

## Motivation and Hypothesis

The motivation behind this work stems from my study of camera projection and 3d modeling from the class (CSE P 576A) and a few lectures that I watched online, including one by [Dr. Shubham Tulsiani of Facebook AI and Research](#) given at the Paul Allen School of UW. I was really amazed at the idea of making machines learn to perceive the world in three dimensions by just using 2d images. Some of the examples that Dr. Tulsiani had taken there were really astonishing in the fact that the system could perceive 3d structure of objects from single 2d image. My work here is a very small step in practising some of the ideas about projecting world to the camera and vice versa using neural networks.

In very simple terms, we are given a dataset of images of an object (a car in this case) taken at different angles, assuming that the camera is at a fixed height and a fixed distance from the object in the horizontal plane as it goes around the object in a circle. The task of the neural network is to look at an image and predict the corresponding angle or position of the camera. To validate this quantitatively I will use an error metric which is the RMS of the error in the prediction of the angle value (in radians). For qualitative validation, I insert a synthetic object (a 3-axis pattern) into the scene facing in the direction of the predicted location of the camera.

The earlier few sections of this work include smaller simpler tasks like loading the data with the help of dataloaders, building capability to add a synthetic object (3-axis pattern) in the scene by making use of camera projection matrices, defining our error metric and then moves towards building the neural network training loop that takes in the input images and corresponding camera locations (i.e. angle in radians) as labels and learns to predict the camera position using a CNN. This work is by no means exhaustive or state of the art but really a humble approach towards learning various aspects of 3d scene construction and understanding. For this purpose, I will experiment with some ideas in the neural network that were the takeaways from my last work on building simple CNN to categorize objects in the CIFAR dataset. Lastly I will conclude with the learnings developed so far.

## Set up the data loaders and visualize a mini-batch

For this part, I filled in the `setup_xforms_and_data_loaders` function with the help of the comments provided in the starter code. I basically added a line to code to return train and test data loaders as below:

```
train_loader, test_loader = data_io.data_loaders.setup_data_loaders(par)
```

To validate this, I ran the unit tests using `run_all_unit_tests()` method:

```
py pose_regress.py --mode=unit_test --outdir=out --h5_filename=data/pontiac_360.h5  
--split_name=cvd_split_every5
```

So I provided the `split_name` as `cvd_split_every5` and I was able to see that the unit test `ut_setup_xforms_and_data_loaders` returned `True`.

```

device: cpu
device: cpu
-----run_all_unit_tests()-----
-----ut_setup_xforms_and_data_loaders()-----
-----setup_xforms_and_data_loaders()-----
----- setup_data_loaders()
True
-----setup_xforms_and_data_loaders()-----
----- setup_data_loaders()

```

Next, I proceeded to complete the `load_batch_of_data` function. This was also simple. I basically wrapped the `data_loader` in `iter` and then called `next` on it to get a batch of data.

```
batch_data = next(iter(data_loader))
```

After doing this, I ran the unit tests again to verify and I could see that `ut_load_batch_of_data` also returned `True` this time.

```

device: cpu
device: cpu
-----run_all_unit_tests()-----
-----ut_setup_xforms_and_data_loaders()-----
-----setup_xforms_and_data_loaders()-----
----- setup_data_loaders()
True
-----setup_xforms_and_data_loaders()-----
----- setup_data_loaders()
-----ut_load_batch_of_data()-----
True

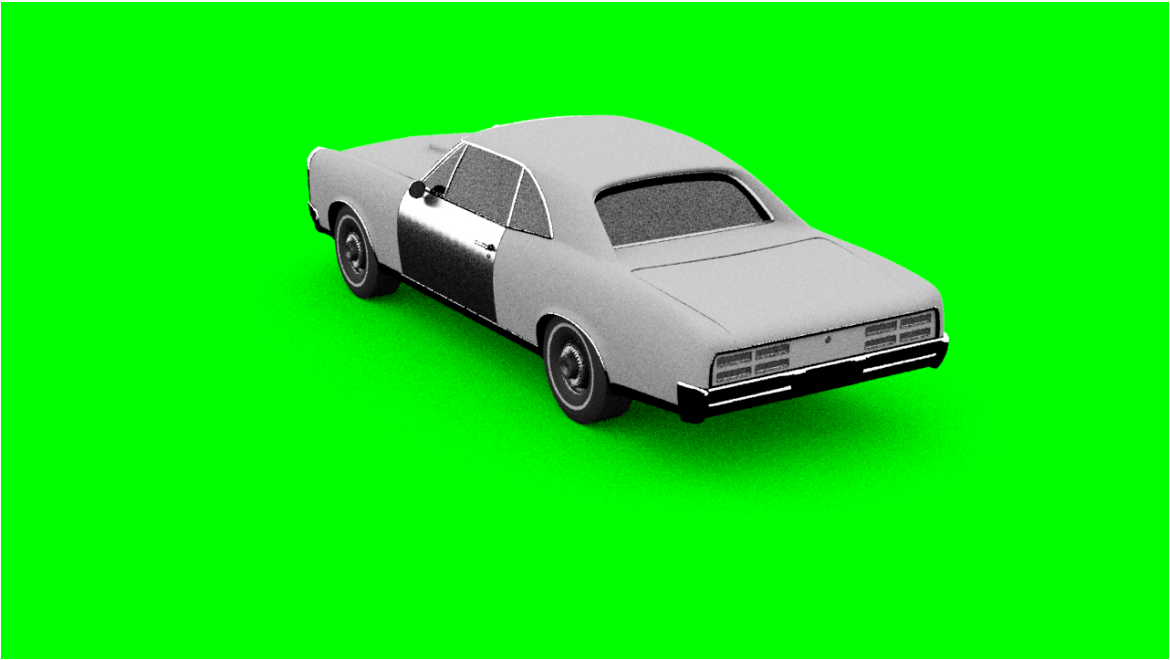

```

This means that our data loaders are correctly setup now.

Let us have a look at some of the input images to see what we are dealing with here

| Image | Camera<br>Rotation |
|-------|--------------------|
|-------|--------------------|

| Image   | Camera Rotation |
|---|-----------------|
|   | 0.00            |
|  | 0.62            |

| Image   | Camera<br>Rotation |
|---|--------------------|
|   | 2.12               |
|  | 3.70               |

Image

Camera  
Rotation

4.17

We can see that as the angle increases, the camera is moving in clockwise direction (as seen from the top) around the object which creates different perspectives of the object.

## Insert a synthetic object into the scene

For this part, I basically used the 3D projection formulas used in the class together with a reading of LookAt calculations provided by the course staff. The basic understanding of camera projection and 3D vector products is good enough to do this work. I read a really nice article on [scratchapixel.com](https://scratchapixel.com) that explain the projection matrices through the diagram below:

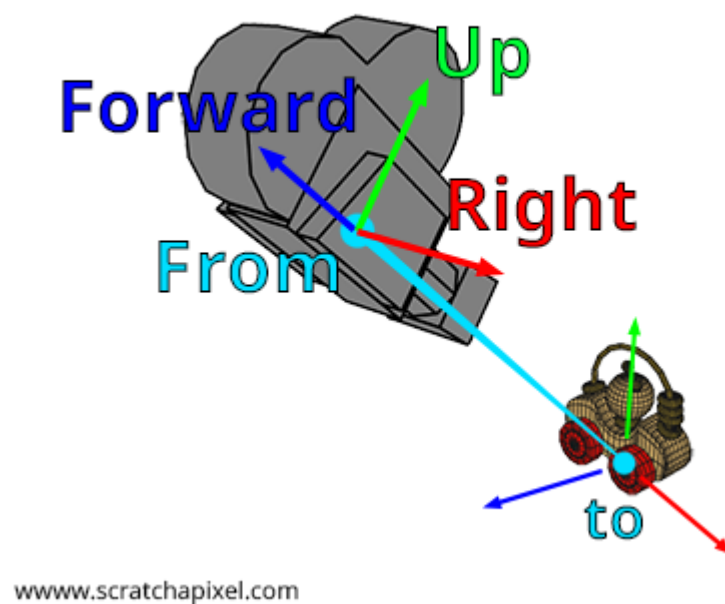


Image Source [ScratchPixel](https://scratchapixel.com)

In the code comments P1 is a transformation in the homogeneous coordinates that moves the camera's coordinate system (3-axis) from its old location, coincident with the world coordinate system to a new location, consistent with "eye" and "look at." For the calculation of P1, I first calculated three unit vectors i.e. `z_axis`, `x_axis`, and `y_axis` of the camera's new coordinate system.

