Project Report: Fingertips Position Estimation of a Robot Hand

Kayan Shih
NETID: ks5250

1 Background

1.1 Goal

In this project, the goal is to use supervised learning to predict the positions of the tips of a 4-fingered robot hand from RGBD images. The RGBD images contain both color (RGB) and depth information. The project involves implementing a dataset class that loads and augments the data, and a convolutional neural network (CNN) that takes the RGBD images as input and outputs the positions of the fingertips. The CNN will be trained using the provided dataset in Kaggle and will be evaluated using the root mean squared error(RMSE) loss. The project will be submitted to a Kaggle competition, where the model's performance will be evaluated on the RMSE value.

1.2 GitHub Link

The project's code is uploaded to GitHub Repository for reference.

2 Input and Output

2.1 Description of Data

The data used in the project is from the train and test datasets in Kaggle's Final Project Competition: Fingertips Position Estimation of a Robot Hand. The datasets are provided in the form of three files: trainX.pt, trainY.pt, and testX.pt. These files contain respectively the training images, the corresponding hand states for each image, and the test images that will be used for evaluation. The datasets are formatted in a way that can be used with PyTorch, a popular deep-learning framework.

2.2 Description of Input

2.2.1 File trainX.pt

File trainX.pt contains the images that will be used to train the model. It consists of three components: RGB images, depth images, and file IDs. The RGB images have dimensions (number of data samples, number of camera views, number of channels, height, width), and the depth images have dimensions (number of data samples, number of camera views, height, width). The file IDs contain the sample ID for each data sample. Figure 1 shows the example of RGB images from the dataset.

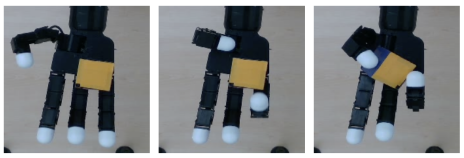

Figure 1: Example RGB images from the dataset

2.2.2 File trainY.pt

File trainY.pt contains the robot states (outputs) for training the model. The data is organized as a tensor with dimensions (number of data samples, 12), where 12 corresponds to the 3D coordinates (x, y, z) of the four fingertips. This tensor can be used with the training images in trainX.pt to train the model in a supervised manner.

2.2.3 File testX.pt

File testX.pt has a similar format to trainX.pt, with the same dimensions for the RGB and depth images and the same format for the file IDs. The testX.pt file can be used to evaluate the performance of the trained model on unseen data.

2.3 Description of Output

The output is a CSV file, which saves the model predictions on the file testX.pt. The file contains columns for the sample IDs and the 12 predicted positions of the robot's fingertips.

3 Methodology

3.1 Import Dataset

In this part, the code imports data from the Kaggle competition into Google Colab. It begins by mounting my Google Drive to the Colab environment, which allows me to access files in my Drive within Colab. Next, it installs the Kaggle Python package, which provides a command-line interface for interacting with the Kaggle API. Then, the code copies

a file called kaggle.json from Google Drive to the .kaggle directory in the Colab environment. This file contains my Kaggle API credentials, which are needed to authenticate my requests to the Kaggle API. After that, the code uses the kaggle command-line interface to download the zip file containing the data to the Colab environment. Finally, the code uses the unzip command to extract the contents of the zip file, which will create a new directory containing the data files.

3.2 Preprocess Data

3.2.1 Train_dataest Class

In this part, the code creates a custom dataset class called Train_dataest by extending the Dataset class from PyTorch. This allows the class to be used with the DataLoader class to load the data in batches and feed it to a model for training or evaluation. The Train_dataest class takes a number of parameters when it is initialized:

- root_path: The path to the root directory of the dataset.
- data_file: The name of the file that contains the data. This file should be a PyTorch tensor saved using torch.save. The default value is 'trainX.pt'.
- target_file: The name of the file that contains the targets. This file should also be a PyTorch tensor saved using torch.save. The default value is 'trainY.pt'.
- preprocess_rgb: A function that preprocesses the RGB images. This function should take a single image as input and return a preprocessed image. The default value is None.
- preprocess_depth: A function that preprocesses the depth images. This function should take a single image as input and return a preprocessed image. The default value is None.

In the __init__ method, the class loads the data and targets from the specified files using torch.load and saves the preprocessing functions.

The __getitem__ method is used to retrieve a single item from the dataset. It takes an index as input, retrieves the corresponding RGB and depth images from the data, applies the preprocessing functions, concatenates the preprocessed RGB and depth images into a single RGB-D image, and returns a tuple containing the RGB-D image and its target.

The __len__ method returns the length of the dataset, which is the number of images it contains.

3.2.2 Data Transformations

In this part, the code defines a set of transformation steps to be applied to the images in the training dataset. The specific transformations include a random resize and crop to a specific size (224x224 pixels), color jittering to augment random perturbations to the brightness, contrast, and hue of the images, and normalization to scale the pixel values to a range of 0 to 1.

The transforms.Compose() method allows these transformations to be applied sequentially to the input images, so that the images are first cropped, then jittered, and finally normalized. The resulting preprocessed images are then used to create an instance of the Train_dataset class, which is likely used to load and manage the images for training the model. This dataset instance can then be used with a PyTorch DataLoader to easily load and iterate over the images in the dataset during training.

3.2.3 Train-Test Split

In this part, the code splits a dataset into a training set and a test set and creates PyTorch DataLoaders for each set. It allows the model to be trained on a portion of the data and then tested on a separate, unseen portion of the data to evaluate its performance.

