Capstone Project

Haoyu Yang (hy2122)

1. Method

In this project, I will predict the positions of the tip of each finger based on the CNN model trained on the robotic hand from three camera views. The RGB-D dataset consists of 3396 training images (with depth image respectively) and 849 test images.

In the first part, I load the dataset to Colab by lazy loading. Lazy loading takes the path of the dataset and outputs an object for data storage. The train lazy loader stores img0, img1, img2, depth, field_id, and Y (six parameters in total), while the test loader stores img0, img1, img2, depth, and field_id (five parameters in total). In lazy loading, I apply data normalization by transforms (figure 1). The transforms from torchvision for RGB images (both the training and test sets) contain CenterCrop(), toTensor(), and Normalize(). For depth images, I first divide them by 1000 to limit their ranges and apply a max-and-min normalization on them.

```
[ ]  # import the training set
     trans = transforms.Compose([transforms.ToPILImage(),
                                 transforms.CenterCrop(224),
                                 transforms.ToTensor(),
                                 transforms.Normalize((0.485, 0.456, 0.406), (0.229, 0.224, 0.225))
                                ])
     dataset = LazyLoadDataset('./', train = True, transform = trans)
```

(figure 1)

The CNN model I am going to use is VGG16 from tensorflow. To fit the model, I first convert data in the lazy data loader to numpy arrays. I use loops and store the array in the .npy files for fast access since the Colab often disconnects (figure 2). Besides, I transpose the RGB images set to size (num_samples, 224, 224, 3) to fulfill the input size requirement of VGG16. For the Depth images set, likewise, I stack three identical images (from the same camera view) with size (num_samples, 224, 224, 1) to size (num_samples, 224, 224, 3) to fit in the input of the model. At the same time, I split the training set to training set (80%) and validation set (20%) to better evaluate the model.

```
train_img0 = np.zeros([2716, 3, 224, 224])
train_img1 = np.zeros([2716, 3, 224, 224])
train_img2 = np.zeros([2716, 3, 224, 224])
train_depth = np.zeros([2716, 3, 224, 224])
train_field_id = np.zeros([2716, 1])
train_y = np.zeros([2716, 12])

for i in range(2716):
  (img0, img1, img2, depth, field_id), y = dataset[i]
  train_img0[i] = img0.numpy()
  train_img1[i] = img1.numpy()
  train_img2[i] = img2.numpy()
  train_depth[i] = depth
  train_field_id[i] = field_id
  train_y[i] = y
```

(figure 2)

The CNN training process in my project consists of three parts:

1. Training for RGB/Depth images (6 models)
2. Fusion network training for RGB-D features (3 models)
3. Fusion network training for three camera views (1 model)

For training RGB/Depth images, I fine-tune the pretrained VGG16 model with input size (num_samples, 224, 224, 3) in which 3 are RGB channels for RGB images and the stacked channels for Depth images. The architecture of VGG16 is comparably simple with less trainable parameters so that I do not have to worry too much about the vanishing gradients and dead neurons. I exclude the top layer of VGG16 and replace it with a Dense layer with 12 outputs since we have 12 features to be predicted (figure 3). For finetuning, I freeze all trainable layers except the last one to adjust our model. To compile the model, since I am training CNN for regression, I select Mean Squared Error for loss function and RMSprop for the optimizer (it divides the gradient by the root of the average). Also, 32 and 20 are respectively selected as the batch size and the number of epochs and shuffling is enabled while fitting (since I do not use the data loader). The fine-tuning process is the same across img0, img1, img2, depth(1st camera view), depth(2nd camera view), and depth(3rd camera view) training.

```
train_img0 = np.zeros([2716, 3, 224, 224])
train_img1 = np.zeros([2716, 3, 224, 224])
train_img2 = np.zeros([2716, 3, 224, 224])
train_depth = np.zeros([2716, 3, 224, 224])
train_field_id = np.zeros([2716, 1])
train_y = np.zeros([2716, 12])

for i in range(2716):
  (img0, img1, img2, depth, field_id), y = dataset[i]
  train_img0[i] = img0.numpy()
  train_img1[i] = img1.numpy()
  train_img2[i] = img2.numpy()
  train_depth[i] = depth
  train_field_id[i] = field_id
  train_y[i] = y
```

(figure 3)