The first step is to calculate the `z_axis` which is basically the same direction in which camera looks at the origin. This means

```
z_axis is a unit vector in the direction (at_world - eye_world)
```

Now, since the camera should be horizontal i.e has no roll, to get the `x_axis`, all that is needed is to cross product the vertical axis (world) `[0, 0, 1]` with camera's `z-axis`.

```
x_axis is a unit vector in the direction cross_product(vertical_axis_unit_vector, z_axis)
```

Finally, it's easy to get `y_axis` by cross product of `z_axis` and `x_axis`.

```
y_axis is a unit vector in the direction cross_product(z_axis, x_axis)
```

Once I am done calculating the `x`, `y` and `z` axes of camera coordinate system, I needed to create the `R` matrix which is basically `x`, `y` and `z` axes of camera coordinate system arranged as columns. And the `T` vector is basically the displacement/translation of camera which is `eye_world`.

Using these, I was able to form the matrix `P1`. This is the projection from camera to world. But since we want the projection of world on camera, we take an inverse of `P1`.

Now, there is another way to calculate the world to camera projection which is using the calculation of `P` and then taking its transpose.

To form the matrix `P`, I followed the following steps:

```
V = -np.dot(R.T, T)
P[0:3, 0:3] = R
P[-1, 0:3] = V
```

Once `P` is formed, I take the transpose of `P` which gives us the projection of world to camera.

```
P = P.T
```

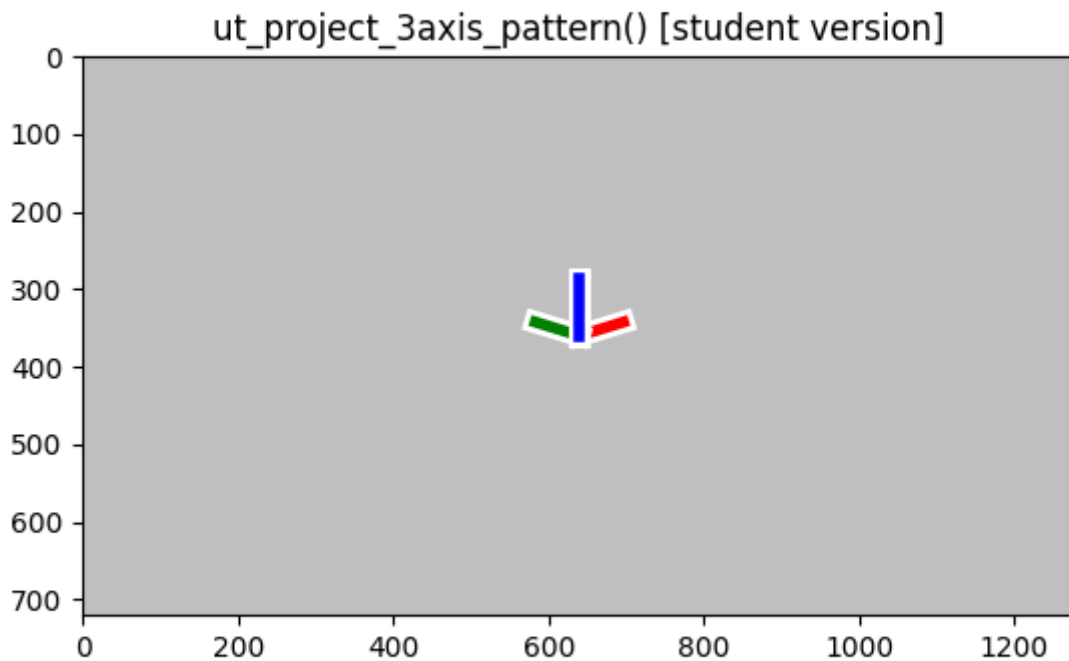
Since we are trying to calculate the same thing but with two different methods, the output from both ways should be the same, so I added an assert statement for checking if `P1` and `P` are equal.

```
assert np.allclose(P, P1)
```

After implementing this, I ran the unit tests to validate if the test `ut_project_3axis_pattern` passes. The test passes successfully as shown below, I also printed the values of `P1` before and after inverse and `P` before and after transpose:

```
-----run_all_unit_tests()-----
-----ut_setup_xforms_and_data_loaders()-----
-----setup_xforms_and_data_loaders()-----
----- setup_data_loaders()
True
-----setup_xforms_and_data_loaders()-----
----- setup_data_loaders()
-----ut_load_batch_of_data()-----
True
-----ut_project_3axis_pattern()-----
P1 [[ 0.707 -0.219 -0.672  14.142]
     [-0.707 -0.219 -0.672  14.142]
     [ 0.000  0.951 -0.309  6.500]
     [ 0.000  0.000  0.000  1.000]]
P1_inv [[ 0.707 -0.707  0.000  0.000]
        [-0.219 -0.219  0.951  0.000]
        [-0.672 -0.672 -0.309  21.030]
        [ 0.000  0.000  0.000  1.000]]
P [[ 0.707 -0.219 -0.672  0.000]
   [-0.707 -0.219 -0.672  0.000]
   [ 0.000  0.951 -0.309  0.000]
   [-0.000  0.000  21.030  1.000]]
P_T [[ 0.707 -0.707  0.000 -0.000]
     [-0.219 -0.219  0.951  0.000]
     [-0.672 -0.672 -0.309  21.030]
     [ 0.000  0.000  0.000  1.000]]
projected 3-axis pattern vsualized in out\test_3axis.png
True
```

We can see that the `P1` after inverse is equal to `P` after transpose. And the `P1` that is returned by the function `camera_xform_from_lookat` is used to generate the 3 axis pattern which is shown below



This looks alright, and we can see that the unit test has passed.

