Introduction to Machine Learning (CSCI-UA.473): Final Project Report

Jason Zhang

December 14, 2022

Github Link

Methodology

- Mention the libraries used in your solution.
  - NumPy manipulate the data.
  - cv2 is used to process the rgb images and turn them into NumPy arrays and PyTorch Tensors.
  - From PyTorch, Datasets and Dataloader were imported to create and load a custom dataset (lazy loaded rgb images).
  - From PyTorch, the nn module was loaded as a backbone for designing neural networks.
  - PyTorch.transforms was used to apply image augmentations.
  - PyTorch.models was imported to utilize the pre-trained cnn models, specifically testing the accuracy of different ResNets.
  - From PyTorch.optim, Adam and SGD were the two optimizers imported and compared.
  - MatPlotLib.pyplot was used to visualize the effects of different image augmentations
- Explain the details of the methodology used to solve the homework - how did you read the data, scaler/normalizer used for the data, data split, etc.
  - The data was read using the cv2 library and stored in a custom Dataset class. Using the train files (containing 3 rgb images and 2 numpys array with the depth and target), the Dataset class would process the rgb images, depth, and target and store it as X and Y. Since it was lazy loaded, train and test files would only be loaded into memory when required to save memory and improve overall performance.

- Next, the normalization statistics were calculated. I decided to use 0 mean unit variance normalization and therefore needed to calculate the mean and variance of each rgb image and depth image.
- Next step was creating a loss function. Since the Kaggle leader boards were calculated using RMSE loss and since PyTorch does not have an in-built RMSE loss function, I utilized the MSE loss function and square root the result to create a custom RMSE loss.
- To create a train and test dataset, the "Train" folder was split using torch.utils.data.random_split, which would randomly split the data into train and test dataset. I used 80/20 test train splits for my project to help check for overfitting.
- For the optimizer and parameters, I primarily evaluated optim.Adam() and optim.SGD(). In the experimation section, I have a table and a quick discussion about the results.
- Next was defining the CNN used for this project. There were 2 primary CNN models discussed in the experimentation section. Both involved using the ResNet models in combination with a custom fully connected network.
- The train function I used was relatively standard (in line with the train function given in HW5).
    - Call optimizer.zero_grad() to zero all gradients to prevent gradients from summing together.
    - Apply transformations on images (normalization and other augmentations).
    - Send images to device (moving data to cpu, gpu, or tpu depending on the platform).
    - Plug img0, img1, img2, and depth into the model.
    - Next, I multiply the ground truth and output by 1000x to help convergence.
    - Then loss.backward() and optimizer.
- The test function was relatively standard as well, the model would be turned into evaluation mode and would print loss on the test dataset to check for overfitting.
- Finally, using the code given to us for project submission, the model was used on the true test data, using the X data to create Y outputs for submission.

Experimentation

The two sections for experimentation were evaluating the performance of different fully connected network architectures and the performance of different optimizers with different parameters.

Architecture

When exploring architecture, the primary concern was how altering the fully connected layers would affect the accuracy of my model. The inspiration behind these architectures are found in the research paper by Hengkai Guo, et al [1] which describes different ways that we can construct convolutional neural networks for finger tip recognition.

The table is concerned with adjusting the fcn, using ResNet18, optim.Adam() with learning rate of 10^{-4}. One of the architectures had the fully connected layers of each ResNet separate, concatenating after each ResNet produces a (batch_size, 12) tensor, where as another architecture which concatenates after the convolutional layers and has the fully connected components of all ResNets intertwined, listed below.

- Architecture 1 : ResNet18.fc = nn.Sequential(nn.Linear(512, 100), nn.ReLU(), nn.Linear(100, 12), nn.ReLU())
  self.fc_layers = nn.Sequential(nn.Linear(12 * 4, 12))
- Architecture 2: ResNet18.fc = nn.Identity()
  self.fc_layers = nn.Sequential(nn.Linear(512 * 4, 500), nn.ReLU(), nn.Dropout(0.5), nn.Linear(500, 500), nn.ReLU(), nn.Linear(500, 12))

Architecture	Epoch	1	2	3	4
Architecture 1	Train Loss	0.0132	0.0099	0.0070	0.0069
	Test Loss	0.0278	0.0124	0.0079	0.0075
Architecture 2	Train Loss	0.0234	0.0142	0.0120	0.0099
	Test Loss	0.0247	0.0157	0.0101	0.0103

Hyper-parameter optimization

The first problem was selecting a suitable optimizer for my CNN. I experimented with the two most common optimizers, SGD and Adam. The table is used to explore loss when adjusting learning rate. The learning rates tested range from 10^{-2} to 10^{-6}. Although SGD also has the momentum parameter, for this table the value was fixed at 0.5. Due to time constraints, the loss reported in the tables are test losses and they are reported after 3 epochs of training. The architecture used in this table was Architecture 1.

