CS-473 Capstone Project Report

Grace Yang, gy654, N10610063

December 2022

1 Problem Statement

The capstone project works on state prediction of a 4-fingered robot hand given RGBD (RGB + Despth) images. I implement a supervised learning algorithm that outputs the spacial positions of the tip of each finger given depth images and RGB images of the robotic hand from the top view, left view, and right view as input.

2 Method

2.1 Load and Preprocess Data sets

To prevent the RAM limitation issue during training, I create a "LazyLoadDataset" class where I load the data, apply necessary augmentations and return a tensor at each iteration of my training procedure.

Each sample in the dataset is associated with three RGB images and three depth images from the top, left, and right view respectively. I convert images to tensors, and the dimension of an RGB tensor is $3 \times 224 \times 224$. Three depth images are contained in a single depth file with loaded shape as $3 \times 224 \times 224$. Given that the RGB tensor and depth tensor have different ranges in values, I further standardize tensors using mean and standard deviation calculated from the train set.

To fully exploit spacial information contained in different views for better prediction outcomes, for each camera view, I append the depth image ($1 \times 224 \times 224$) to its corresponding RGB image along the first axis, and the concatenated result is a $4 \times 224 \times 224$ tensor, which will be loaded as the input to CNN. True labels (12 coordinates) are loaded when creating the train dataset, and the labels are multiplied by a scalar of 1000 to increase the loss thus encouraging the model to learn. Field_ids are loaded when creating the test dataset for inference purpose. Dataloaders are created from instances of "LazyLoadDataset" so that the training can proceed with batches. Also, dataset is shuffled when loaded using a dataloader. Each dataloader iteration returns a tensor of shape $batchsize \times 3 \times 4 \times 224 \times 224$.

The only data augmentation procedure I perform is to transform images into gray scale images, and the result comparing the model's performance with and without augmentation can be found in section 3. I omit other popular augmentation techniques such as rotation, flipping, cropping, and changing the brightness of the image since they do not seem to apply to this setting and would only confuse the model. There does not seem to be a distribution shift between the train and test set, as the camera angles, camera's color rendering are fixed.

I split the entire dataset given into a train and a validation dataset. The later will be used to select the best model during training judging from it's performance on the validation set. 10% of the dataset is reserved at the very beginning for model validation.

2.2 Model Architecture

The model I select is a Resnet50 network fine-tuned on the specific task of coordinate prediction. The first convolution layer has an input channel of 4 if using RGB images otherwise 2 using graysacle images (3 channels for RGB image and 1 channel for depth image or 1 channel for grayscale image and 1 channel for depth image). To use the pre-trained resnet50 [2] model, I modify the first 2d convolution layer of the resnet50 model so that it's architecture is aligned with input channel shape, which equals either 2 or 4 depending on the color option. I also modify the fully connected layers by adding a dropout layer and two linear layers. The ultimate model has 23.6M trainable parameters.

Given input of each batch, I separate out tensors along its second axis(3 views) and concatenate 3 tensors of shape $batchsize \times 4 \times 224 \times 224$ along the first axis. The output after simple processing is passed into the network. The output of which is split into three chunks and concatenated along axis 1, which is then fed to a linear layer with input shape 12*3 and output shape 12. I manually set the bias of the last linear layer to be the mean of finger tip coordinates in the train set. Please refer to subsection 4.2 for a rationale behind this.

2.3 Model Training

I train the network using Pytorch Lightning [1], which abstracts away the training loop under its hood. I create a customized pytorch lightning model inheriting "pl.LightningModule", where I specify the CNN network's architecture, its forward function, training and validation steps and optimizers (use Adam optimizer in the project). As the training proceed, the model is validated on the validation dataset at the end of each epoch and its performance is logged. It the end, the model with the best performance on validation set is saved for test inference.

In terms of training time, GPU provided by the google colab platform accelerates training. Each epoch takes approximately 2 minutes to train when batch size equals 16. I use 'Weights and Bias" platform to supervise the training. The logging and visualizations of training and validation loss provide me with insights to further modify the model architecture and training parameters.

2.4 Model Inference

I use the best performing model on the validation dataset with respect to root mean squared error loss that I obtained in the previous step to make inferences on the test dataset. A test dataloader is created from the test set with a much larger batch size to reduce the inference time. Each batch of test samples are passed into the network which outputs a batch of predictions of 12 coordinates for each sample. Note that the raw predictions are divided by 1000 as the final output since we initially scale the ground truth by 1000 to facilitate training. The predictions, together with their corresponding Field_id, are saved in "submission.csv" file for final evaluation on the instructor's end.

3 Experimental Results

3.1 Metric

The evaluation metric for the project is the root mean squared error loss. The formula is:

$$RMSE = \sqrt{\sum_{i=1}^{n} \frac{(\hat{y}_i - y_i)^2}{n}}$$

3.2 Model's Performance Evaluation

Recall that I split the train and validation set at the very beginning and there is no leakage between the two, with the validation set making up 10% of the total samples in the entire dataset. subsection 2.1

If I preserve color information during preprocessing, the best performing model selected based on its performance on validation dataset achieves a root mean squared error loss of 0.00399 on the validation set. The root mean squared error loss on the validation set reduces from 64 to 0.00399 after 32 epochs of training. Though the training loss continues to decrease after 32 epochs, the model's validation loss start to increase, suggesting an over-fitting trend. After submitted to the kaggle and tested on 60% of the test samples, the best performing model's RMSE is 0.00372.