## Define an error metric for angular errors

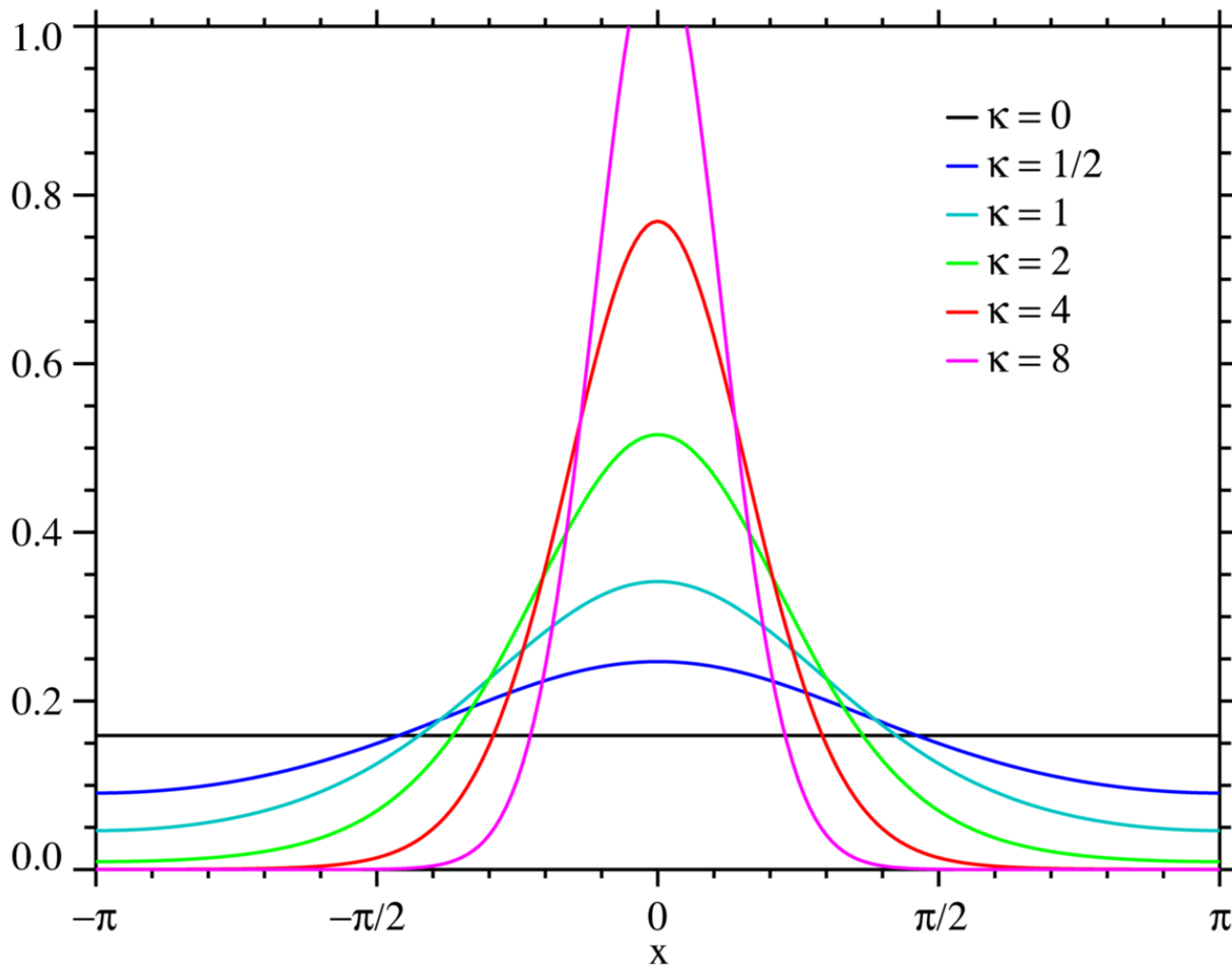
The error metric that I decided to start with is the RMS error of predicted outputs. To implement RMSE, I added a function in `perf_eval/metrics.py` called **rmseerror**

```
def rmseerror(o1, o2):  
    error = o1-o2  
    error_normalized = np.fmod(np.fmod(error+np.pi, 2*np.pi)+2*np.pi, 2*np.pi)-  
np.pi  
    rmse = torch.sqrt(torch.mean(torch.square(error_normalized)))  
    return rmse
```

This function will calculate the root mean square of normalized error to give us an idea of how models are performing.

For the angular errors unit test, we know that  $\kappa$  affects the shape of the vonmises distribution. When  $\kappa$  is low, the distribution is spread out and when  $\kappa$  is high, distribution is squeezed towards zero.





So, I tried different values of kappa from 0.5 to 4.0 and found that the unit test passes at kappa = 2.0.

Finally, after updating the code to succeed all unit tests, I ran the unit tests all at once and here is the output:

```
device: cpu
device: cpu
-----run_all_unit_tests()-----
-----ut_setup_xforms_and_data_loaders()-----
-----setup_xforms_and_data_loaders()-----
----- setup_data_loaders()
True
-----setup_xforms_and_data_loaders()-----
----- setup_data_loaders()
-----ut_load_batch_of_data()-----
True
-----ut_project_3axis_pattern()-----
projected 3-axis pattern vsualized in out\test_3axis.png
True
-----ut_compute_and_viz_angular_error_metrics()-----
viz saved to out\hist_angular_error_1mc.png
True
```

Complete the training code and run evaluations

I implemented the training loop by deciding to start with using `MSELoss` as the loss function and `Adam` as the optimizer. I used a learning rate of `1e-4` and weight decay of `1e-5` for the Adam algorithm.

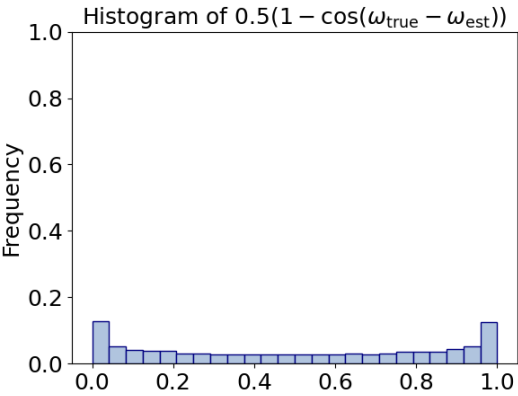
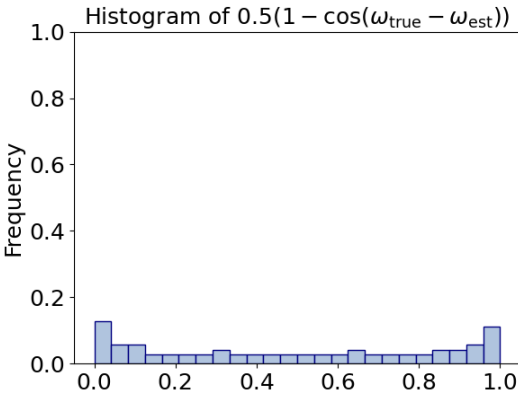
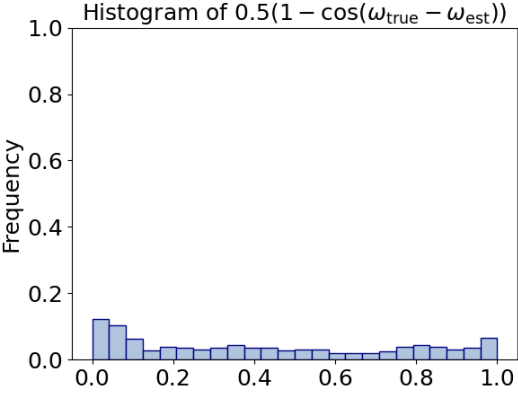
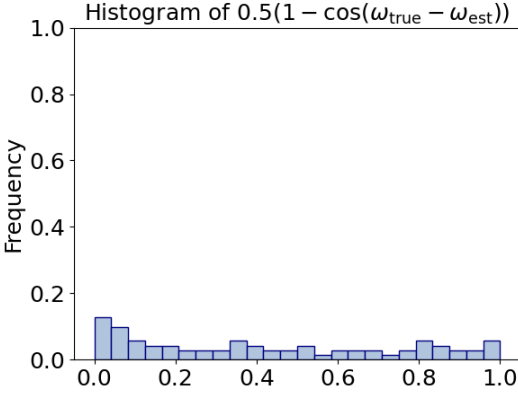
I also modified the code in **perform\_testing method to calculate RMSE** using the function I created in last step and print that together with loss on every evaluation.

