CSCI-UA 473: Intro to Machine Learning
Final Project Report

Fingertips Position Estimation of a Robot Hand

Boyuan Zhang (bz2058)

December 16, 2022

1 Introduction

1.1 Problem

This is a regression problem, where we are given the images of the hand of a robot from different angles, and we're asked to predict the coordinates of each of the fingertips of the robot. Since we're given the training data and labels, this is also a supervised learning problem.

1.2 Approach

In order to address this problem, we are going to implement a neural network. In specific, the neural network could transform the image data, and would then enable us to learn the representation from the transformations, which would be helpful and critical for us to predict the target that we're interested in.

1.3 Implementation

This project is implemented using Python programming language and within Google Colab environment. The majority of the works are conducted within Tensorflow framework, along with other libraries including Numpy, Pandas, Scikit-learn, and PyTorch. In order to make the implementation process more efficient, hardware accelerators, in specific, NVIDIA P100 and T4 GPUs, are used.

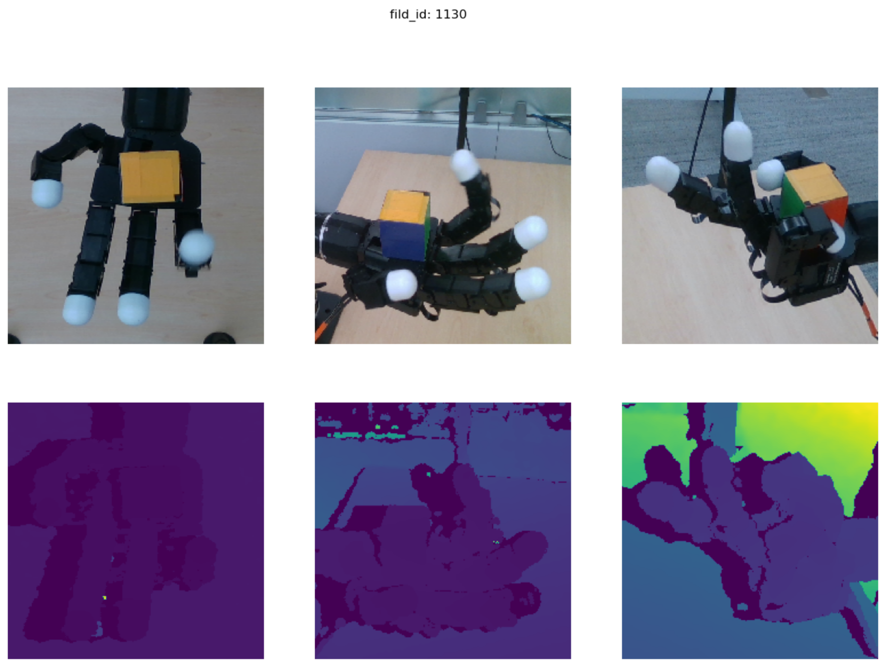
2 Data

The data we're given is the RGBD images of the hand of a robot from 3 different angles. In specific, each hand position(sample) is comprised of the following three properties:

- **RGB image**: has dimension *(num_ data_ samples, num_ camera_ views, num_ channels, height, width)*
- **Depth image**: has dimension *(num_ data_ samples, num_ camera_ views, height, width)*
- **File ID**: contains the sample ID

Since the robot hand has 4 fingers, the output of the model is expected to have the (x, y, z) coordinates of all 4 fingertips, which would end up with an output of shape *(num_ data_ samples, 12)* with columns corresponding to $(x_1, y_1, z_1, x_2, y_2, z_2, x_3, y_3, z_3, x_4, y_4, z_4)$ and each row corresponding to a sample uniquely identified by *file_id*.

Specifically, there are 3396 samples in the training data we're given. So we have RGB images with dimensions (3396, 3, 3, 224, 224) and depth images with dimensions (3396, 3, 224, 224). Below provides an example displaying a sample in the training dataset with its file_id, rgb_image, and depth_image along with its training labels.



$(x_1, y_1, z_1) = (0.0548, 0.0530, 0.1185)$ $(x_2, y_2, z_2) = (0.0712, -0.0024, 0.1091)$

$(x_3, y_3, z_3) = (0.0993, -0.0497, 0.0397)$ $(x_4, y_4, z_4) = (0.0497, 0.0948, -0.0077)$

3 Method

The main methodology of this project is to implement transfer learning technique, as the training dataset we're given is still relatively small in sample size. By utilizing a pre-trained model and then fine-tuning on our training dataset, we could have a better chance to achieve a better prediction performance while maintaining the stability of our model.

