1. Overview

In my project, I utilized a Kaggle notebook for efficient coding. The approach combined CNN and RNN methodologies to address the problem at hand effectively.

The CNN component began with three convolutional layers. The first layer applied a kernel size of 5 and padding of 2 to extract features from the data. The subsequent layer employed a kernel size of 5 and padding of 1 to further refine these features. Finally, a convolution operation with a kernel size of 2 and padding of 1 was applied to capture fine-grained details. The purpose of these convolutional layers was to identify important patterns and structures in the data.

Following the CNN layers, a fully connected layer was introduced to expand the extracted features and capture complex relationships between them. This layer enhanced the model's ability to learn higher-level abstractions from the data.

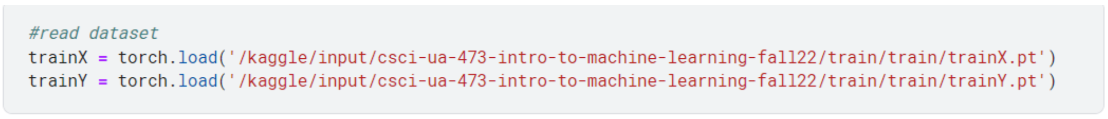
To incorporate temporal information, an LSTM layer was used. LSTMs exceled at capturing long-term dependencies and timing information in sequential data. By utilizing an LSTM layer, the model can effectively learn and understand temporal patterns within the data.

The project involved performing regression prediction for 12 consecutive points. To train the model, the Mean Squared Error (MSE) loss function was employed. MSE was commonly used for regression tasks as it quantified the average squared difference between predicted and actual values, providing a measure of the model's performance.

To optimize the model's parameters, stochastic gradient descent (SGD) was employed due to the large dataset. SGD randomly selected subsets of the data for each iteration, making it computationally efficient for training on large datasets.

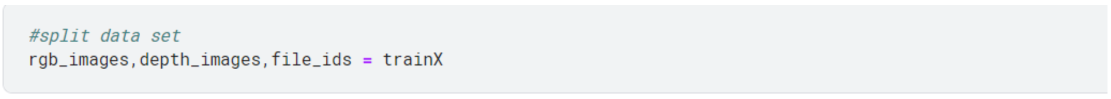
2. Method

The first thing I do was to read dataset and split dataset, in which I mainly used torch to read data. Since the file format is pt., we could use pytorch to read data.



```
#read dataset
trainX = torch.load('/kaggle/input/csci-ua-473-intro-to-machine-learning-fall22/train/train/trainX.pt')
trainY = torch.load('/kaggle/input/csci-ua-473-intro-to-machine-learning-fall22/train/train/trainY.pt')
```

Next, as the project instruction mentioned, the data was comprised of rgb_images, depth_images, file_ids. I parsed data into above three parts.



```
#split data set
rgb_images,depth_images,file_ids = trainX
```

The function showed the image data. Since the data was in rgb, I used a transpose function to convert it.

```python
#show three different view image
def show_img_three_view(rgb_image):
    for i in range(3):
        one_image = rgb_image[i].numpy()
        plt.imshow(one_image.transpose(2, 1, 0))
        plt.show()
```

The next step involved creating a user-defined dataset class. In order to train using Torch, we needed to utilize a DataLoader. However, the DataLoader can only accept data in the format of a subclass of the dataset. Hence, we needed to convert our data into this format.

```python
from torch.utils.data import Dataset, DataLoader
# from torchvision import transforms, utils
#define train data set
class Trainset(Dataset):
    def __init__(self, rgb_images,labels):


        self.rgb_images = rgb_images
        self.labels = labels

    def __getitem__(self, index):
        return self.rgb_images[index].float(),self.labels[index].float()


    def __len__(self):
        return len(self.labels)

    def len(self):
        return len(self.rgb_images)
```

The next thing to do was to define the DataLoader with a certain batch size.

```python
#define train loader
train_loader = Data.DataLoader(dataset=dataset, batch_size=64, shuffle=True)
```

Next, I defined the CRNN (Convolutional Recurrent Neural Network) architecture to handle the mixed data characteristics of continuous values and image features. I then implemented the MSELoss function for training and select appropriate hyperparameters such as learning rate and momentum. With a defined number of epochs, set to 10, I trained the data using random gradient descent and evaluate performance using RMSE (Root Mean Squared Error).

```
#train data
epoch = 30
for epoch in range(epoch):
    running_loss = 0.0
    for i, data in enumerate(train_loader, 0):
        inputs, labels = data
        if torch.cuda.is_available():
            inputs = inputs.to('cuda:0')
            labels = labels.to('cuda:0')
        # zero the parameter gradients
        optimizer.zero_grad()

        # forward + backward + optimize
        outputs = myNet(inputs)


        loss = criterion(outputs, labels)
        loss.backward()
        optimizer.step()

        running_loss += loss.item()

    print(f'epoch [{epoch + 1}] train loss: {running_loss / i:.5f}')
```

3. Interesting findings

I have observed that performing the shuffle operation when loading data results in more stable gradient descent during training. Without shuffling, the gradient descent tended to be unstable. Shuffling the data helped ensure that the model received a diverse range of samples during each iteration, leading to better convergence.

Additionally, I have noticed that even a slight difference in the learning rate of the optimizer function can have a significant impact on the RMSE (Root Mean Squared Error) output in the CSV results. While the default learning rate was often set to 0.001, I initially found that using a learning rate of 0.005 yielded lower RMSE results. It was important to experiment with different learning rates to identify the optimal value for improved performance. Furthermore, I've observed that using a higher learning rate reduces the training time required.

The number of epochs also played a crucial role in the training process. Increasing the number of epochs improved the overall score. The number of epochs indicated the number of passes the training algorithm makes over the entire dataset. By allowing the model to iterate through the data multiple times, it had more opportunities to learn and refine its predictions, leading to better performance.

4. Future Improvements

1. To increase the depth and capacity of the model, we can add more layers to the neural network. Deeper networks with additional hidden layers have a

greater ability to capture complex patterns and relationships in the data. However, it's important to monitor the risk of overfitting that arises with deeper models. To mitigate this, we can incorporate techniques like regularization and early stopping.

Moreover, incorporating an attention mechanism can further enhance the training process. Attention mechanisms help the model focus on important features or regions within the input data, resulting in improved accuracy and performance. By selectively attending to relevant information, the model can make more informed predictions.

Additionally, increasing the amount of training data can improve the accuracy of the model. More data provides a broader range of examples for the model to learn from, enabling it to generalize better to unseen data.

2. Instead of using the stochastic gradient descent (SGD) optimization algorithm for gradient descent, we can explore using the Adam optimization algorithm. Adam combines the advantages of both adaptive learning rates and momentum methods. It adapts the learning rate for each parameter based on their past gradients, resulting in efficient optimization and faster convergence. Adam has shown superior performance in many deep learning applications.

3. Introducing data augmentation techniques can help enhance the dataset. Data augmentation involves applying various transformations to the existing data, such as rotation, scaling, flipping, or adding noise. These transformations introduce new information to the dataset, effectively increasing its size and diversity. Data augmentation can help prevent overfitting, improve model generalization, and boost overall performance.