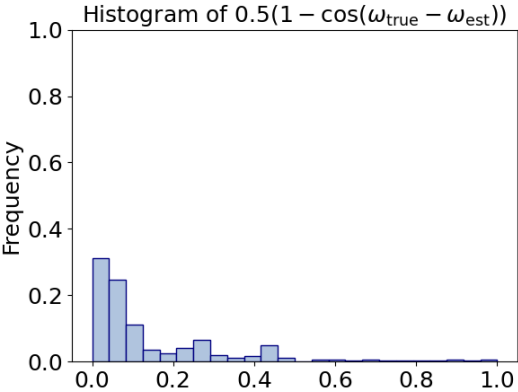
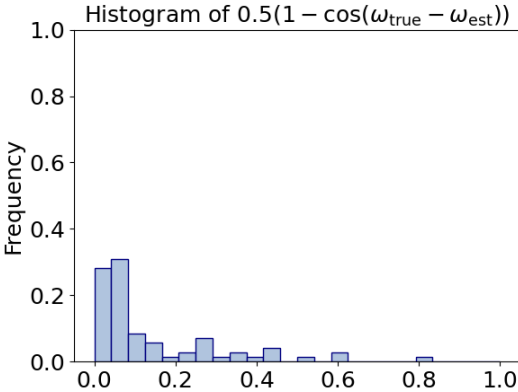
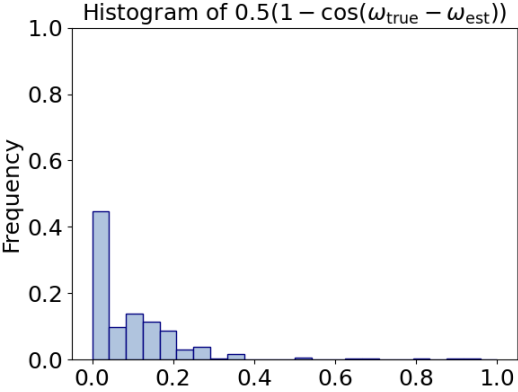
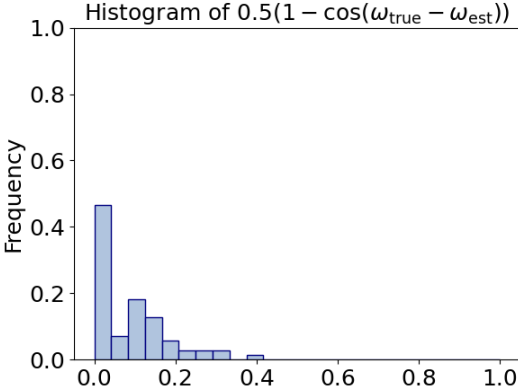
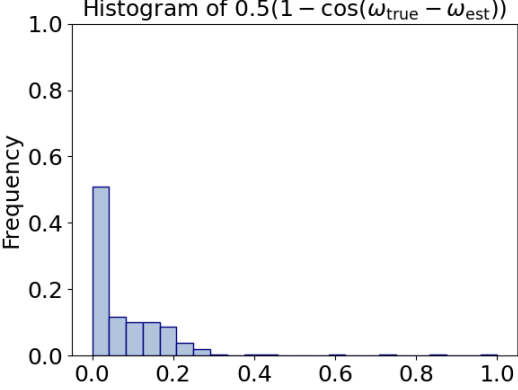
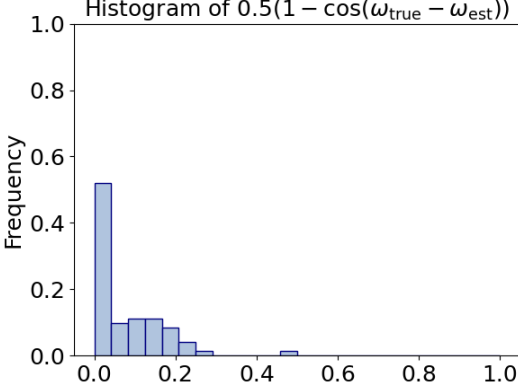
And lastly I updated the `viz_histogram` method in `perf_eval/metrics.py` to **set the number of bins to 24**. This would mean bin error margin is -15 to 15 degrees ( $360/24$ ).

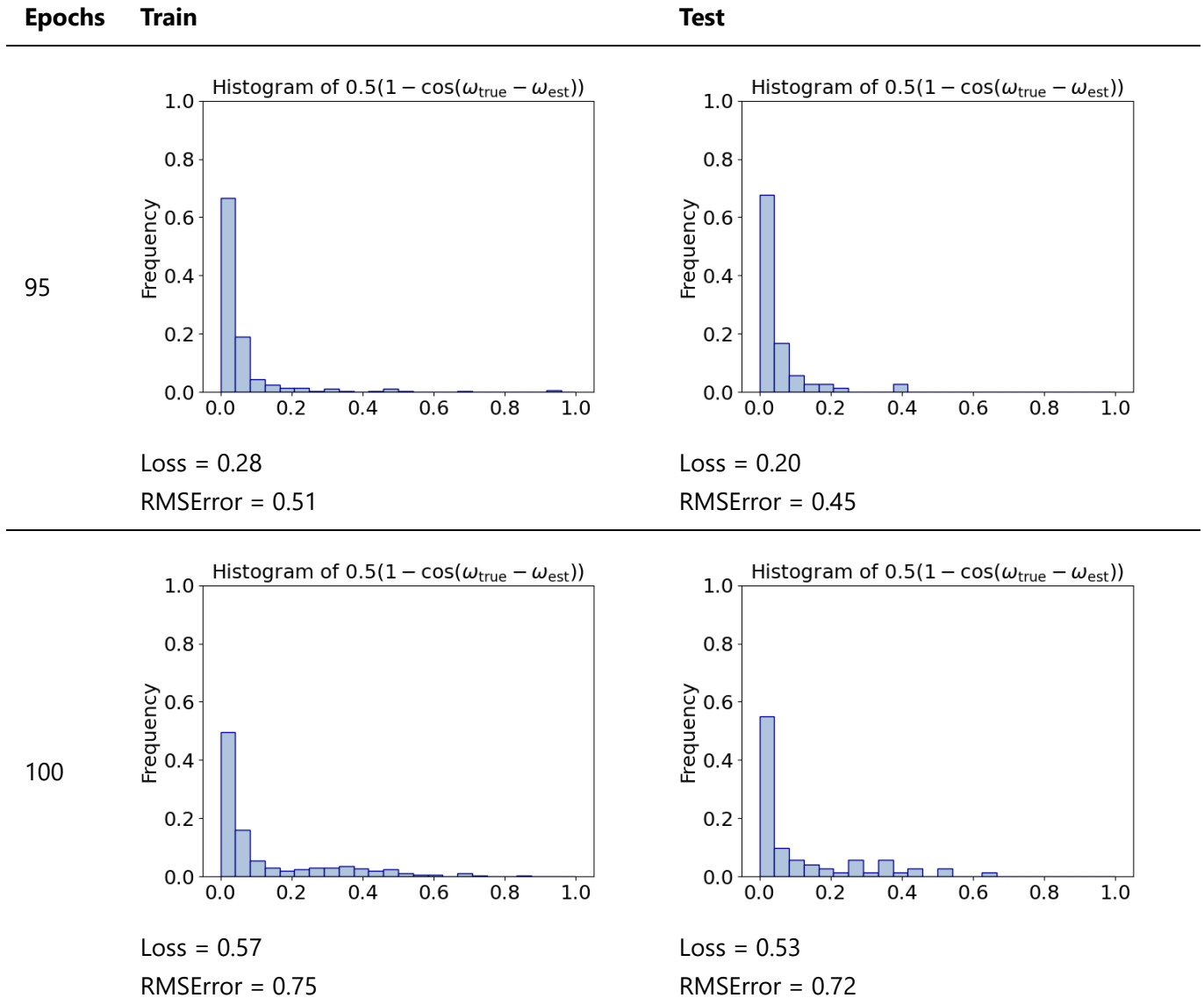
Then I ran the `train_and_test` for 100 epochs with evaluation on every 5th epoch.

```
py pose_regress.py --mode=train_and_test --n_epochs=100 --eval_every_n_epochs=5 --
h5_filename=data/pontiac_360.h5 --outdir=out --split_name=cvd_split_every5 --
viz_pose3d
```