I also experiment with training the network using gray scale images. The best performing model under this setting achieves a validation loss of 0.00337 and a test loss of 0.00351 when submitted to Kaggle. The final result is slightly better than training with RGB images. Therefore, color information does not help predictions in this case.

4 Discussion

4.1 Baseline Model

Apart from training a model fine-tuned on Resnet50, I also train a Convoluted Neuron-network followed by fully connected layers as a baseline model for comparison. The CNN model's architecture takes inspiration from the VGG architecture. [4]

I tailor the parameters of the model to better fit the problem at hand. The model has 857 K trainable parameters and contains the following layer in a sequential order:
Dropout2d(0.2)
Conv2d(4, 6, kernel_size = 3, padding = 1)
ReLu()
MaxPool2d(kernel_size = 2, stride = 2)
Dropout2d(0.2)
Conv2d(6, 12, kernel_size = 3, padding = 1)
ReLu()
MaxPool2d(kernel_size = 2, stride = 2)
Dropout2d(0.2)
Conv2d(12, 24, kernel_size = 3, padding = 1)
ReLu()
MaxPool2d(kernel_size = 2, stride = 2)
Dropout2d(0.2)
Conv2d(24, 48, kernel_size = 3, padding = 1)

ReLu()
MaxPool2d(kernel_size = 2, stride = 2)
Dropout2d(0.2)
Conv2d(48, 48, kernel_size = 3, padding = 1)
ReLu()
MaxPool2d(kernel_size = 2, stride = 2)
Dropout2d(0.2)
Conv2d(48, 32, kernel_size = 3, padding = 1)
Flatten()
Linear(1568, 512, bias = False)
Dropout(0.2)
Sigmoid()
Linear(512, 12, bias = False)

The first convolution layer has an input channel of either 4 or 2 due to the reasoning above. Each 2dConvolution layer is followed by a 2dMaxPool layer that reduce the image to half of its size, and a dropout layer that prevents over-fitting. The output of which passes the Relu activation function for nonlinear feature transformation. The number of filters increase as the network gets deeper. The output of the last convolution layer is converted back to three chunks each corresponding to a view. The concatenation of three chunks are flattened to a long vector of shape 1568 and passed to sigmoid activation and finally a linear layer to generate 12 predicting values, each representing a coordinate of the robot finger tip. Note that I fix the bias of the last linear layer to be the mean of labels. Otherwise, the model is led astray by the bias. It would be trapped at a local minima and simply predicting the mean of images. Bias is fixed to encourage the model to focus on learning other important features.

4.2 Problems Encountered and Solution

After training the CNN, I observe that the inference outputs on the test set has extremely low variation across samples. It seems to be trapped at a local minima and simply predicting the mean of images. The training loss is stuck at approximately 0.0023. I hypothesize that this is due to the bias in the last fully connected layer. To prevent the model from simply learning the mean, I fix the bias of the last linear layer to be the mean of labels so that the model would not be led astray by the bias. After manually fixing the bias, I observe that the training loss decrease to as low as 0.006, and the standard deviation on test inferences has similar distribution with the standard deviation distribution of coordinates in the training set.

4.3 Model Comparison

The baseline CNN model I build from scratch achieves a validation RMSE of 0.00664, which is higher than 0.00372 of the model fine-tuned on Resnet50.

One explanation of the better performance of Pretrained Resnet50 is that compared with the CNN model trained from scratch, pretrained models provide better weight initialization having been trained on very large datasets. In the future, I will also try out googlenet and alexnet architecture and compare their performances. Even with the model I used, I can optimize its performance by adjusting its parameters such as filter size in each layer, dropout rate, and activation function. However, given that the baseline model has 2.5M parameters, approximately 10% of the size of the fine-tuned model, its performance is beyond expectation.

I also experiment with transforming RGB images into gray scale images before passing them into the baseline CNN model with depth images. The optimal RMSE achieved on the validation set is 0.00698, slightly higher than the case using RGB images, suggesting that color is marginally helpful to predictions in this case.

5 Future Work

5.1 Data Loading

Though the current data loading method addresses the RAM limitation issue, we can improve its efficiency in reading data. Currently, CPU is heavily utilized as 5 files are read for each samples while GPU is not fully utilized. In the future, we could store data in hdf5 format, which would achieve both targets of lazy loading and its acceleration since the data for all samples are stored in a single file.

5.2 Optimize Model Architecture, Use Pretrained Model

In this project, I build a model fine-tuned on Resnet50 and compare it with my own CNN network built from scratch. However, the selected pretrained model it is not necessarily the best model option. In the future, I will test out other pretrained models such as ResNet18 and googlenet, etc provided by torchvision and fine-tune them on the robot hand dataset.

5.3 Hyperparameter Tuning

Hyperparamter searching is direction to further improve the model. The set of hyperparameters we could search for include leaning rate, batch size, number of filters in each convolution layer, dropout rate in each dropout layer, type of activation function etc. In this project, I manually test out several reasonable combinations of parameters. We can also use grid search, Bayesian search or Hyperband [3] to automate the searching process within a predefined search space.

References

[1] William Falcon et al. Pytorch lightning. *GitHub. Note: https://github.com/PyTorchLightning/pytorch-lightning*, 3, 2019.

[2] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition.

[3] Lisha Li, Kevin Jamieson, Giulia DeSalvo, Afshin Rostamizadeh, and Ameet Talwalkar. Hyperband: A novel bandit-based approach to hyperparameter optimization.

[4] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition.