README RBE 577 HW 4

Prerequisites

Recommended

- Python 3.7, torch 1.8.0
- Torchvision 0.9.0
- Scikit-image 0.19.3
- TensorboardX-2.6.2.2

Utilized

I was able to get the program to work using updated software versions.

- Python (Version 3.9.20)
- TensorBoard (Version 2.16.2)
- TensorFlow (Version 2.16.2)
- Torchvision (Version 0.20.1)
- Scikit-image (Version 0.24.0)
- PyTorch (Version 2.5.1)
- Keras (Version 3.6.0)
- Github Account
- macOS ARM64 Compatibility: Ensure you are using the correct ARM64 versions of TensorFlow and TensorBoard, especially if you are using macOS with M1/M2 chips.

Getting Started

1. **(Optional)** Open VS Code or your preferred IDE
2. **Download repositories from** github.com

```
git clone https://github.com/nianticlabs/monodepth2.git
```

```
git clone https://github.com/LTTM/Syndrone.git
```

Download Syndrone Repository Town01 color and depth maps provided in the **README** file

3. **Organize Directories**
    - Place the Town01 folders into the Syndrone repository
    - Place the Syndrone repository into the Monodepth2 repository
4. **Install Dependencies**
5. **Required Folders and Files for HW4**
    - train_HW4.py
    - modified_trainer.py
    - depth_decoder_HW4.py
    - resnet_encoder.py
    - options_HW4.py
    - Town01 (color files)
    - Town01_depth (depth files)
6. **Launch Tensorboard** run the below from the repository all the required files are saved in

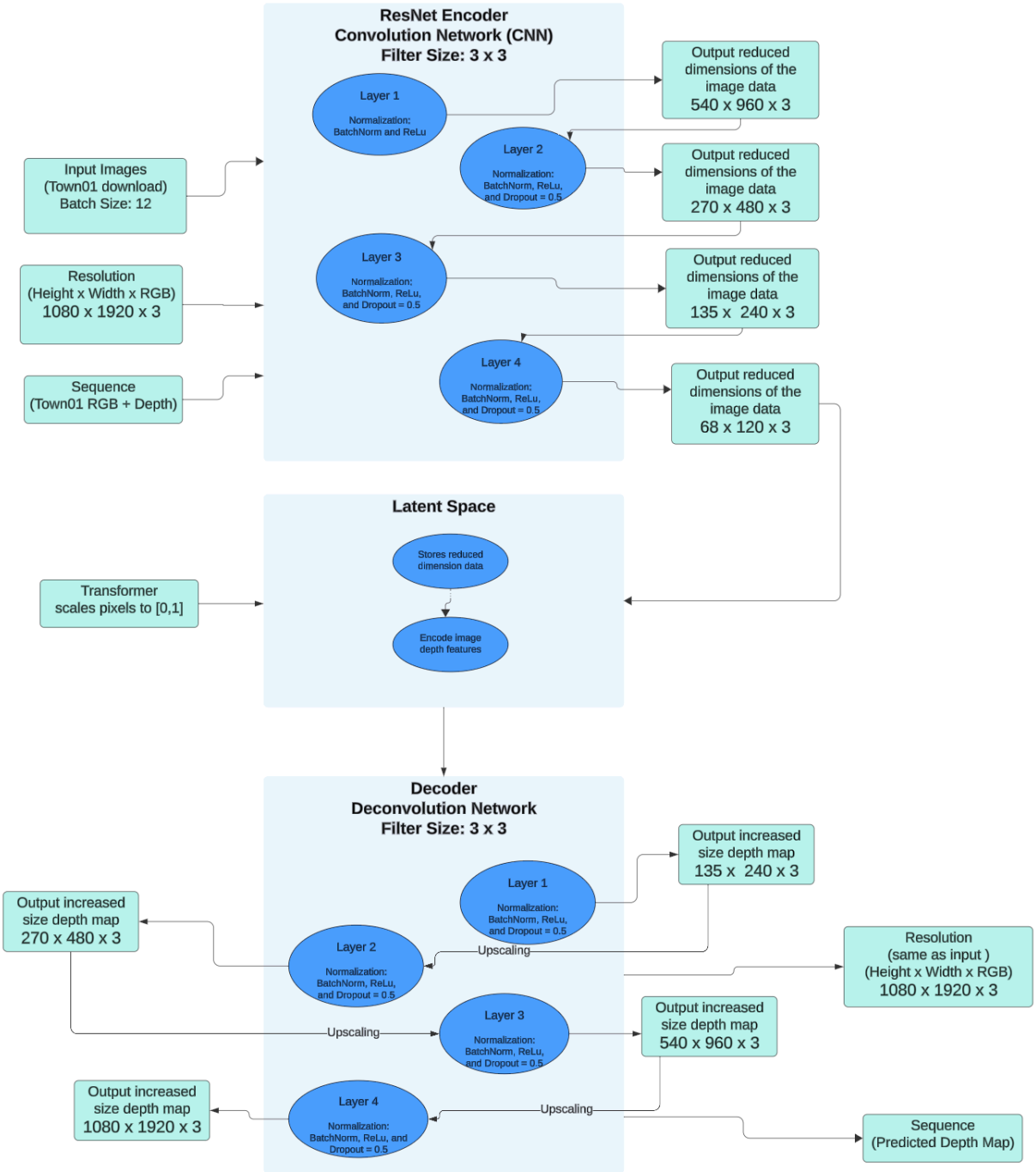
```
tensorboard --logdir=runs --port=6020
```