The observations are summarized as below:

| Epochs | Train   | Test   |
|--------|---|--|
| 0      |   |   |
|        | Loss = 3.25<br>RMSError = 1.8   | Loss = 3.16<br>RMSError = 1.77   |
| 5      |  |  |
|        | Loss = 2.62<br>RMSError = 1.61  | Loss = 2.52<br>RMSError = 1.58   |

| Epochs | Train   | Test   |
|--------|---|--|
| 30     |    |    |
|        | Loss = 0.76<br>RMSError = 0.87  | Loss = 0.66<br>RMSError = 0.81   |
|        |   |  |
| 40     |   |   |
|        | Loss = 0.45<br>RMSError = 0.67  | Loss = 0.36<br>RMSError = 0.60   |
|        |   |  |
| 50     |  |  |
|        | Loss = 0.38<br>RMSError = 0.62  | Loss = 0.32<br>RMSError = 0.56   |
|        |   |  |

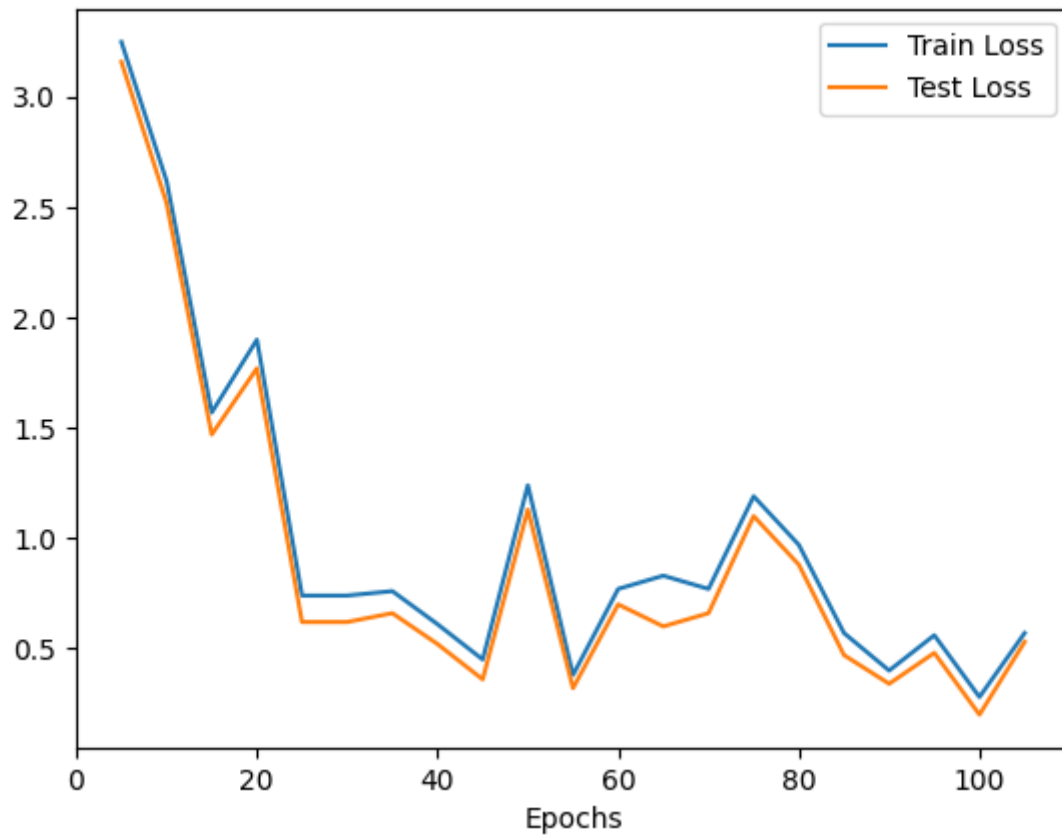


### Observations

We can see here that as the training process goes on, the loss decreases as well as the RMSE metric is decreasing and we achieved the best performance around epoch 95. We can also see that the histogram skews towards the zero bin i.e. most of the errors in prediction are nearing towards zero which is expected as the model learns with successive epochs. This means that our training process is working correctly.

**The best RMSE achieved upto here is Train = 0.51, Test = 0.45.** Those numbers are in radians. In degrees, the train set RMS error is ~30 degrees and test set error is ~26 degrees.

Here is the Loss-Epoch curve for every 5th epoch (captured with evaluation):



We can see that the loss fluctuates here, which could mean the learning rate can be reduced in future iterations.

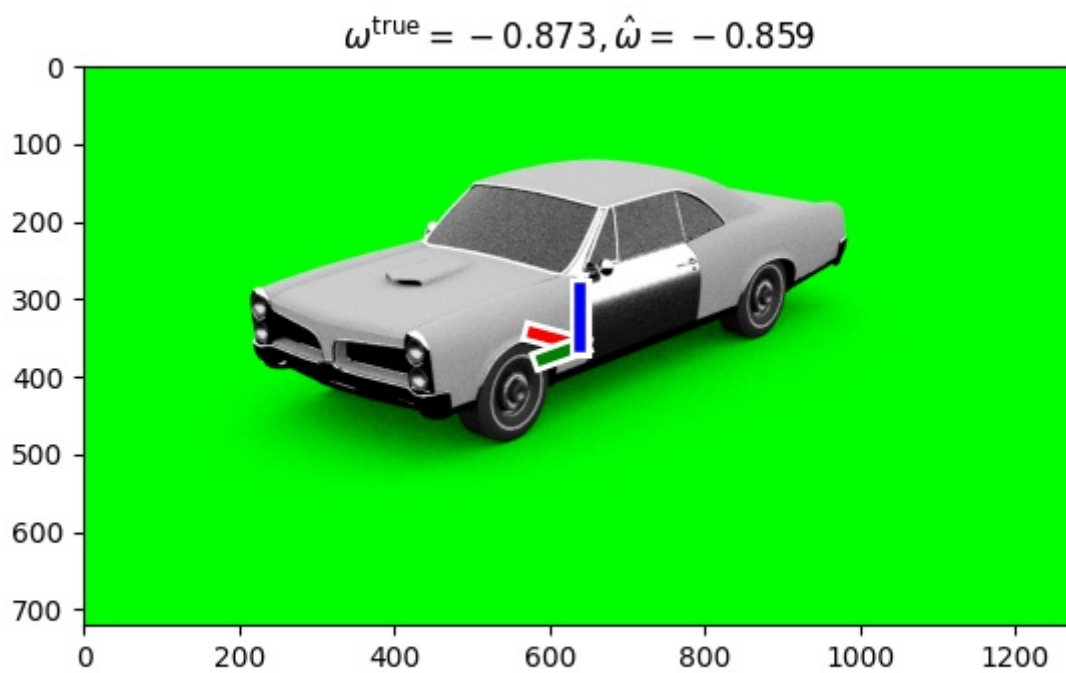
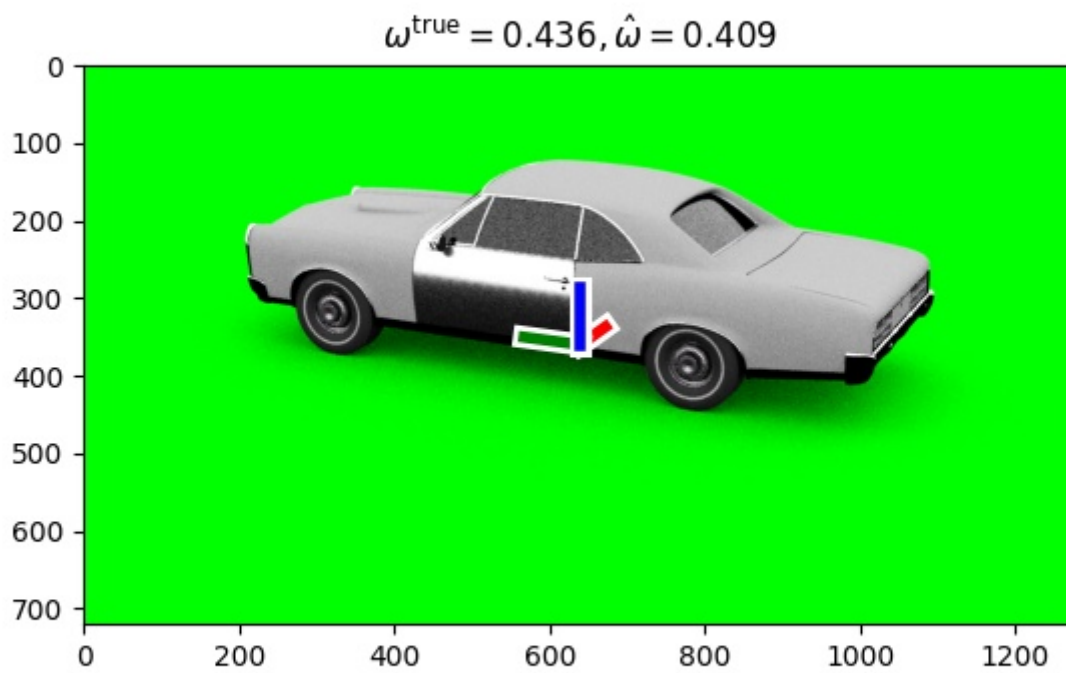
Now, having verified that the model training and prediction is working correctly, I looked at the visualization of these predictions with adding the 3 axis pattern on sample images at varying degrees of rotation.