With this being said, the majority of the work of this project is around the following criteria:

- **Data Preprocessing**: Preprocess the training data so that could fit the specific dimension requirement of the base(pre-trained) model of our choice;
- **Model Training**: choice of base(pre-trained) model; more architectures to be built on top of the base model to cater to our specific prediction task;
- **Model Evaluation**: choice of evaluation metrics specific to our prediction task;

- **Hyperparameter Optimization**: choice of other hyperparameters to fine-tune the base model in order to achieve better training and generalization performance.

4 Preprocessing

Data are loaded and preprocessed to fit the neural network model by using scaling, transformation, splitting, and normalization techniques.

4.1 Data Loading

Data are downloaded directly from Kaggle API. Load in the training images and training labels file in `.pt` format provided using `torch.load()` function from the `torch` package. For training images, further split into *RGB_ images*, *depth images*, and *file_ ids* for each sample.

4.2 Data Scaling

Appropriate scaling is applied to training images and labels. In specific, *RGB_ images* is first scaled by 255, as each channel is bounded between 0 and 255. Similarly, though *depth_ images* do not really have an upper bound, we choose to scale by 1000. After this scaling, we're ensuring each channel in both *RGB_ images* and *depth_ images* is a fraction between 0 and 1.

For training labels, notice that the coordinates in the labels are given in meters units, but they really differ in the millimeter unit. Thus, in order to magnify the loss in the regressions during the training and to converge faster, we multiply the labels by 1000 to convert them into millimeter scale.

4.3 Data Transformation

After scaling, we manage to combine the *RGB_ images* and *depth_ images* together for each sample, by adding *depth_ images* as an extra channel into the *RGB_ images*, so that the training images now have 4 *channels*. Furthermore, for simplicity and in order to reduce the dimension in training data, we combined the 3 different *camera_ views* with 4 *channels*. Thus, the training images end up with a dimension *(num_ data_ samples, height, width, num_ camera_ views*$\times$*num_ channels)*, which is (3396, 224, 224, 12).

4.4 Data Splitting

Training images and labels are split into training set and validation set with a ratio of 70% and 30% using `train_test_split` function in `sklearn` package.

4.5 Normalization

Using `Normalization` layer from the `Keras` package in Tensorflow framework, normalization is fitted only on training images, and performed on both training and validation images. After normalization, the distribution of both training and validation images are shifted from mean 0.57 and standard deviation 0.48 to roughly mean 0 and standard deviation 1.

5 Model

The final model architecture is built with a pre-trained model as a base model together with some additional architectures on top of the base model.

5.1 Base Model

Pre-trained model available online is used to initialize our neural network as a base model. In specific, we have considered and experimented with the popular VGG and ResNet models. After experimental evaluation, we end up using ResNet50 model along with pre-trained weights on ImageNet as our base model.

Specifically, in order to cater to our regression task, the fully-connected layer at the top of ResNet50 is excluded when initializing the base model.

Since ResNet50 model requires exactly 3 input channels, we've changed the configuration of the input shape of the first input layer in our base model to 12 channels in order to fit our training data.

5.2 Additional Architecture

In order to generate a prediction of 12 coordinates, a fully-connected output layer with 12 nodes is built on top of our base model. Since we're working on a regression task, no(linear) activations are needed for the output layer.

Moreover, other layer architecture and techniques have been experimented as well. In specific, in order to prevent overfitting and increase generalization performance, Batch Normalization layer and Dropout layer have been experimented. However, they did not increase the performance during validation, so none of them are included in our final model.

6 Training

6.1 Data Preparation

Both training and validation images and labels are combined to tensor accordingly using the `Dataset` module in Tensorflow framework. Training dataset is shuffled with a buffer size of 1000, and both training and validation datasets are batched according to batch size.

6.2 Optimizer

Different optimizers are considered and experimented. Specifically, we experimented with SGD, SGD with momentum, and Adam. Eventually, we choose to use Adam as the optimizer, with an initial learning rate of 0.001.

6.3 Learning Rate

Learning rate step decay techniques are utilized to better converge. In specific, we set the initial learning rate to be 0.001, and decay exponentially with a parameter $\gamma = 0.1$ after a step size of 13 epochs. Since we set the total epochs for training to 52, the learning rate would decay exactly 3 times. Thus eventually we have a learning rate $1e^{-3}, 1e^{-4}, 1e^{-5}, 1e^{-6}$ accordingly for every 13 epochs.

