20-feature vector. Then, we have another neural network to extract features from the dynamic obstacles themselves. What we do is that we crop out the dynamic obstacle from the original raw image and pass it through a neural network in an effort to extract features from it. As of now, we are extracting a 6-feature vector to which we add 4 other values to get a final 10-feature vector. These 4 other values are the bounding box values for the dynamic obstacle. Our idea is to extract information about the dynamic obstacle from the raw image and then combine it with the bounding box values. Hopefully, this gives a more well-rounded representation of the dynamic obstacle (an image contextual representation of the obstacle and it’s location as well).

The plan right now is to use an RNN-Encoder-Decoder architecture where the encoder consists of bi-directional GRU cells and there is also an attention mechanism. The input to the encoder is a tensor of dimensions (seq\_len, batch\_size, 34). The number 34 is obtained because of the 20 features extracted from the masked image, 6 features obtained from the cropped dynamic obstacle, the 4 bounding box values for the dynamic obstacle and the 4 bounding box values of the agent whose trajectory we wish to predict. However, since an agent’s trajectory depends on not just one dynamic obstacle but potentially many more, we decided to use up to 5 dynamic obstacles. So, 20 features from the masked image plus the 6 features and bounding box values of the cropped image for each of the 5 dynamic obstacles (So, 5 x (6 + 4)) and the 4 bounding box values of the agent under focus, means the final dimensions of the tensor input to the encoder is (seq\_len, batch\_size, 74).

So, how is this tensor formed? Let’s start with actual values for sequence length and batch size. So, as in our code, the sequence length is 10 and the batch size is 10. We take 19 masked images and pass these through the neural network to get a tensor of dimensions (19, 20). So, I have 19 rows and each row is nothing but a 20-feature vector. Then, for each image, we get the bounding box values of our “current” agent (this is the agent for which we wish to predict the trajectory). So, that means we have another tensor of dimensions (19, 4). Now, from the 19 original raw images (whose masked version we used previously to extract 20-feature vectors), we crop out the dynamic obstacles and pass them through the neural network. Since, we have 19 original raw images, that means we have 19 cropped images for a dynamic obstacle. So, after passing these 19 cropped images (each is resized before sending as input to the neural network) through the network, we get a new tensor of dimensions (19, 6) but we also have the 4 bounding box coordinates of the dynamic obstacle in each of the 19 images. So, the final tensor, after concatenation, will have dimensions (19, 10). However, we have 5 such tensors (because we choose to use up to 5 dynamic obstacles in trying to predict the trajectory of our current agent). So, that means 5 tensors of dimension (19, 10). Now we combine these tensors: 1 x (19, 20), 1 x (19, 4), 5 x (19, 10). And our final tensor has dimensions (19, 74). But wait, the input to our encoder is supposed to be a tensor of (10, 10, 74). So, what do we do now?

Here, we explain how we reshape the (19, 74) tensor to get (10, 10, 74). Let’s start off by understanding the general format of the input that is sent to the encoder. The general format is (seq\_len, batch\_size, num\_of\_features). Let’s think of this three-dimensional tensor as a 2D one:



Don’t pay attention to the digits in the boxes. For now, imagine this is a tensor of dimension (10, 10, 74). The vertical column represents a single sequence of size 10. So here, you have 10 sequences (hence, batch\_size is 10). But what is the 74 then? The 74 is supposed to be the number of features. Each numbered box is basically a feature vector of size 74. So now, to put it all together, you have a tensor made up of 100 boxes. 10 boxes lined up horizontally and 10 lined up vertically. And each box represents a 74-feature vector.

But what about the digits? Probably you would have guessed by now, but the number represents the image in a sequence. So, box 0 is a 74-feature representation of image 0 and so on. So, think of it like this, the first sequence consists of image 0 up to image 9. Then, instead of having a new sequence starting from image 10 and going up to image 19, we instead decide to start the next sequence from image 1 (which was in second place in the previous sequence) and go all the way to image 10. This allows us to recycle images from previous sequences. So, think of this like a film and, at any given time, you can only see 10 images sequentially. So, you are shown the first 10 images (image 0 to image 9) and then the sequence is moved ahead by one image and the next sequence you now see consists of 10 images but now the sequence starts from image 1 and goes up to image 10. This is a very important idea to understand and it is very important that you, the reader, understand this as this represents how the data is ultimately arranged and sent as input to the encoder.

From this point onwards, we will assume that the reader understood how the data is arranged into a 3-dimensional tensor before being sent as input to the encoder. Now, we talk about the encoder. Our encoder has as many GRUs as there are images in a sequence. So, taking the above 3D tensor as an example, each sequence consists of 10 images. So, that means 10 GRU cells. But since we are using a bi-directional encoder, then that means we have 20 GRU cells. At the end of the encoder, we use the final hidden state as the first hidden state of our first GRU cell in the decoder.

Now, we deal with the decoder. The decoder receives the first input as the bounding box coordinates of our current agent in the last image of the previous sequence. So, if our sequence consists of 10 images, starting from image 0 up to image 9. Then the input to the first GRU cell is the bounding box coordinates of the current agent from image 9. The output of each GRU cell is also bounding box coordinates. Instead of using the output of each GRU cell as input to the next GRU cell, we use teacher forcing. This means that we use the ground truth as input to the next GRU cell. To make it more understandable, let’s say the decoder wants to predict the bounding box coordinates of the current agent in image 10 to image 19 (given that it got the bounding box coordinates in image 0 till image 9). So, it grabs the bounding box coordinates from image 9 (from the sequence that was fed as input to the encoder) and feeds it as input to the first GRU cell in decoder. Then, the next GRU cell receives the bounding box coordinates of image 10 as input. So, instead of using the output of the first GRU cell, we instead use the target bounding box coordinates (the ground truth) for the first GRU cell as input to the next one.

Line-by-line code explanation:

First, we import all the libraries:

**import** torch  
**from** torch **import** optim  
torch.cuda.current\_device()  
torch.autograd.set\_detect\_anomaly(**True**)  
  
**from** network\_classes **import** \*  
**from** torch.utils.data **import** DataLoader, Subset  
**import** torchvision.transforms **as** transforms  
**import** torchvision.models **as** models  
**import** pickle  
**import** random  
**import** cv2  
**from** PIL **import** Image  
**import** matplotlib.pyplot **as** plt  
**import** numpy **as** np  
**import** math  
**import** json  
  
**from** random **import** shuffle

Then, the hyperparameters:

seq\_len = 10  
batch\_size = 10  
other\_agent\_limit = 5  
forwards\_per\_agent = 20  
masked\_output\_ftrs = 20  
unmasked\_output\_ftrs = 6  
encoder\_n\_layers = 3  
decoder\_n\_layers = 3  
teacher\_forcing\_ratio = 1.0  
encoder\_hidden\_size = 100  
decoder\_hidden\_size = 100  
encoder\_dropout = 0.1  
decoder\_dropout = 0.1  
learning\_rate = 0.001  
decoder\_learning\_ratio = 5.0  
prediction\_horizon = 10  
clip = 50.0  
attn\_model = **'dot'**

Since the neural networks require tensors, we are going to create a transform object of the following type:

transform = transforms.Compose([transforms.Resize((224,224)),  
 transforms.ToTensor(),  
 transforms.Normalize(mean=[0.485, 0.456, 0.406], std=[0.229, 0.224, 0.225])])

So, this is a transformation that resizes the image to have dimensions (224, 224) because this is the size needed for the neural networks we plan on using for feature extraction. After that, we first convert the resized image to a tensor and then we normalize the tensor across its three dimensions. Normalizing is done by subtracting a mean value from the pixel value and then dividing the result by the standard deviation. So, for example, here, for each pixel value in the first channel of the image, we subtract 0.485 and divide the result with 0.229.

Now, set the device you wish to use. Use the GPU if it is available, otherwise just use the cpu:

device = torch.device(**'cuda'**) **if** torch.cuda.is\_available() **else** torch.device(**'cpu'**)

The masked and unmasked images are contained in the following locations:

ROOT\_DIR = **"data/labeled\_images"**TRAIN\_DIR\_MASKED = **"for\_training"**TRAIN\_DIR\_UNMASKED = **"unmasked\_images\_training"**

Grab the json file and the corresponding json object:

json\_file = open(os.path.join(ROOT\_DIR, **"bdd100k\_tracking\_cvpr2019\_val.json"**))  
data = json.load(json\_file)

Create your training data set:

TRAIN\_DATASET = CustomDataSet(ROOT\_DIR, TRAIN\_DIR\_MASKED, transform=transform, json\_data=data)

CustomDataSet is a class written in network\_classes.py

**class** CustomDataSet(Dataset):  
 **def** \_\_init\_\_(self, root\_dir, sub\_dir, transform, json\_data):  
 self.root\_dir = root\_dir  
 self.sub\_dir = sub\_dir  
 self.transform = transform  
 self.sanity\_transform = sanity\_transform  
 self.total\_imgs = []  
  
 **for** frame **in** json\_data:  
 **if** frame[**'labels'**] **is not None**:  
 self.total\_imgs.append(frame[**'name'**])  
  
 **def** \_\_len\_\_(self):  
 **return** len(self.total\_imgs)  
  
 **def** \_\_getitem\_\_(self, idx):  
 img\_loc = os.path.join(self.root\_dir, self.sub\_dir, self.total\_imgs[idx])  
 image = Image.open(img\_loc).convert(**"RGB"**)  
  
 sanity\_image = []  
 tensor\_image = []  
 **if** self.transform:  
 tensor\_image = self.transform(image)  
 sanity\_image = self.sanity\_transform(image)  
  
 **return** tensor\_image, sanity\_image

So, here, we want to get all the images according to our json file. Let’s look at the specific code that does that:

**for** frame **in** json\_data:  
 **if** frame[**'labels'**] **is not None**:  
 self.total\_imgs.append(frame[**'name'**])

the json\_file is a json object (from the code before, it is the data object obtained because of this line: data = json.load(json\_file)). This object is nothing more than a list of dictionaries. Each dictionary corresponds to a specific image in a sequence. This dictionary has several keys. For now, since we only need to grab the name of the frame, we just create a list of image names by grabbing the value for the ‘name’ key. At the end of the above for loop, the class object has an attribute total\_imgs which is a list of names and each name refers to images.

We want a list that is nothing but a long list of numbers. The numbers are sequential and start with 0 and end at (1 – length of the training data set):

indices = list(range(len(TRAIN\_DATASET)))

So, for example, if the length of the training dataset was 5, the list obtained would be [0 1 2 3 4]. Next, we want to use a small portion of this dataset. The actual dataset has a size of more than 11,000 images. For the sake of this code demonstration, we will only use the first 100 images:

start\_index = 0  
end\_index = 100  
TRAIN\_DATASET\_SUBSET = Subset(TRAIN\_DATASET, indices[start\_index:end\_index])  
train\_loader = DataLoader(dataset=TRAIN\_DATASET\_SUBSET, batch\_size=batch\_size+seq\_len-1, shuffle=**False**, num\_workers=4)

But what is the train\_loader. Well, in machine learning, we forward a batch of images and backprop only after an entire batch has gone through. Here, however, it’s a bit different. Provided that you perfectly understood how the data is arranged before being sent to the encoder, then this is pretty much self-explanatory. So, recalling the previous example, we have a tensor of size (19, 74) which is repackaged as a tensor of size (10, 10, 74). Here, the batch size and sequence length both are 10. In general, for an encoder that requires an input with these dimensions (seq\_len, batch\_size, num\_of\_ftrs), the number of images to be extracted must be equal to batch\_size + seq\_len – 1. And this is exactly what we are doing here. We are extracting 19 images which will be sent through a neural network to obtain a tensor of size (19, 20) and this will be combined with other tensors to get the final tensor of dimensions (19, 74). On the other hand, num\_workers refers to the number of threads (multiprocessing) and there must obviously be no shuffle (you don’t want to shuffle a sequence of images!)

Next, we want to obtain the bounding boxes. For this, we have a separate text file which is nothing but, once again, a list of dictionaries. Each dictionary is for a specific frame and the sequence of frames here is the same as in the json file. This means that the first 100 images (as is the case here) grabbed from the json file correspond to the first 100 dictionaries grabbed from this text file. Each dictionary has keys that refer to the agent in the frame and the value of each key is the bounding box. So, just to make sense of everything, if you take the first image grabbed from the json file and take the first dictionary from this text file, then you the bounding boxes in the dictionary can be drawn on the image from the json file. And this correspondence carries all the way through the first 100 images of the json file and the entirety of this text file (since this text file only contains dictionaries for the first 100 images of the json file). Now, with that thing in mnd, first we get a list of all the agents in all the 100 images.

fp = open(os.path.join(ROOT\_DIR, **"bbox\_files"**, **"bboxes"** + str(start\_index) + **"to"** + str(end\_index)) + **".txt"**, **"rb"**)  
list\_of\_dicts = pickle.load(fp)  
all\_agents = []  
**for** dict **in** list\_of\_dicts:  
 **for** agent **in** dict.keys():  
 all\_agents.append(agent)

Next, we go back to our dataloader object:

masked\_image\_array = []  
sanity\_masked\_image\_array = []  
**for** i, (image\_batch, sanity\_image\_batch) **in** enumerate(train\_loader):  
 masked\_image\_array.append(image\_batch)  
 sanity\_masked\_image\_array.append(sanity\_image\_batch)

So, what is going on here? To understand this, let’s go back to our CustomDataSet class and look at one of its member functions:

**def** \_\_getitem\_\_(self, idx):  
 img\_loc = os.path.join(self.root\_dir, self.sub\_dir, self.total\_imgs[idx])  
 image = Image.open(img\_loc).convert(**"RGB"**)  
  
 sanity\_image = []  
 tensor\_image = []  
 **if** self.transform:  
 tensor\_image = self.transform(image)  
 sanity\_image = self.sanity\_transform(image)  
  
 **return** tensor\_image, sanity\_image

Let’s go through this line by line:

img\_loc = os.path.join(self.root\_dir, self.sub\_dir, self.total\_imgs[idx])

I already know what self.root\_dir, self.sub\_dir and self.total\_imgs[idx] are. Because I initialized them when I created the object for the CustomDataSet class. Self.total\_imgs is a list of frame names. So, to actually get the image, we have to define the image location. Then, we load the actual image by:

image = Image.open(img\_loc).convert(**"RGB"**)

And now, we apply transformations to it before returning it:

sanity\_image = []  
 tensor\_image = []  
 **if** self.transform:  
 tensor\_image = self.transform(image)  
 sanity\_image = self.sanity\_transform(image)  
  
 **return** tensor\_image, sanity\_image

Recall the transformations:

transform = transforms.Compose([transforms.Resize((224,224)),  
 transforms.ToTensor(),  
 transforms.Normalize(mean=[0.485, 0.456, 0.406], std=[0.229, 0.224, 0.225])])

sanity\_transform = transforms.Compose([transforms.ToTensor()])

What is sanity\_transform? This is just for my own reference. I want to keep track of the original image as well. This will be explained in more detail later on. For now, let’s go back to our original code:

masked\_image\_array = []  
sanity\_masked\_image\_array = []  
**for** i, (image\_batch, sanity\_image\_batch) **in** enumerate(train\_loader):  
 masked\_image\_array.append(image\_batch)  
 sanity\_masked\_image\_array.append(sanity\_image\_batch)