### **Prediction (3-axis pattern)**

---

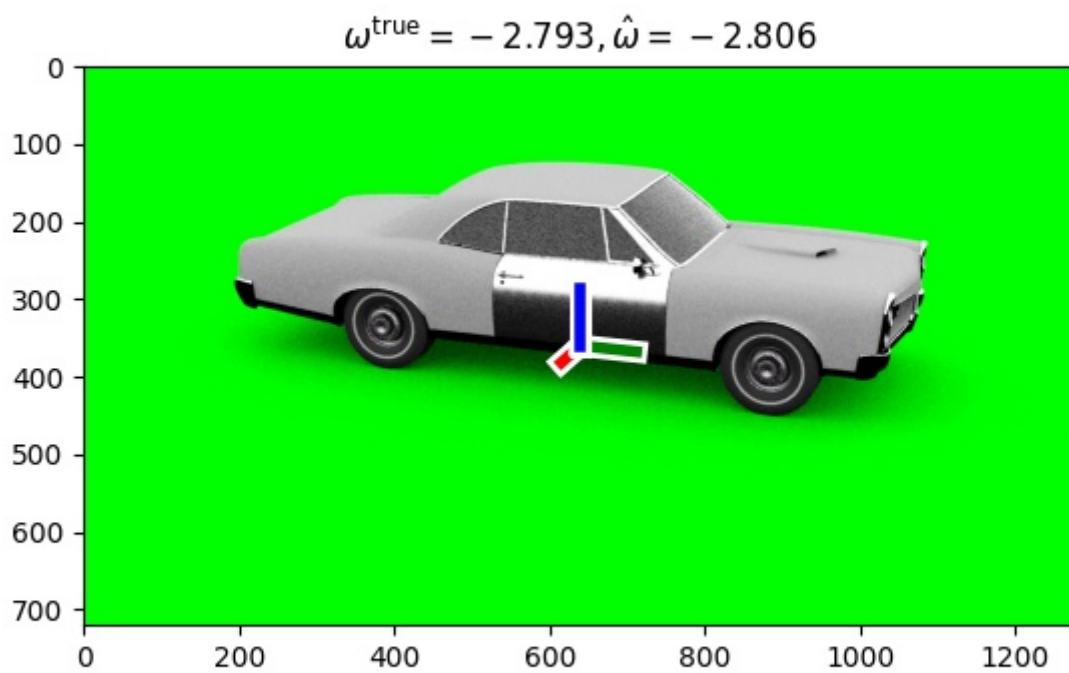
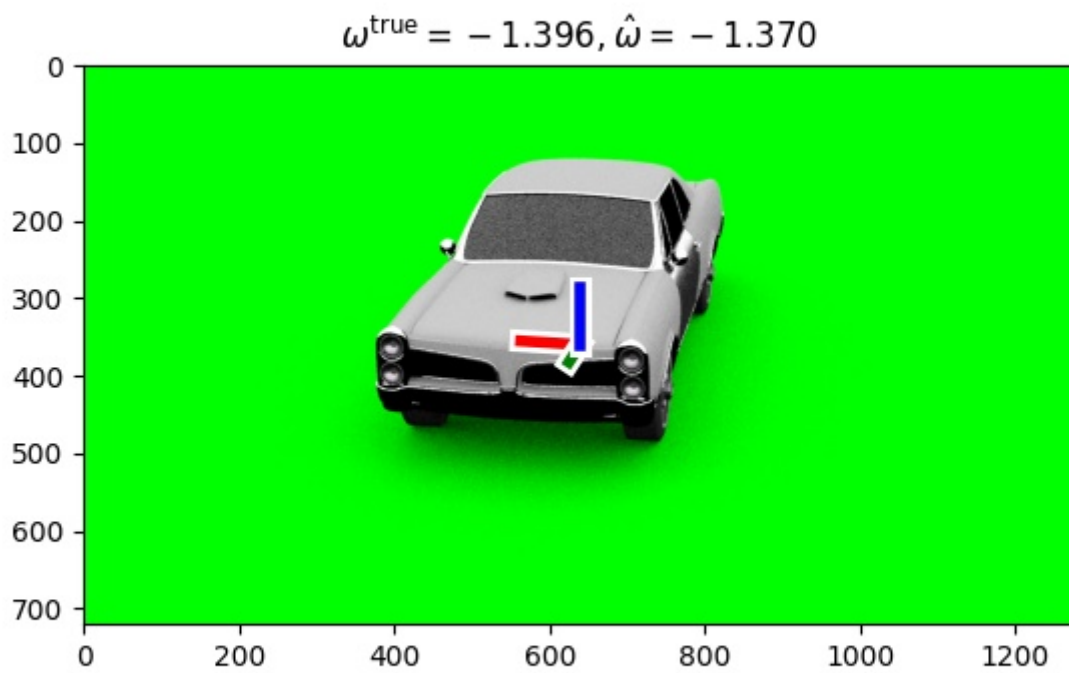
### Prediction (3-axis pattern)

---



**Prediction (3-axis pattern)**

---



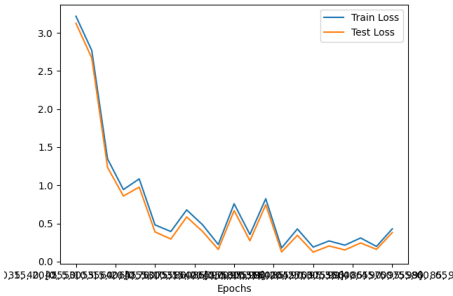

# Extend the model and/or try new data

## Changing the model architecture

I changed the kernel size in first Convolutional layer from 5 to 3 and added BatchNormalization to it. Also I halved the learning rate from 1e-4 to 5e-5.