In the second and the third part of CNN (fusion network for RGB-D and fusion network for three camera views), I apply an identical simple model which only contains a Dense layer with 12 outputs. I do not include an activation function for regression prediction. The model respectively takes size (num_samples, 2 * 12) (RGB features and Depth features) and size (num_samples, 3 * 12) (features from three camera views) as input and outputs with size (num_samples, 12) (figure 4). Compilation parameters are still the same as above.

```
train_img0 = np.zeros([2716, 3, 224, 224])
train_img1 = np.zeros([2716, 3, 224, 224])
train_img2 = np.zeros([2716, 3, 224, 224])
train_depth = np.zeros([2716, 3, 224, 224])
train_field_id = np.zeros([2716, 1])
train_y = np.zeros([2716, 12])

for i in range(2716):
  (img0, img1, img2, depth, field_id), y = dataset[i]
  train_img0[i] = img0.numpy()
  train_img1[i] = img1.numpy()
  train_img2[i] = img2.numpy()
  train_depth[i] = depth
  train_field_id[i] = field_id
  train_y[i] = y
```

(figure 4)

Noticeably, I multiply the ground truth by 1000 via training. Since the given ground truth is estimated in meters, transforming it to millimeters encourages the model to converge faster and give a better accuracy. Still, the predicted output should divide 1000.

2. Experimental Results

```
history = model.fit(rgbd_train, train_y * 1000, epochs = 20, validation_split = 0.1, verbose = 1, shuffle = True)

Epoch 1/20
77/77 [==============================] - 0s 4ms/step - loss: 18.2293 - accuracy: 0.8699 - val_loss: 17.1912 - val_accuracy: 0.8713
Epoch 2/20
77/77 [==============================] - 0s 3ms/step - loss: 17.6273 - accuracy: 0.8658 - val_loss: 16.8853 - val_accuracy: 0.8529
Epoch 3/20
77/77 [==============================] - 0s 3ms/step - loss: 17.1612 - accuracy: 0.8719 - val_loss: 16.7137 - val_accuracy: 0.8824
Epoch 4/20
77/77 [==============================] - 0s 3ms/step - loss: 16.6931 - accuracy: 0.8674 - val_loss: 15.6532 - val_accuracy: 0.8713
Epoch 5/20
77/77 [==============================] - 0s 3ms/step - loss: 16.2353 - accuracy: 0.8642 - val_loss: 15.5057 - val_accuracy: 0.8750
Epoch 6/20
77/77 [==============================] - 0s 3ms/step - loss: 15.8877 - accuracy: 0.8674 - val_loss: 14.5072 - val_accuracy: 0.8824
Epoch 7/20
77/77 [==============================] - 0s 3ms/step - loss: 15.6862 - accuracy: 0.8666 - val_loss: 16.0785 - val_accuracy: 0.8750
Epoch 8/20
77/77 [==============================] - 0s 3ms/step - loss: 15.3221 - accuracy: 0.8646 - val_loss: 17.9866 - val_accuracy: 0.8566
Epoch 9/20
77/77 [==============================] - 0s 3ms/step - loss: 14.9779 - accuracy: 0.8740 - val_loss: 13.1175 - val_accuracy: 0.8934
Epoch 10/20
77/77 [==============================] - 0s 3ms/step - loss: 14.7804 - accuracy: 0.8650 - val_loss: 14.2149 - val_accuracy: 0.8750
```

(figure 5)

The training set accuracy and validation set accuracy in the final fusion network (camera views fusion network) is around 86%. Also, from the RMSE result shown on Kaggle (0.00437), there is still a way to go to improve the model accuracy. Generally, the performance of the model is not too bad.

3. Discussion

The accuracy for the VGG16 pretrained model for RGB images and Depth images is not too high (around 80% and 76% respectively), indicating that VGG16 may not be the best model for this training set. Furthermore, the architecture of the added top layer, the number of layers frozen, compilation parameters, the batch size selected, and even the normalization still have an impact on the accuracy, so do the fusion networks.

Additionally, the runtime of the model is large. When RGB images and Depth images are trained separately, the runtime is doubled. Probably a better model to accept RGB images and Depth images at the same time would be better.

4. Future Work

In the future, more pretrained models can be attempted for RGB images and Depth images training, including Resnet-18 and Resnet-50. Moreover, hyperparameter optimization can be performed on the batch size and learning rate chosen, including grid search and hyperband.

5. Reference

https://www.kaggle.com/code/viratkothari/image-classification-of-mnist-using-vgg16#3.-Preparing-Data