7 Evaluation

Since we are doing a regression task, the loss is defined and calculated by the mean squared error between the label and the prediction.

By using the cross-validation technique, we train the model on training dataset and evaluate the performance of the model on the validation dataset. In order to align with the final evaluation on the Kaggle platform, we choose the evaluation metric to be the root mean square error.

8 Hyperparameter Optimization

Hyperparameter optimization is an important part of our fine-tuning approach. In order to better convergence, grid search technique is performed over the batch size, initial learning rate, and step size in learning rate step decay. Specifically, the batch size is experimented on over a range from 16 to 128, and the initial learning rate has experimented over a range from 0.1 to $1e^{-5}$. Eventually, we find that a batch size of 32 together with an initial learning rate of 0.001 gives the best validation performance.

9 Results

With all the methods and techniques mentioned above being implemented, and after careful experiments, we end up with a training loss of 0.7796, training RMSE of 0.883, validation loss of 7.8494, and validation RMSE of 2.8017 after all 52 epochs of training.

In order to strengthen the final model with more data, once we determined with all the hyperparameters within the model, we retrained the final model with both training and validation datasets together. We end up reaching a retraining loss of 1.1185 and a retraining RMSE of 1.0576.

10 Discussion

Throughout this final project, we have practiced the pipeline of building neural network models to solve a practical problem. In specific, we exercised various methods and techniques in data preprocessing, model building and training, and fine-tuning.

Moreover, throughout the tuning process, we noticed that the performance could increase a lot by using a relatively smaller batch size, which might be because of the consistency of the distribution of the image data in the training data of this task.

Besides, we notice that because of the relatively small difference between the prediction and the label under the original unit, evaluation and the converges of the model would be extremely hard if we did not convert the labels into a different unit to magnify the difference during data preprocessing. Also, we observed the problem of vanishing gradients during the model training, which again highlights the importance of data preparation, model structure, and choice of hyperparameters.

11 Future Work

For data preprocessing, noticed that some of the training images given include views of noisy backgrounds toward the boundary of the image or far away from the robot's hand. Thus, it might be useful to consider cropping techniques so that the model could better focus on the hand and not be distracted by the noise. Besides, augmentation techniques might be useful to enhance the generalization performance of the model. However, as our task is to predict the coordinates of the fingertips of the robot, it does not make sense to rotate or flip the images. In the future, it might be a good idea to explore other data augmentation techniques to increase the sample size. Another approach to increase the sample size and might be worth considering in this scenario is weak supervised or semi-supervised learning techniques. But it might not be very useful to deploy on this relatively simple task.

In terms of the model architecture, here we've only experimented VGG16/VGG19/ResNet50 as a base model and added a fully-connected output layer on top of it. It might be worth considering trying out other more complicated architectures and deeper networks as the base model, so that the model would have more capacity to learn more structures from the data. Besides the base model, it might also be a good idea to add more additional layers on top of the base model that is more cater to the specific characteristics of this task. Building the network from scratch might also be an option. However, I personally do not believe that it would benefit too much in terms of performance given the time consumption.

Lastly, since this is a relatively simple and straightforward task, it might be a good idea to consider implementing the AutoML framework, which would make the model selection and hyperparameter tuning process a lot easier compared to doing them manually. I tried to implement H2O and Auto Pytorch framework when I started on this project, but unfortunately, neither of them works because of the dimension problems of the input data, I guess. Thus, if we could fit the data into AutoML framework, there might be a chance to elevate the performance overall, especially considering that AutoML framework enables ensemble networks which might construct a more effective network architecture for this task.

Overall, as long as there's a loss, there would be infinitely many possible solutions to consider to keep minimizing the loss. But there would always be a trade-off between the model performance and the time consumed in building and optimizing the model. Thus, it might be a good idea to find a balance between model performance and time constraints depending on the practical usage of the model.

12 References

- Aakanksha, I. Güzey, R. Anant, and S. Haldar. (2022). CSCI-UA. 473 Intro to Machine Learning, Fall 2022. https://kaggle.com/competitions/csci-ua-473-intro-to-machine-learning-fall22
- François Chollet, et al. (2015). Keras. https://keras.io
- L. Pinto. (2022). CSCI-UA 473 Introduction to Machine Learning, Fall 2022. Campuswire. https://campuswire.com/c/G6C251796
- P. Dube. (2022). DS-UA 301 Advanced Topics in Data Science: Advanced Techniques in ML and Deep Learning, Fall 2022. NYU Brightspace. https://brightspace.nyu.edu/d2l/home/219804