So, here, I load the image tensors to a list. Keep in mind that each entry of this list is not just a single image tensor but a full batch of images. So, let there be some index and so masked\_image\_array[index] will return a tensor of dimension (19, 224, 224, 3). So, imagine a single entry of this list as a stack of 19 images and each image has 3 channels and is 224 by 224. The same idea applies to sanity\_masked\_image\_array but for this we are not resizing the image so a single entry of this array is a tensor of dimensions (19, 1280, 720, 3). After this, we want the original raw image (we crop out the other agents from this image):

unmasked\_img\_array = []  
**for** frame **in** data[start\_index:end\_index]:  
 **if** frame[**'labels'**] **is not None**:  
 unmasked\_img\_array.append(frame[**'name'**])

Once again, because the data json object and bounding box text file are identical (in the sense that, the image in the json file and the dictionary in the text file at the same index, correspond), that means we can safely index the data object and grab the unmasked images. Now, we create the neural network for the extracting features from the masked images: (masked\_output\_ftrs is 20)

*#convnet for masked image*mask\_conv\_model = models.resnet50(pretrained=**True**)  
**for** param **in** mask\_conv\_model.parameters():  
 param.requires\_grad = **False**num\_ftrs = mask\_conv\_model.fc.in\_features  
mask\_conv\_model.fc = nn.Linear(num\_ftrs, masked\_output\_ftrs)  
mask\_conv\_model = mask\_conv\_model.to(device)  
mask\_params = []  
**for** name, param **in** mask\_conv\_model.named\_parameters():  
 **if** param.requires\_grad == **True**:  
 mask\_params.append(param)

So, we use resnet50 but obviously resnet50 is used for classification and it classifies images into 1000 classes and so the final output vector is 1000 elements long. We only need 20. So, we take a pre-trained resnet50 (nobody has the time or patience to train a resnet50 from scratch) and replace the last layer with our own layer. We will only backprop up to this layer and we will never go further back from here. For our intents and purpose, we only need to train the part of the resnet50 that is responsible for extracting features. To have only some parameters be changeable by backprop, we first get all the parameters in the neural network and make them untrainable:

**for** param **in** mask\_conv\_model.parameters():  
 param.requires\_grad = **False**

Then, we replace the final layer of our neural network (mask\_conv\_model.fc refers to the last layer of our neural net):

num\_ftrs = mask\_conv\_model.fc.in\_features  
mask\_conv\_model.fc = nn.Linear(num\_ftrs, masked\_output\_ftrs)

Now, we put assign our custom neural network to a GPU and create a list of parameters. But this time, this list of parameters will only contain those parameters that are trainable. By default, all parameters of a neural network are trainable but we set all parameters to not be trained. And then, after doing that, we took out the final layer and replaced it with a new layer. However, by convention, this new layer has all parameters trainable by default. So, what we do now, is to just go through all the parameters of our custom neural net and only add those which are trainable:

mask\_conv\_model = mask\_conv\_model.to(device)  
mask\_params = []  
**for** name, param **in** mask\_conv\_model.named\_parameters():  
 **if** param.requires\_grad == **True**:  
 mask\_params.append(param)

Similar idea for the neural network that is responsible for extracting features from the cropped images. However, this time, we have more than one neural network (5 in this case):

*#convnets for unmasked images*other\_agent\_models = []  
other\_agent\_model\_optimizers = []  
**for** i **in** range(other\_agent\_limit):  
 other\_agent\_model = models.resnet50(pretrained=**True**)  
 **for** param **in** other\_agent\_model.parameters():  
 param.requires\_grad = **False** num\_ftrs = other\_agent\_model.fc.in\_features  
 other\_agent\_model.fc = nn.Linear(num\_ftrs, unmasked\_output\_ftrs)  
 other\_agent\_model = other\_agent\_model.to(device)  
 other\_agent\_params = []  
 **for** name, param **in** other\_agent\_model.named\_parameters():  
 **if** param.requires\_grad == **True**:  
 other\_agent\_params.append(param)  
 other\_agent\_models.append(other\_agent\_model)  
 other\_agent\_model\_optimizer = optim.Adam(other\_agent\_params, lr=learning\_rate)  
 other\_agent\_model\_optimizers.append(other\_agent\_model\_optimizer)

And finally, we initialize the neural network for the encoder-decoder-RNN:

encoder = EncoderRNN(input\_size=masked\_output\_ftrs + 4 + other\_agent\_limit \* (unmasked\_output\_ftrs + 4),  
 hidden\_size=encoder\_hidden\_size, n\_layers=encoder\_n\_layers, dropout=encoder\_dropout)  
encoder = encoder.to(device)  
decoder = LuongAttnDecoderRNN(attn\_model, input\_size=4, hidden\_size=decoder\_hidden\_size, output\_size=4,  
 n\_layers=decoder\_n\_layers, dropout=decoder\_dropout)  
decoder = decoder.to(device)

A detailed explanation of this can be found here: <https://pytorch.org/tutorials/beginner/chatbot_tutorial.html#define-models>

Define the loss function (mean squared error) and optimizers:

loss = nn.MSELoss()  
  
mask\_conv\_model\_optimizer = optim.Adam(mask\_params, lr=learning\_rate)  
encoder\_optimizer = optim.Adam(encoder.parameters(), lr=learning\_rate)  
decoder\_optimizer = optim.Adam(decoder.parameters(), lr=learning\_rate \* decoder\_learning\_ratio)

Since agents are repeated across frames, we want to make sure that there is only one instance of each agent in our final list of agents. So, for example, if agent 1 is in frame 1 and also in frame 2 and so on, we need to make sure that our final list of agents only has agent 1 in it once and not as many times as it appears across all images in the sequence:

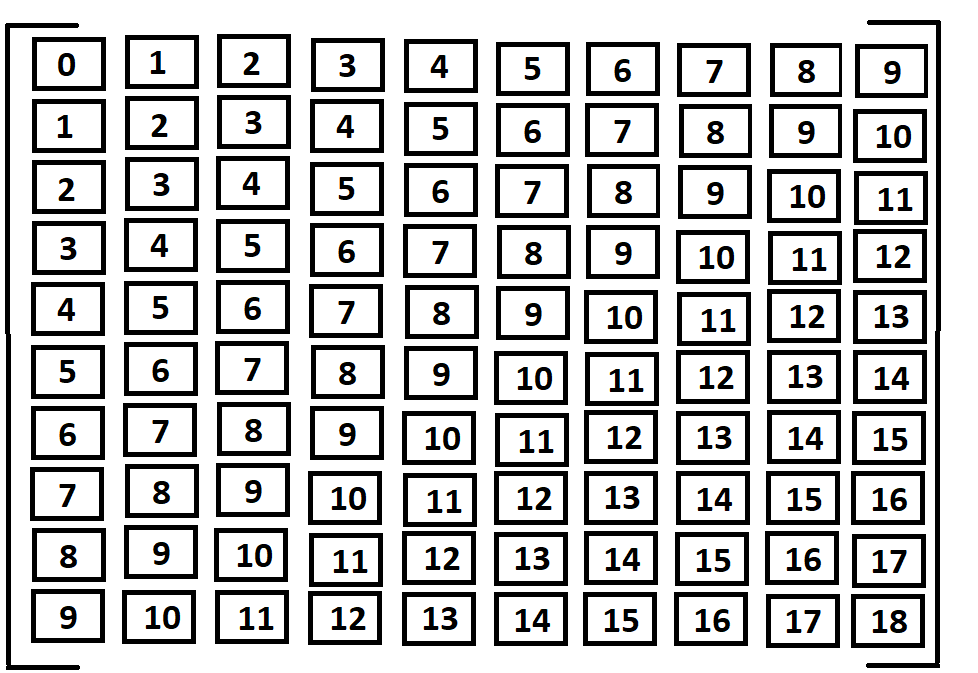
all\_agents\_unique = list(set(all\_agents))  
print(**"TOTAL UNIQUE AGENTS "** + str(len(all\_agents\_unique)))