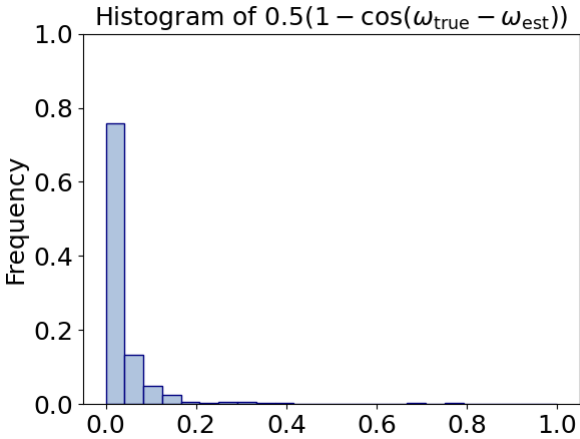
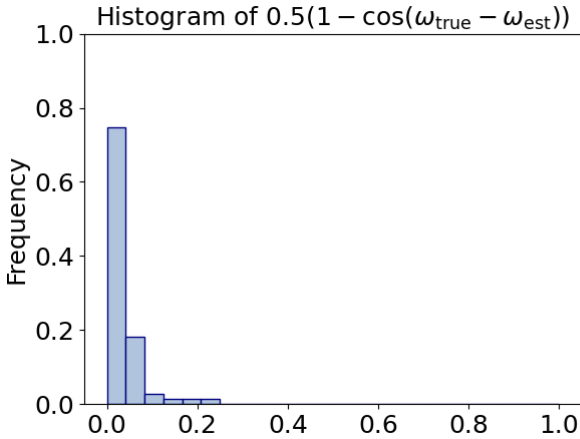
Running this for 100 epochs gives below results

### Observations

| Epoch | Losses | Error (RMSE) | Loss Epoch Curve  | Error Epoch Curve   |
|-------|--------|--------------|---|---|
| 65    | Train  | Train        |  |  |
|       | Loss = | Error =      |   |   |
|       | 0.178  | 0.42         |   |   |
|       | Test   | Test         |   |   |
|       | Loss = | Error =      |   |   |
|       | 0.126  | 0.35         |   |   |

**Best RMSE values are train = 0.42, test = 0.35.** Those numbers are in radians. In degrees, the train set RMS error is ~24 degrees and test set error is ~20 degrees.

Below are the error histograms on test and train sets

| Train set histogram   | Test set histogram   |
|---|--|
|  |  |

This is better than the last architecture where the best RMSE achieved was 0.51 on train set and 0.45 on test set. Through changing kernel size and adding batch normalization we have achieved a **nearly 20% improvement on our error metric** in much lesser number of epochs.

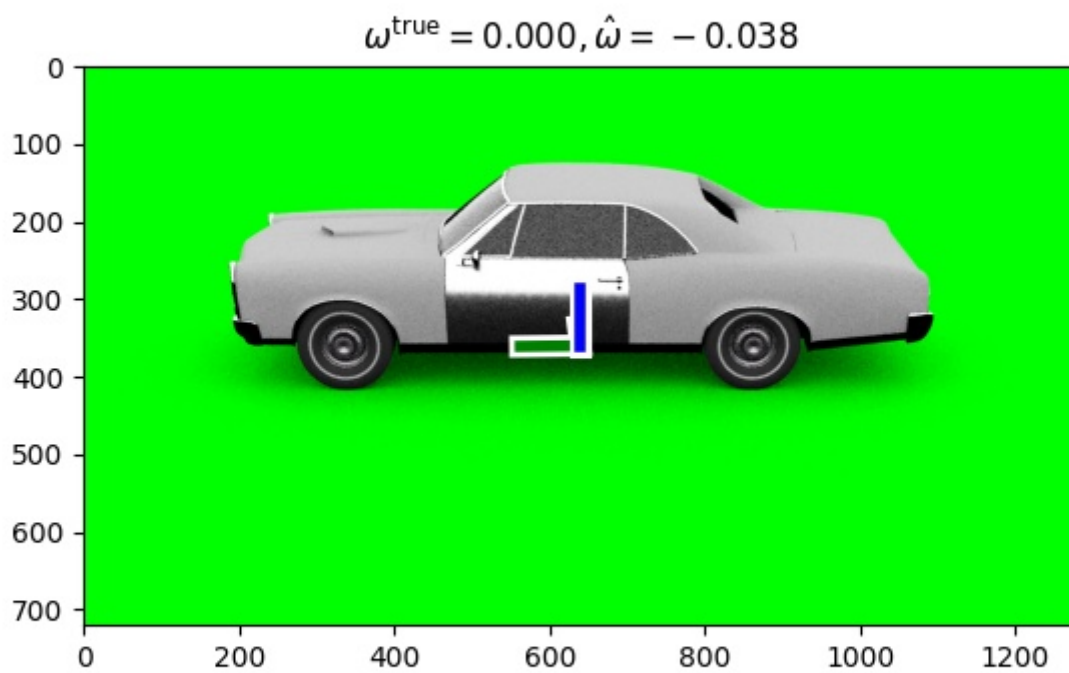
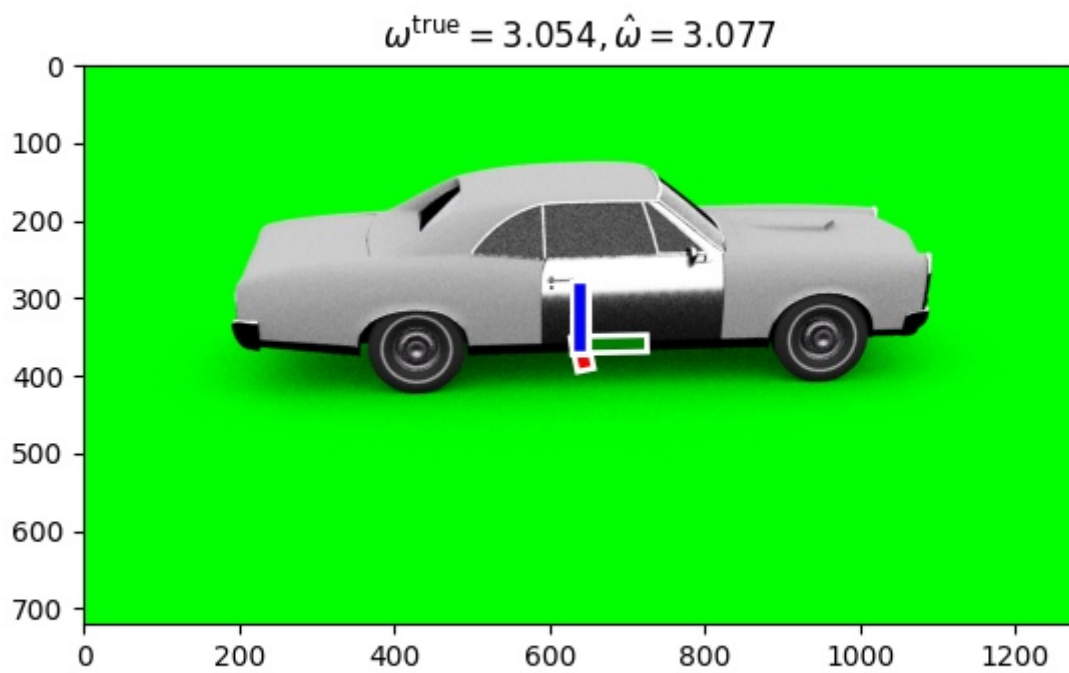
Looking at some examples of projected 3-axis pattern.