Optimizer	**Learning Rate**	10^{-2}	10^{-3}	10^{-4}	10^{-5}	10^{-6}
Adam	Best Test Loss	0.0158	0.0111	0.0079	0.0102	0.0843
SGD	Best Test Loss	*nan*	0.0193	0.0180	0.0593	0.1144

Discussion

- Architecture
  - The most interesting part of this project was looking at how architecture affected the training of my models.
  - The two primary ideas I had for architectures came from a study done in 2016 based on fingertip detection [1]. The study proposes 3 main architectures, early fusion, late fusion, and slow fusion networks, however due to the fact that I used ResNet18 as my convolutional component, the alterations I could make were slightly limited.
  - Personally, from a logical standpoint, Architecture 1 made the most sense since the processing of images should be separate, the fully connected layers should remain independent for as long as possible. Since we have 4 separate images, activation from pixels of one image should not interact and activate neurons associated with another image. This lead to the final architecture I submitted, each ResNet18 has it's own set of linear layers with output (batch size, 12), which are then concatenated and processed by a final linear layer to produce the final output.
  - Another thing I found interesting was the train vs test accuracy for the different architectures. With Architecture 1, there as a significant difference between the train and test accuracies (0.0132-0.0278, 0.0099, 0.0124), until the third epoch, where as with Architecture 2, there was no huge difference in accuracies. I found this interesting since there was a not significant change in architecture, I'm assuming that fewer connections between networks causes slower ability to generalize as the last layer has less influence over the overall network.
  - One idea which I wanted to include in the architecture section was the performance of different ResNets as the base for the convolutional layers. What stopped this section was that my computer was unable to handle ResNet34 and ResNet50 with the current architecture, and therefore the plan was abandoned.
- Hyper-Parameter Optimization
  - When looking at hyper-parameters, I spent most of time manually tuning and experimenting with different hyper-parameters.
  - One thing which I didn't include in the experimentation section but found interesting was the adjusting of learning rates between epochs. What I found was that lowering the learning rate whenever convergence was reached was a great way to optimize error, but I didn't have a standard way of documenting this process nor did I have the time to fully explore this with all the settings.

  - One thing that I noticed when testing the hyper-parameters, I found that generally, none of the neural networks had converged within the 3 epochs for learning rates between 10^{-3} to 10^{-6}. This mattered especially since I've read that Adam converges faster than SGD, which may explain why my accuracy for Adam was better than SGD. Since I needed to obtain enough experimentation data, I only had 3 epochs of time for each set of hyper parameters however I'm sure that if I allowed all hyper parameter configurations to run to completion, that SGD would have better accuracy.

Future Work

- For future work, my two primary goals are further exploring and improving this work as well as seeing how this project expands onto more general cases.
- Due to time constraints, I was unable to fully experiment with all configurations.
  - With the architecture, I wish I could explore creating deeper networks and how it reacts to the different hyper parameters.
  - Furthermore, I wish I had to opportunity to see how fusing the networks at different times affects accuracy, which would require me to build the entire model from scratch this time.
  - With the lower learning rates, I was unable to run all of them until convergence and would love to see what their losses were when reached.
  - With SGD, I didn't have time to explore how momentum affects the convergence speed and accuracy of models. Originally I planned to include a section on this in the experiments section, however I ran out of time and was unable to get the test results for it.
- As well, I hope to explore how my CNN generalizes to more difficult situations. Since our example had the finger tips of robots, the finger tips are very distinct and separate from the rest of the image as they are white, however, I wish to explore how my CNN performs on cases such as human finger tips and what modifications are required to create models for that use case.

Bibliography

[1] Et al Hai-Duong Nguyen, Et al Sohail Ahmed Khan, Et al Sumit Laha, Et al Soumik Sarkar, Et al Leslie Ching Ow Tiong, Et al Rodmonga Potapova, and Et al Piotr Dollár. 2016. Two-stream convolutional neural network for accurate RGB-D fingertip detection using depth and edge information. DeepAI. Retrieved December 15, 2022 from https://deepai.org/publication/two-stream-convolutional-neural-network-for-accurate-rgb-d-fingertip-detection-using-depth-and-edge-information?adlt=strict&toWww=1&redig=FEE472DBB9B44520AC5E2AF6C33F443A