Now, we start looping through our list of agents. For each agent, we first grab five other agents randomly:

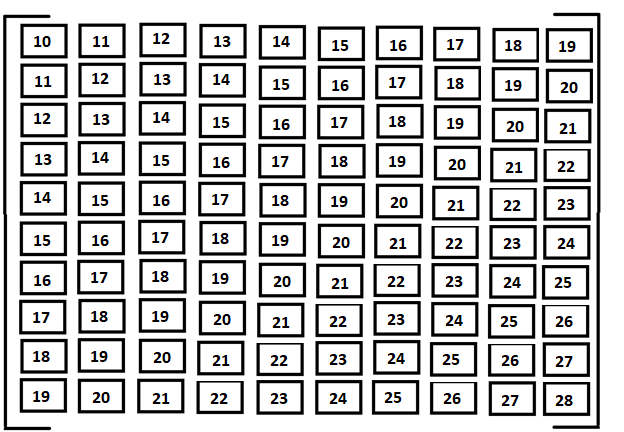
**for** current\_agent **in** all\_agents\_unique:  
 agentFile = open(**"AGENT "** + str(current\_agent) + **".txt"**, **'w+'**)  
  
 all\_agents\_copy = all\_agents\_unique.copy()  
 **for** i **in** range(forwards\_per\_agent):  
 other\_agents = []  
 random.shuffle(all\_agents\_copy)  
 k = 0  
 **for** agent **in** all\_agents\_copy:  
 **if** current\_agent != agent **and** k < other\_agent\_limit:  
 other\_agents.append(agent)  
 k += 1  
 **if** k >= other\_agent\_limit:  
 **break** other\_agents\_string = **','**.join(str(e) **for** e **in** other\_agents)  
 SANITY\_CHECK\_DIR = os.path.join(ROOT\_DIR, **"sanity\_check\_agent"** + str(current\_agent) + **"to"** + other\_agents\_string)  
 **try**:  
 os.makedirs(SANITY\_CHECK\_DIR)  
 **except** FileExistsError:  
 **pass**

agentFile is for my own reference. In this text file, I want to record all the steps taken for organizing the data. This will become more clear as I step you through the code a bit more. Now, onwards, we are going to make a dictionary object. The key of the dictionary object is going to be the batch number and the value will be a tuple that contains: (tensor of masked images, the 4 bounding box coordinates of the current agent across several sequence of images, the 4 bounding box coordinates of our current agent in the last image of each sequence, tensor of bounding box coordinates for the current agent in the target sequence of images, list of bounding box values of other agents across a sequence of images, list of cropped images of other agents across a sequence of images, a list of Booleans telling us whether the selected other agent was present in the batch or not)

Obviously, this is a mouthful and requires visual explanation. With our example of 19 images, the tensor of masked images will have the dimension of (19, 224, 224, 3). As explained before, our data is repackaged from a (19, 74) tensor to a (10, 10, 74) tensor. Here, I put the image again for reference.

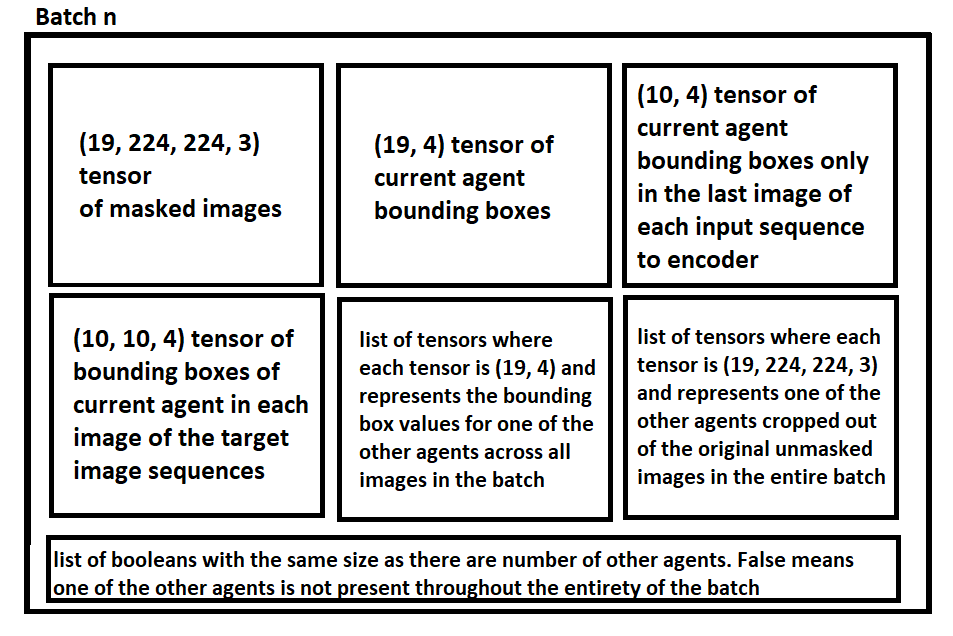


The column is a single sequence of images and in the above example we have 10 sequences and each sequence is 10 images long. For the decoder, we need the bounding box coordinates of the current agent from the last image in each sequence. So, in the above example, this means we need the bounding box coordinates from image 9 to 18. So, that means the 4 bounding box coordinates of our current agent in the last image of each sequence is nothing but a tensor of dimension (10, 4). The target tensor is the following:



So, here, the first sequence (the first column) is a continuation of the first sequence in the input tensor that was sent to the encoder. However, now every box represents the four bounding box coordinates of the current agent. So, to make the difference clear, the input to the encoder is a tensor of size (10, 10, 74) where the 74-element feature vector also includes the bounding box coordinates of the current agent among other information. Meanwhile, the output from the decoder is compared to a tensor of size (10, 10, 4) where the 4 represents the bounding box coordinates.

And finally, the list of bounding box values of the other agents is a list that contains the bounding box values for each of the other agent in the image. Each bounding box coordinate is a (19, 4) tensor. Similarly, the list of cropped images of the other agent contains tensors and each tensor is of dimensions (19, 224, 224, 3). A final visual representation of the dictionary is given below:



The list of Booleans might seem confusing at first. The basic idea is that we can use bounding boxes and cropped versions of the other agent as long as they are present for an entire batch of images. So, for example, if we have a batch of images and we are trying to predict the trajectory of some current agent 1 with reference to other agents 0, 2, 3, 4, 5; then, if agent 0 is not present in all 19 images of the batch, then we will simply ignore its contribution and use a tensor of zeros instead. So, instead of having a tensor of size (19, 224, 224, 3) for the other agent that gives us a feature tensor of size (19, 6) which is combined with the other agent’s bounding box coordinates to get a tensor of size (19, 10), we instead don’t do any feature extraction and just use a tensor of zeros of size (19, 6). So, in this scenario, the (19, 20) feature tensor, obtained from the masked images, is combined with a (19, 4) tensor of bounding box coordinates of the current agent and then it is combined with a (19, 10) tensor of all zeros.