The code first calculates the size of the training and test sets by multiplying the total number of images in the dataset by the desired split proportions (70% for training and 30% for testing in this case). It then uses the random_split() method to split the dataset into two subsets, with the sizes determined by the previously calculated train and test sizes. This method ensures that the images are randomly assigned to either the training or test set.

The DataLoader() class is then used to create a data loader for each of the training and test datasets. The data loaders are configured with a batch size of 16, which specifies the number of images to be returned in each batch during training or testing. The data loaders are also set to shuffle the images before returning them, which can improve the model's performance during training by ensuring that the images are presented to the model randomly. The data loaders can be used to easily iterate over the images in the datasets during training and testing, and they automatically handle the process of loading and preprocessing the images as

needed. Once the data loaders have been created, the original dataset is deleted to free up memory.

3.3 Train Model

To build a convolutional neural network, the code loads a pre-trained ResNet-34 model using the models.resnet34() method. ResNet-34 model is a deep residual network that has been trained on a large dataset and has achieved state-of-the-art performance on many image recognition tasks. By setting pretrained=True, the code uses the pre-trained weights for the network, which helps the network to converge faster and often achieves better performance than training from scratch.

The code then modifies the input and output layers of the model to match the dimensions of the input and output data for the new dataset. The first convolutional layer of the model is modified to have 4 input channels instead of 3 to match the number of channels in the RGB and depth images. This allows the model to accept both the RGB and depth images as input, which are concatenated together into a 4-channel tensor before being passed to the model. The output layer of the model is also modified to have 12 output units to match the size of the output vector. This allows the model to predict the class of the input image based on its RGB and depth values.

After modifying the network architecture, the code defines a mean squared error(MSE) loss function using the torch.nn.MSELoss class and an Adam optimizer using the torch.optim.Adam class. The code then trains the network on a set of labeled training data using a for loop that iterates over a specified number of epochs. At each epoch, the code loops over the training data in mini-batches, computes the network's predictions and the MSE loss between the predictions and the ground truth labels, and uses the Adam optimizer to update the network's weights to reduce the loss. Adam is a stochastic gradient descent algorithm that uses an adaptive learning rate, which helps the network to converge faster and often achieves better performance than using a fixed learning rate. The code also periodically evaluates the network's performance by computing the average training loss and validation loss at each epoch and prints these values to monitor the progress of the training process.

3.4 Output Result

After training, the code uses the trained network to make predictions on a set of unlabeled test data. The predictions are made by looping over the test data in mini-batches, computing the network's predictions for each mini-batch, and appending the predictions to a list. The code then writes the list of predictions to a CSV file using the pandas library. This CSV file can be used to evaluate the performance of the trained network on the test data in Kaggle.

4 Experimental Results

The evaluation metric used in Kaggle is the root mean squared error(RMSE), which is a measure of the difference between values predicted by a model and the true values. It is calculated by taking the square root of the average of the squared differences between the predicted and true values. In general, a smaller RMSE indicates that the model is making predictions that are closer to the true values, and therefore is performing better.

When I trained the data, I tried different numbers of epochs, resulting in different RMSE values, which are shown in the table 1 below. I reached the lowest RMSE value of 0.00759 with 300 epochs. But there seemed to be an overfitting issue when I trained with 500 epochs, which led to a higher RMSE value.

Number of Epochs	RMSE Value
4	0.02557
100	0.01737
200	0.00861
300	0.00759
500	0.02414

Table 1: RMSE Value for different number of epochs

5 Discussion

During data preprocessing, the code uses the method __getitem__ to retrieve items from the dataset. It selects the middle image, img1, from each data sample without choosing the first or last images, img0 and img2. The reason of choosing one angle of the image for each data sample is because there is also depth information provided and the depth is the same for the three images for each data sample.

By selecting the middle image, the code ensures that it uses a representative sample of the data, rather than just using the first or last images. This can help to improve the performance of a model that uses RGB-D data. This is because it

reduces the amount of data that needs to be processed, which can make the training process more efficient and can potentially lead to faster training times and better model performance. In addition, using only one image can help to avoid redundancy in the data, which can improve the accuracy of the model and potentially lead to a lower root mean square error.

Besides, the code uses several data transformation methods including cropping the images to a fixed size, applying random color jitter to the RGB images, and normalizing the RGB images using the mean and standard deviation values for the ImageNet dataset.

The advantage of these transformations is that they can help to improve the performance of a model that uses RGB-D data. By cropping the images to a fixed size, the model can learn more effectively from the data, and the random color jitter can help to improve the robustness of the model. In addition, the normalization of the RGB images can help to improve the convergence of the model during training.

Moreover, the code uses the mean and standard deviation values of the ImageNet dataset from the ResNet-34 model to normalize the input images. These values are commonly used in preprocessing for deep learning models, and they can help to improve the performance of the model by making the data more consistent and easier for the model to learn from.

6 Future Work

The project could use hyperparameter tuning methods or libraries, such as grid search or Ray Tune, for hyperparameter optimization when training a pre-trained model. Grid search is a method of hyperparameter optimization that involves training a model using a range of different hyperparameter values, and then select the combination of hyperparameters that yields the best performance. Ray Tune is a hyperparameter optimization library that uses various algorithms to search for the optimal combination of hyperparameters for a model. In general, these methods can help to find the optimal combination of hyperparameters, such as batch size, the number of filters, and learning rate, for the model, which can improve its performance and potentially lead to a lower root mean square error.

In addition, the project could use a regularization technique, early stopping, which monitors the performance of a model on a validation set during training, and stops the training process when the performance on the validation set stops improving. This can help to prevent overfitting and can improve the generalization of the model. And it can be combined with other regularization methods, such as weight decay and dropout, to further improve the model's performance.