### Prediction (3-axis pattern)



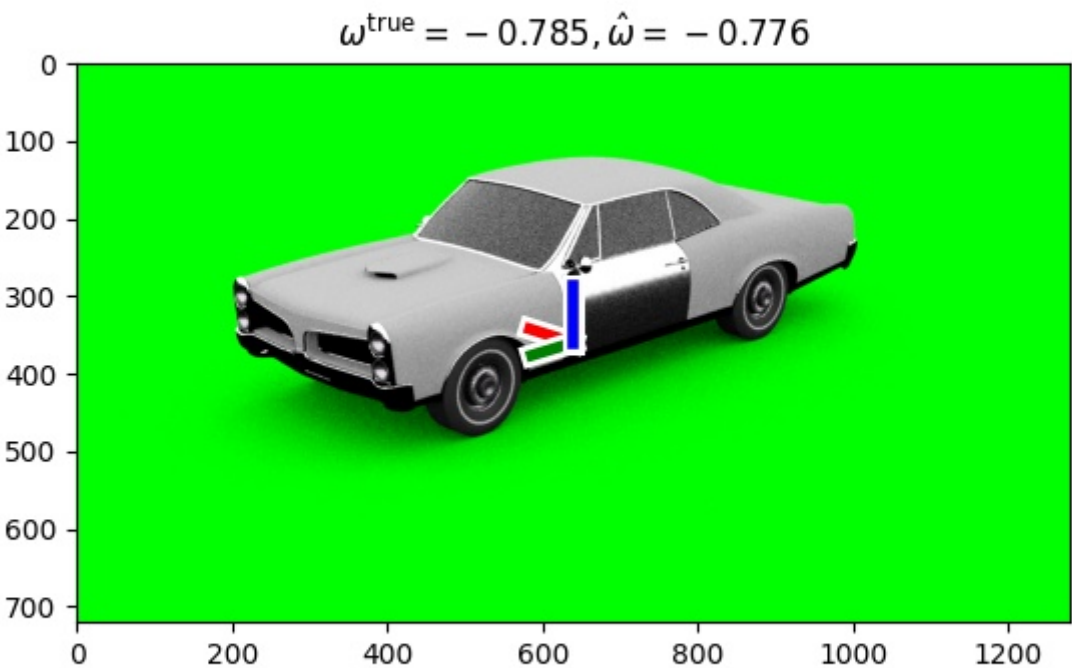
### Prediction (3-axis pattern)

---



Prediction (3-axis pattern)

---



So here are our observations uptil now:

| ModelInfo                | Loss function | Num epochs | RMSE (radians)              | RMSE (degrees)          |
|--------------------------|---------------|------------|-----------------------------|-------------------------|
| Initial Model            | MSELoss       | 95         | Train = 0.51<br>Test = 0.45 | Train = 30<br>Test = 26 |
| Kernel size & batch norm | MSELoss       | 65         | Train = 0.42<br>Test = 0.35 | Train = 24<br>Test = 20 |

We have seen that by changing little things in model architecture, we have achieved a performance gain of about 20%.

Changing the output of neural network to predict cos and sin

For this part, I basically added a new mode called train\_and\_test\_2\_out for which I updated the n\_out parameter to 2.

```
par.update({'viz_hist': True,
           'n_out' : 2 if par['mode'] == 'train_and_test_2_out' else 1})
```

And then added L2 normalization upon regression output in the forward method of the model class.

```
def forward(self, x):
    output = self.m_regress(x)
    if self.mode == 'train_and_test_2_out':
        output = F.normalize(output)
    return output
```

Since we are now predicting the cosine and sine of the angle (camera position) for our RMS error metric, we need to convert the cos and sine values to angle, for this, I made a small change to my rmserror function as below:

```
def rmserror(o1, o2, mode):
    if mode == 'train_and_test_2_out':
        o1 = o1.detach().numpy()
        o2 = o2.detach().numpy()
        o1 = np.apply_along_axis(lambda x: np.arctan2(x[0], x[1]), 0, o1)
        o2 = np.apply_along_axis(lambda x: np.arctan2(x[0], x[1]), 0, o2)
        error = o1-o2
        error_normalized = np.fmod(np.fmod(error+np.pi, 2*np.pi)+2*np.pi,
2*np.pi)-np.pi
        rmse = np.sqrt(np.mean(np.square(error_normalized)))
        return rmse
    else:
        error = o1-o2
        error_normalized = np.fmod(np.fmod(error+np.pi, 2*np.pi)+2*np.pi,
2*np.pi)-np.pi
        rmse = torch.sqrt(torch.mean(torch.square(error_normalized)))
        return rmse.item()
```

After making these changes I ran the code in train\_and\_test\_2\_out mode using below command:

```
py pose_regress.py --mode=train_and_test_2_out --n_epochs=100 --
eval_every_n_epochs=5 --h5_filename=data/pontiac_360.h5 --outdir=out --
split_name=cvd_split_every5 --viz_pose3d
```

The best value of RMSE error I could achieve with this is:

Num\_epoch = 70

Train set error = 0.097

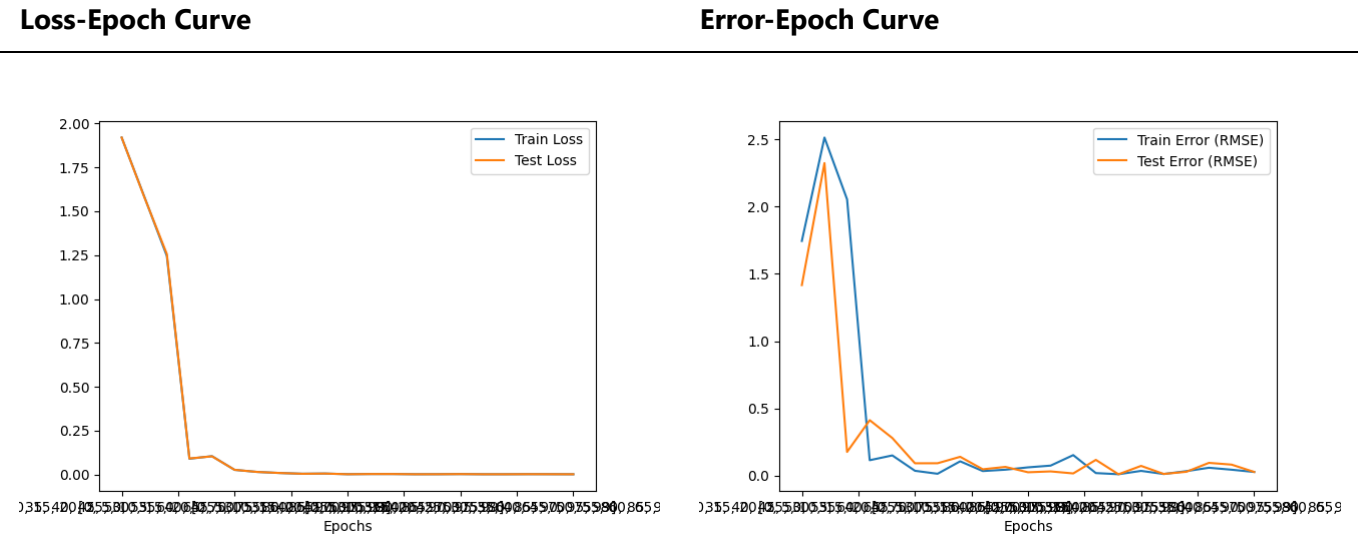
Test set error = 0.095

This is in radians, converting to degrees

Train set error = 5.55

Test set error = 5.44

Below are the Loss-Epoch curve and Error-Epoch curve



This is the best performance achieved so far in terms of the error metric.

Conclusion

I started this assignment with knowledge of camera projection and by the end of it, I am able to build a neural network that can predict the right camera angle with a pretty low error rate. Starting with basic CNN architecture I was able to achieve a good model which when analyzed qualitatively, has some really good 3-axis pattern projections. I then changed the model architecture a bit by changing kernel size of convolution filters and adding batch normalization. This improved the performance both quantitatively and qualitatively as can be seen in above observations. Further after, I changed the neural network model to output cos and sin of angle as two outputs instead of angle only. Running this made it even better and I was able to achieve error in the range of 0.097 radians on train set and 0.095 on test set which is nearly ~5.5 degrees.