But, how are these tensors extracted? For that, we must dive into the network\_classes.py once again:

**def** getAgentAndOtherAgentBatches(current\_agent, other\_agent, list\_of\_dicts, batch\_size, seq\_len,  
 unmasked\_img\_array, masked\_img\_array, sanity\_masked\_img\_array, transform, device, agentFile,  
 unmasked\_output\_features, img\_dir, sanity\_check\_directory):

This is the function that creates the dictionary we are interested in. Next, we do some basic initialization:

size\_of\_batch = batch\_size + seq\_len - 1  
other\_agent\_batches = 0  
  
other\_agent\_absence\_counter = 0  
other\_agent\_presence\_counter = 0  
  
list\_offset = 0  
dict\_of\_batches = {}

Now, our loop starts:

**while True**:  
 agent\_bbox\_encoder = []  
 target\_bbox = []  
 input\_bbox = []  
 found\_agent = **False** other\_agent\_bbox\_encoder = []  
 other\_agent\_img\_encoder = []  
 found\_other\_agent = **False** other\_agent\_is\_zero = **False** starting\_image = list\_offset \* size\_of\_batch  
 ending\_image = starting\_image + size\_of\_batch  
  
 agentFile.write(**"NOW ATTEMPTING TO GRAB INPUT IMAGES FOR BATCH "** + str(list\_offset) + **"\n"**)  
 agentFile.write(**"THE STARTING IMAGE IS "** + str(starting\_image) + **"\n"**)  
 agentFile.write(**"THE ENDING IMAGE IS "** + str(ending\_image) + **"\n"**)

list\_offset is used to go through images batch-wise. So, list\_offset equal to 0 means we are dealing with the first 19 images (in this case, batch\_size + seq\_len – 1 = 19) and so on. Then, we make sure that we don’t overshoot our list:

**if** (ending\_image <= len(list\_of\_dicts)):  
 image\_counter = starting\_image  
 skip = **False** sanity\_check\_array\_current\_agent = []  
 sanity\_check\_array\_other\_agent = []  
 sanity\_check\_array\_unmasked = []  
 sanity\_check\_array\_masked = []  
 sanity\_check\_counter = 0

So, let’s recap. We define the variable size\_of\_batch which tells us how many images a single batch is supposed to have. In our case, it is 19. Then, we get the starting\_image and ending\_image of this batch. The length of list\_of\_dicts is 100 (in this specific example, it is 100). So, that means we have to be careful that the ending\_image isn’t greater than the length of this list, since this ending\_image is used to index into this list. Now, let’s first grab the bounding boxes of the current agent and the other agent:

**for** dict **in** list\_of\_dicts[starting\_image:ending\_image]:  
 other\_agent\_img = cv2.imread(os.path.join(img\_dir, unmasked\_img\_array[image\_counter]))  
  
 *#get bounding box of agent whose trajectory you wish to predict* **if** current\_agent **in** dict:  
 agent\_bbox\_encoder.append(torch.from\_numpy(dict[current\_agent]).float().unsqueeze(0))  
  
 *#keep copy of current agent for sanity check* this\_agent\_bbox = [abs(i) **for** i **in** dict[current\_agent]]  
 sanity\_check\_this\_agent = other\_agent\_img[int(this\_agent\_bbox[1]):int(this\_agent\_bbox[3]),  
 int(this\_agent\_bbox[0]):int(this\_agent\_bbox[2])]  
 sanity\_check\_this\_agent = cv2.resize(sanity\_check\_this\_agent, (400, 400),  
 interpolation=cv2.INTER\_AREA)  
 sanity\_check\_array\_current\_agent.append(sanity\_check\_this\_agent)  
  
 *#keep copy of unmasked image for sanity check* sanity\_check\_array\_unmasked.append(other\_agent\_img)  
  
 *#keep copy of masked image for sanity check*masked\_img\_tensor = sanity\_masked\_img\_array[list\_offset][sanity\_check\_counter, :, :,]  
original\_masked\_img = transforms.ToPILImage()(masked\_img\_tensor)  
sanity\_check\_array\_masked.append(original\_masked\_img)

*#get bounding box and cropped image of other agents* **if** other\_agent **in** dict **and** current\_agent **in** dict:  
 other\_agent\_bbox\_encoder.append(torch.from\_numpy(dict[other\_agent]).float().unsqueeze(0))  
 other\_agent\_bbox = [abs(i) **for** i **in** dict[other\_agent]]  
 other\_agent\_img\_encoder.append(  
 transform(Image.fromarray(other\_agent\_img  
 [int(other\_agent\_bbox[1]):int(other\_agent\_bbox[3]),  
 int(other\_agent\_bbox[0]):int(other\_agent\_bbox[2])])).unsqueeze(0))  
  
 *#keep copy of other agent, current agent and, mask and unmask scene images for sanity check* sanity\_check\_other\_agent = other\_agent\_img[int(other\_agent\_bbox[1]):int(other\_agent\_bbox[3]),  
 int(other\_agent\_bbox[0]):int(other\_agent\_bbox[2])]  
 sanity\_check\_other\_agent = cv2.resize(sanity\_check\_other\_agent, (400, 400),  
 interpolation=cv2.INTER\_AREA)  
 sanity\_check\_array\_other\_agent.append(sanity\_check\_other\_agent)image\_counter += 1  
 sanity\_check\_counter += 1

So, first we grab the original raw image in which both our current agent and other agent should be located. I say “should” because either of the two (or both) could also be absent from one of the images. Then, we check whether the current agent’s bounding box coordinates are in the dictionary for the image (the current agent’s integer value is used as a key in the dictionary). If it is, then we gab the bounding box coordinates by converting it to a tensor and then putting that tensor in a list. But before putting the tensor to a list, we also add an extra dimension at the beginning. So, that means, if the original tensor has dimensions (4), then after unsqueezing, the dimensions are (1, 4). This will make sense in the end when we combine a list . The code for these actions is given below:

other\_agent\_img = cv2.imread(os.path.join(img\_dir, unmasked\_img\_array[image\_counter]))  
  
*#get bounding box of agent whose trajectory you wish to predict***if** current\_agent **in** dict:  
 agent\_bbox\_encoder.append(torch.from\_numpy(dict[current\_agent]).float().unsqueeze(0))

Now, we do a sanity check. I want to make sure that the bounding box values are the correct ones. The only way to make sure of that is apply these bounding box values on the original raw image and crop out the current agent and other agent. Then, I can save these cropped images and check them to verify that the process of preparing data is going as I planned for it to go. These sanity check operations are nothing more than personal bookkeeping and have nothing to do with the actual machine learning process. You may as well comment these lines out to speed things up a bit:

*#keep copy of current agent for sanity check*this\_agent\_bbox = [abs(i) **for** i **in** dict[current\_agent]]  
sanity\_check\_this\_agent = other\_agent\_img[int(this\_agent\_bbox[1]):int(this\_agent\_bbox[3]),  
 int(this\_agent\_bbox[0]):int(this\_agent\_bbox[2])]  
sanity\_check\_this\_agent = cv2.resize(sanity\_check\_this\_agent, (400, 400),  
 interpolation=cv2.INTER\_AREA)  
sanity\_check\_array\_current\_agent.append(sanity\_check\_this\_agent)

I do a similar sanity check for the unmasked and masked image:

*#keep copy of masked image for sanity check*masked\_img\_tensor = sanity\_masked\_img\_array[list\_offset][sanity\_check\_counter, :, :,]  
original\_masked\_img = transforms.ToPILImage()(masked\_img\_tensor)  
sanity\_check\_array\_masked.append(original\_masked\_img)

I repeat the above procedure for the other agent too:

*#get bounding box and cropped image of other agents***if** other\_agent **in** dict **and** current\_agent **in** dict:  
 other\_agent\_bbox\_encoder.append(torch.from\_numpy(dict[other\_agent]).float().unsqueeze(0))  
 other\_agent\_bbox = [abs(i) **for** i **in** dict[other\_agent]]  
 other\_agent\_img\_encoder.append(  
 transform(Image.fromarray(other\_agent\_img  
 [int(other\_agent\_bbox[1]):int(other\_agent\_bbox[3]),  
 int(other\_agent\_bbox[0]):int(other\_agent\_bbox[2])])).unsqueeze(0))  
  
 *#keep copy of other agent* sanity\_check\_other\_agent = other\_agent\_img[int(other\_agent\_bbox[1]):int(other\_agent\_bbox[3]),  
 int(other\_agent\_bbox[0]):int(other\_agent\_bbox[2])]  
 sanity\_check\_other\_agent = cv2.resize(sanity\_check\_other\_agent, (400, 400),  
 interpolation=cv2.INTER\_AREA)  
 sanity\_check\_array\_other\_agent.append(sanity\_check\_other\_agent)

Now, I count the number of instances of current agent and the other agent. Ideally, the current agent should be present in every image of a batch. However, that is sometimes not the case and the bounding box might be present in, going with our example, 18 of the 19 images and not in all of them. This is a problem but it doesn’t happen so frequently. In the event that it does happen, I simply decide to ignore the batch of images and move on to the next one hoping that the current agent is present in all 19 images of the next batch. The same reasoning is used for the other agent. However, in the event that not all images of the current batch contain the other agent, I don’t skip the entire batch of images. Instead, I use a tensor of zeros for the other agent. This is also a rarity and, when it does happen, the other agent is present in only 3 or 4 of the total 19 images. Anyways, here is the code for doing the above:

**if** len(agent\_bbox\_encoder) == size\_of\_batch **and** len(other\_agent\_bbox\_encoder) == size\_of\_batch:  
 **for** array\_counter **in** range(len(sanity\_check\_array\_current\_agent)):  
 numpy\_horizontal\_concat = np.concatenate((sanity\_check\_array\_current\_agent[array\_counter],  
 sanity\_check\_array\_other\_agent[array\_counter]),axis=1)  
 cv2.imwrite(os.path.join(sanity\_check\_directory, **"THIS\_AGENT\_"** + str(current\_agent) + **"\_OTHER\_AGENT\_"** + str(other\_agent) + **"\_"** + str(array\_counter) + **".jpg"**), numpy\_horizontal\_concat)  
 cv2.imwrite(os.path.join(sanity\_check\_directory, **"UNMASKED\_IMAGE\_"** + str(array\_counter) + **".jpg"**),  
 sanity\_check\_array\_unmasked[array\_counter])  
 sanity\_check\_array\_masked[array\_counter].save(os.path.join(sanity\_check\_directory, **"MASKED\_IMAGE\_"** +  
 str(array\_counter) + **".jpg"**))  
**elif** len(agent\_bbox\_encoder) == size\_of\_batch **and** len(other\_agent\_bbox\_encoder) != size\_of\_batch:  
 **for** array\_counter **in** range(len(sanity\_check\_array\_current\_agent)):  
 cv2.imwrite(os.path.join(sanity\_check\_directory, **"THIS\_AGENT\_"** + str(current\_agent) + **"\_"** +  
 str(array\_counter) + **".jpg"**), sanity\_check\_array\_current\_agent[array\_counter])  
 cv2.imwrite(os.path.join(sanity\_check\_directory, **"UNMASKED\_IMAGE\_"** + str(array\_counter) + **".jpg"**),  
 sanity\_check\_array\_unmasked[array\_counter])  
 sanity\_check\_array\_masked[array\_counter].save(os.path.join(sanity\_check\_directory, **"MASKED\_IMAGE\_"** +  
 str(array\_counter) + **".jpg"**))  
  
**if** len(agent\_bbox\_encoder) != size\_of\_batch:  
 agentFile.write(**"THE NUMBER OF AGENT "** + str(current\_agent) + **" BBOXES GRABBED IS: "** + str(len(agent\_bbox\_encoder)) + **"\n"**)  
 agentFile.write(**"BUT THE REQUIREMENT IS "** + str(size\_of\_batch) + **"\n"**)  
 agentFile.write(**"THEREFORE, I AM SKIPPING TO THE NEXT BATCH!"** + **"\n"**)  
 list\_offset += 1  
 skip = **True  
  
if** len(other\_agent\_bbox\_encoder) != size\_of\_batch:  
 agentFile.write(**"THE NUMBER OF BBOXES, FOR OTHER AGENT "** + str(other\_agent) + **", GRABBED IS: "** + str(len(other\_agent\_bbox\_encoder)) + **"\n"**)  
 agentFile.write(**"BUT THE REQUIREMENT IS "** + str(size\_of\_batch) + **"\n"**)  
 agentFile.write(**"THEREFORE, I WILL USE ZEROS FOR THE OTHER AGENT "** + str(other\_agent) + **" IN THIS BATCH"** + **"\n"**)  
 other\_agent\_bbox\_encoder[:] = []  
 other\_agent\_bbox\_encoder = torch.zeros(size\_of\_batch, unmasked\_output\_features + 4, dtype=torch.float).to(device)  
 other\_agent\_is\_zero = **True**

Of course, I also save all those images, I got previously, for a sanity check. Now, if everything went well (somewhat) and the current agent was present in all images of the batch, then I can go ahead and get the bounding box coordinates of the current agent from the last image of each sequence in the current batch:

**if not** skip:  
 *#get the bbox for every last image in each of your image sequences* agentFile.write(**"NOW GRABBING LAST BBOX OF A SEQUENCE ACROSS A BATCH"** + **"\n"**)  
 starting\_image = starting\_image + seq\_len - 1  
 ending\_image = starting\_image + batch\_size  
 agentFile.write(**"THE STARTING IMAGE IS "** + str(starting\_image) + **"\n"**)  
 agentFile.write(**"THE ENDING IMAGE IS "** + str(ending\_image) + **"\n"**)  
  
 image\_counter = starting\_image  
 sanity\_check\_array\_last\_agent = []  
 other\_agent\_img = cv2.imread(os.path.join(img\_dir, unmasked\_img\_array[image\_counter]))  
 **for** dict **in** list\_of\_dicts[starting\_image:ending\_image]:  
 input\_bbox.append(torch.from\_numpy(dict[current\_agent]).float().unsqueeze(0))  
  
 *#keep copy of last agent for sanity check* this\_agent\_bbox = [abs(i) **for** i **in** dict[current\_agent]]  
 sanity\_check\_this\_agent = other\_agent\_img[int(this\_agent\_bbox[1]):int(this\_agent\_bbox[3]),  
 int(this\_agent\_bbox[0]):int(this\_agent\_bbox[2])]  
 sanity\_check\_this\_agent = cv2.resize(sanity\_check\_this\_agent, (400, 400),  
 interpolation=cv2.INTER\_AREA)  
 sanity\_check\_array\_last\_agent.append(sanity\_check\_this\_agent)  
 image\_counter += 1  
  
 **for** array\_counter **in** range(len(sanity\_check\_array\_last\_agent)):  
 cv2.imwrite(os.path.join(sanity\_check\_directory, **"LAST\_AGENT\_"** + str(current\_agent) + **"\_"** + str(array\_counter) + **".jpg"**), sanity\_check\_array\_last\_agent[array\_counter])

Then, I get the bounding boxes of the current agent from the target sequences of my current sequences:

*#get the bbox for target images*agentFile.write(**"NOW GRABBING TARGET BBOXES"** + **"\n"**)  
found\_agent = **False**starting\_image += 1  
ending\_image = starting\_image + size\_of\_batch  
agentFile.write(**"THE STARTING IMAGE IS "** + str(starting\_image) + **"\n"**)  
agentFile.write(**"THE ENDING IMAGE IS "** + str(ending\_image) + **"\n"**)  
**if** ending\_image < len(list\_of\_dicts):  
 image\_counter = starting\_image  
 sanity\_check\_array\_target\_agent = []  
 other\_agent\_img = cv2.imread(os.path.join(img\_dir, unmasked\_img\_array[image\_counter]))  
 **for** dict **in** list\_of\_dicts[starting\_image:ending\_image]:  
 **if** current\_agent **in** dict:  
 target\_bbox.append(torch.from\_numpy(dict[current\_agent]).float().unsqueeze(0))  
  
 *#keep copy of target agent for sanity check* this\_agent\_bbox = [abs(i) **for** i **in** dict[current\_agent]]  
 sanity\_check\_this\_agent = other\_agent\_img[int(this\_agent\_bbox[1]):int(this\_agent\_bbox[3]),  
 int(this\_agent\_bbox[0]):int(this\_agent\_bbox[2])]  
 sanity\_check\_this\_agent = cv2.resize(sanity\_check\_this\_agent, (400, 400),  
 interpolation=cv2.INTER\_AREA)  
 sanity\_check\_array\_target\_agent.append(sanity\_check\_this\_agent)  
 **if not** found\_agent:  
 agentFile.write(**"FOUND AGENT AT THE BEGINNING OF BATCH "** + str(list\_offset) + **"\n"**)  
 found\_agent = **True** image\_counter += 1  
  
 **for** array\_counter **in** range(len(sanity\_check\_array\_target\_agent)):  
 cv2.imwrite(os.path.join(sanity\_check\_directory, **"TARGET\_AGENT\_"** + str(current\_agent) + **"\_"** + str(array\_counter) + **".jpg"**), sanity\_check\_array\_target\_agent[array\_counter])  
  
**if** len(target\_bbox) != size\_of\_batch:  
 agentFile.write(**"THE NUMBER OF TARGET BBOXES GRABBED FOR "** + str(current\_agent) + **" IS: "** +  
 str(len(target\_bbox)) + **"\n"**)  
 agentFile.write(**"BUT THE REQUIREMENT IS "** + str(size\_of\_batch) + **"\n"**)  
 agentFile.write(**"THEREFORE, I AM SKIPPING TO THE NEXT BATCH!"** + **"\n"**)  
 list\_offset += 1  
 skip = **True**

And now, finally, let’s put it all together:

*#create dictionary of inputs indexed by batch number***if not** skip:  
 agent\_bbox\_encoder\_tensor = torch.cat(agent\_bbox\_encoder).to(device)  
 input\_bbox\_tensor = torch.cat(input\_bbox).to(device)  
 target\_bbox\_tensor = torch.cat(target\_bbox).to(device)  
  
 agentFile.write(**"SANITY CHECK: "** + **"\n"**)  
 agentFile.write(**"SIZE OF INPUT BATCH OF MASKED IMAGES: "** + str(masked\_img\_array[list\_offset].size()) + **"\n"**)  
 agentFile.write(**"FOUND A FULL INPUT BATCH "** + str(agent\_bbox\_encoder\_tensor.size()) + **"\n"**)  
 agentFile.write(**"FOUND A FULL LAST BATCH "** + str(input\_bbox\_tensor.size()) + **"\n"**)  
 agentFile.write(**"FOUND A FULL TARGET BATCH "** + str(target\_bbox\_tensor.size()) + **"\n"**)  
  
 **if not** other\_agent\_is\_zero:  
 other\_agent\_bbox\_encoder\_tensor = torch.cat(other\_agent\_bbox\_encoder).to(device)  
 other\_agent\_img\_encoder\_tensor = torch.cat(other\_agent\_img\_encoder).to(device)  
 dict\_of\_batches[list\_offset] = (masked\_img\_array[list\_offset], agent\_bbox\_encoder\_tensor,  
 input\_bbox\_tensor, target\_bbox\_tensor, [other\_agent\_bbox\_encoder\_tensor],  
 [other\_agent\_img\_encoder\_tensor], [other\_agent\_is\_zero])  
 other\_agent\_batches += 1  
 other\_agent\_presence\_counter += 1  
  
 agentFile.write(  
 **"FOUND A FULL OTHER AGENT BBOX BATCH "** + str(other\_agent\_bbox\_encoder\_tensor.size()) + **"\n"**)  
 agentFile.write(  
 **"FOUND A FULL OTHER AGENT IMAGE BATCH "** + str(other\_agent\_img\_encoder\_tensor.size()) + **"\n"**)  
 **else**:  
 dict\_of\_batches[list\_offset] = (masked\_img\_array[list\_offset], agent\_bbox\_encoder\_tensor,  
 input\_bbox\_tensor, target\_bbox\_tensor, [other\_agent\_bbox\_encoder],  
 [other\_agent\_bbox\_encoder], [other\_agent\_is\_zero])  
 other\_agent\_absence\_counter += 1  
  
 agentFile.write(  
 **"FOUND A FULL OTHER AGENT BBOX BATCH "** + str(other\_agent\_bbox\_encoder.size()) + **"\n"**)  
 agentFile.write(  
 **"FOUND A FULL OTHER AGENT IMAGE BATCH "** + str(other\_agent\_bbox\_encoder.size()) + **"\n"**)

We will stitch all the list elements together along the first dimension. So, if I have a list of 19 tensors and each tensor has size (1, 4), then I can stack them on top of one another to get a tensor of size (19, 4):

agent\_bbox\_encoder\_tensor = torch.cat(agent\_bbox\_encoder).to(device)  
input\_bbox\_tensor = torch.cat(input\_bbox).to(device)  
target\_bbox\_tensor = torch.cat(target\_bbox).to(device)

And then, we do this:

dict\_of\_batches[list\_offset] = (masked\_img\_array[list\_offset], agent\_bbox\_encoder\_tensor,  
 input\_bbox\_tensor, target\_bbox\_tensor, [other\_agent\_bbox\_encoder\_tensor],  
 [other\_agent\_img\_encoder\_tensor], [other\_agent\_is\_zero])

At first glance, you might be wondering why I am putting a one element list at the last three positions. Well, these last three elements are related to the other agent and we have, usually, more than one other agent. So, keeping that in mind, we go back to our main python file where we call this function from network\_classes.py:

dict\_of\_batches = {}  
got\_mandatory\_vals = **False  
for** other\_agent **in** other\_agents:  
 other\_agent\_dict\_of\_batches = getAgentAndOtherAgentBatches(current\_agent, other\_agent, list\_of\_dicts, batch\_size,  
 seq\_len, unmasked\_img\_array, masked\_image\_array, sanity\_masked\_image\_array,  
 transform, device, agentFile, unmasked\_output\_ftrs,  
 os.path.join(ROOT\_DIR, TRAIN\_DIR\_UNMASKED), SANITY\_CHECK\_DIR)  
 **if not** got\_mandatory\_vals:  
 dict\_of\_batches = other\_agent\_dict\_of\_batches  
 got\_mandatory\_vals = **True  
 else**:  
 **for** key **in** dict\_of\_batches.keys():  
 dict\_of\_batches[key][4].append(other\_agent\_dict\_of\_batches[key][4][0])  
 dict\_of\_batches[key][5].append(other\_agent\_dict\_of\_batches[key][5][0])  
 dict\_of\_batches[key][6].append(other\_agent\_dict\_of\_batches[key][6][0])

So, we grab the dictionary for our current agent. This dictionary has as many keys as the number of allowable image batches. By “allowable”, I mean batches in which the current agent was present in all the images. So, for example, current agent 1 is present in batch 0, 1, 2 (and not in 3 while the total number of batches is 4), which means that the dictionary has keys 0, 1 and 2. Then, for each key, you have a tuple (as explained before with the boxy images). Now, the last three parts of this tuple are related to the other agent. What we do is first we get a normal dictionary. Then, for the next dictionaries, we are only interested in the last three items as they are related to one of the other agents. So, for example, our agent 1’s trajectory needs to be predicted against other agents 0, 2 and 3. So, you execute the loop once for other agent 0 and get a dictionary. Then, in the next loop, you get a dictionary for other agent 2 (the current agent is still 1) and you take the last three items and append them to the previous dictionary’s last three items (and that’s why the last three items were a list):

**if not** got\_mandatory\_vals:  
 dict\_of\_batches = other\_agent\_dict\_of\_batches  
 got\_mandatory\_vals = **True  
else**:  
 **for** key **in** dict\_of\_batches.keys():  
 dict\_of\_batches[key][4].append(other\_agent\_dict\_of\_batches[key][4][0])  
 dict\_of\_batches[key][5].append(other\_agent\_dict\_of\_batches[key][5][0])  
 dict\_of\_batches[key][6].append(other\_agent\_dict\_of\_batches[key][6][0])

And finally, we finish off our data preparation:

**for** batch, input **in** dict\_of\_batches.items():  
 mask\_conv\_output = mask\_conv\_model(input[0].to(device))  
 agentFile.write(**"SIZE OF CONV AND FC ON MASK: "** + str(mask\_conv\_output.size()) + **"\n"**)  
 mask\_conv\_output = torch.cat((mask\_conv\_output, input[1]), 1)  
  
 other\_agent\_outputs = []  
 **for** j **in** range(len(input[6])):  
 **if not** input[6][j]:  
 agentFile.write(**"FOUND "** + str(j) + **" OTHER AGENT"** + **"\n"**)  
 other\_agent\_output = other\_agent\_models[j](input[5][j].to(device))  
 agentFile.write(**"SIZE OF CONV AND FC ON OTHER AGENT CROP: "** + str(other\_agent\_output.size()) + **"\n"**)  
 mask\_conv\_output = torch.cat((mask\_conv\_output, other\_agent\_output, input[4][j]), 1)  
 **else**:  
 agentFile.write(**"OTHER AGENT WAS NOT PRESENT, SO CATING WITH ZEROS INSTEAD"** + **"\n"**)  
 agentFile.write(**"SIZE OF CURRENT AGENT BBOX: "** + str(input[1].size()) + **"\n"**)  
 agentFile.write(**"SIZE OF OTHER AGENT 'ZERO' BBOX: "** + str(input[4][j].size()) + **"\n"**)  
 mask\_conv\_output = torch.cat((mask\_conv\_output, input[4][j]), 1)

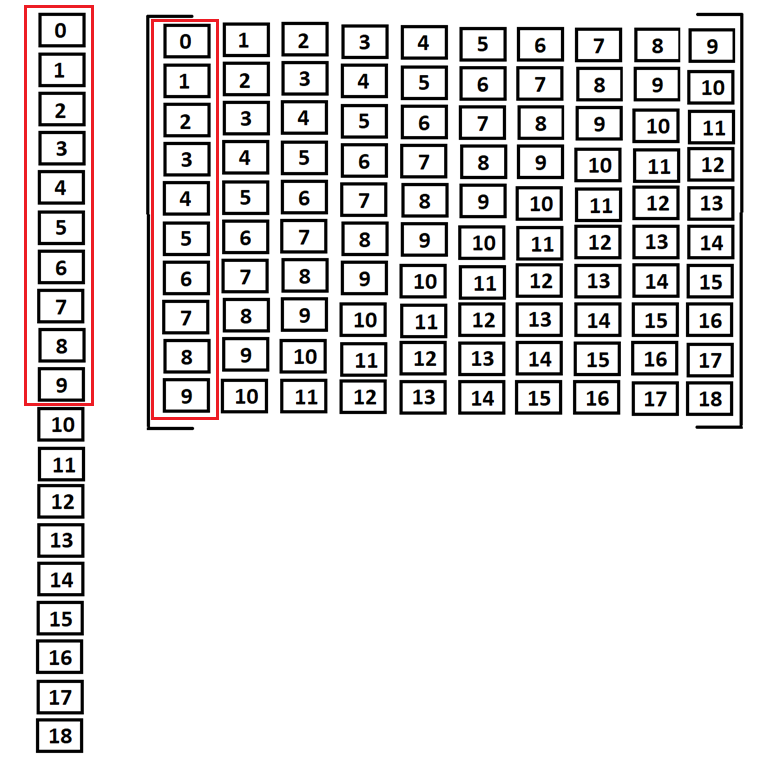
We have a (19, 74) tensor and a (19, 4) tensor. Both of them need to be repackaged to make them suitable for our encoder and decoder.

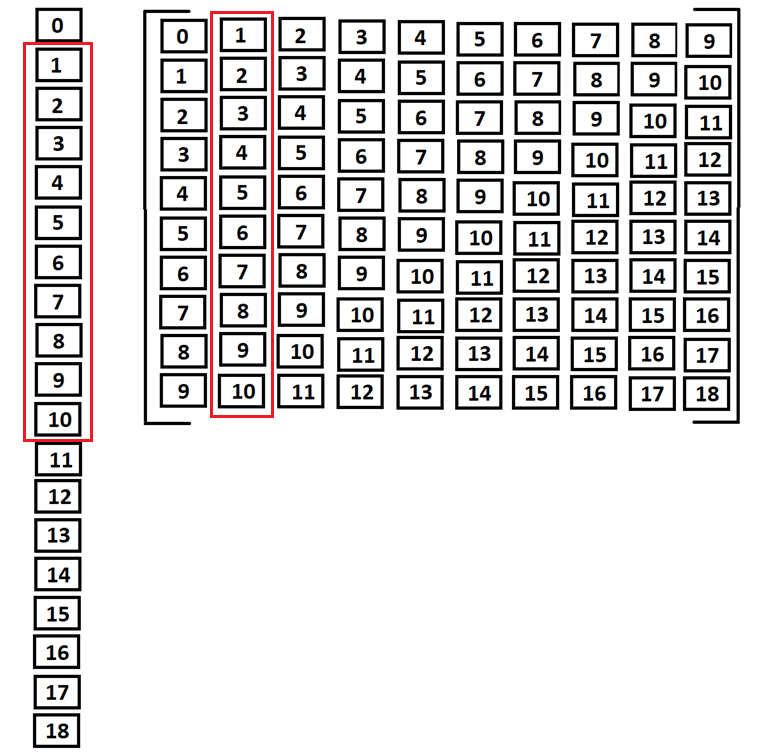
final\_mask\_conv\_features = convertToEncoderInput(mask\_conv\_output, seq\_len, batch\_size)  
final\_target\_features = convertToEncoderInput(input[3], seq\_len, batch\_size)

And this is done by:

**def** convertToEncoderInput(tensorToConvert, seq\_len, batch\_size):  
 offset = 1  
 newTensor = tensorToConvert[0:seq\_len]  
 **for** i **in** range(batch\_size - 1):  
 newTensor = torch.cat((newTensor, tensorToConvert[offset:offset + seq\_len]), dim=0)  
 offset += 1  
 newTensor = newTensor.unsqueeze(-1)  
 newTensor = newTensor.view(seq\_len, batch\_size, newTensor.size()[1])  
  
 **return** newTensor

So, just to give an idea, we have a (19, 74) tensor. Now imagine a sliding window (in red), going over this tensor. The sliding window can capture 10 elements at a time (10 being the sequence length) and then it moves down by one image and the newly captured 10 elements are added next to our previously captured 10 elements. The sliding window repeats this 10 times. Don’t get it? Here is a picture:





And so on. After this, the rest of the code is purely inspired from the pytorch chatbot tutorial. Please visit this link to understand it further: <https://pytorch.org/tutorials/beginner/chatbot_tutorial.html#define-training-procedure>