Paste http://localhost:6080/ into the url

1. **Run the python script** run train_HW4.py from the directory it has been saved to from a different terminal window running the arm64 environment:

```
python3 ./train_HW4.py
```

Encoder-Decoder Architecture



Data Set

1. **Data Source:**

   The data sets for this project come from the Syndrone repository, which includes

RGB images and corresponding depth maps specifically created for depth estimation and computer vision tasks in synthetic environments.

2. **Town01 Data:**
   Within the Syndrone repository, the Town01 data set provides RGB and depth map pairs. This data is organized into two main directories:
   Town01: Contains RGB images saved as .jpg files.
   Town01_depth: Contains depth maps saved as .png files.
3. **Data Organization:**
   Each RGB image in Town01 has a corresponding depth map in Town01_depth. It's essential to ensure the images are correctly paired by matching the .jpg files in Town01 with the .png files in Town01_depth.
4. **Training:**
   This data is processed through the SynDroneDataset class, where the RGB images are normalized and converted into tensor format suitable for neural network training.

Methodology and Process

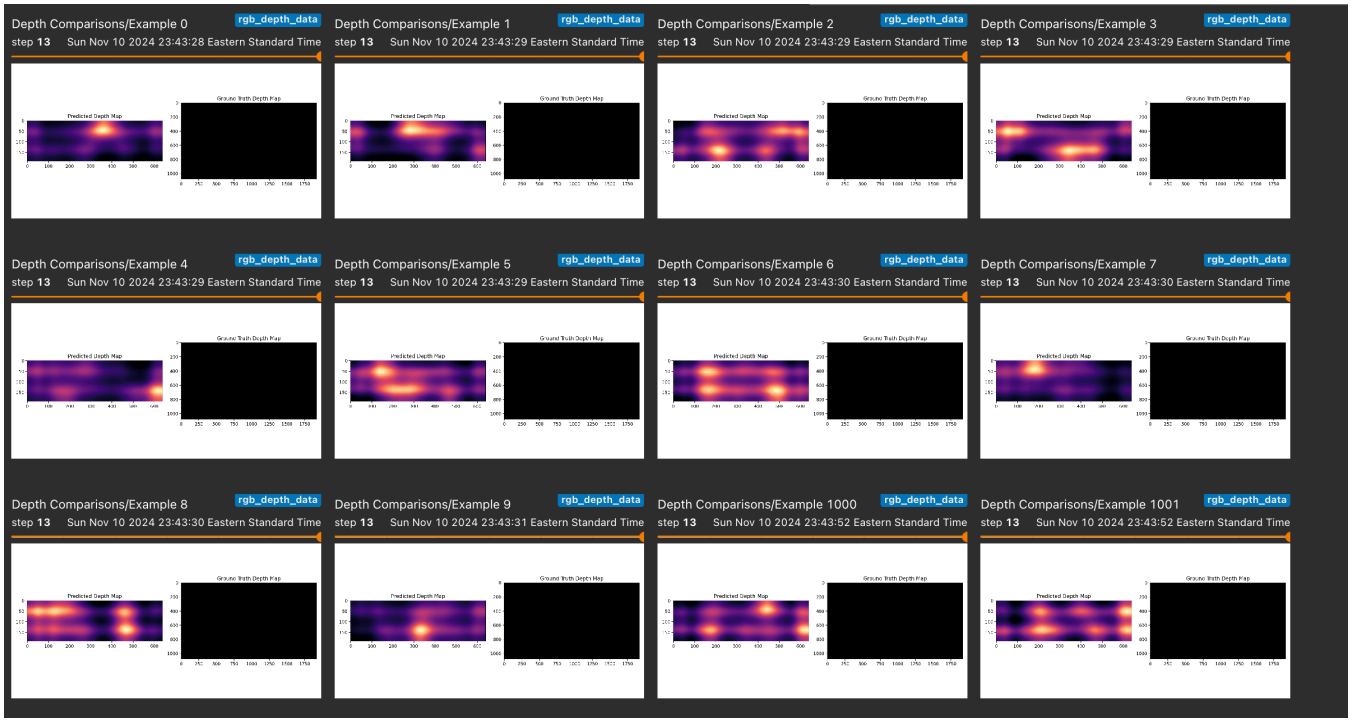
1. **Data Preparation:** Set up the SynDrone dataset for depth estimation by organizing RGB and depth image pairs from respective folders. Ensure that the dataset directory structure is clear and that paths for RGB and depth images are specified in the data loader class. Each RGB image (.jpg format) should correspond directly to a depth map (.png format), ensuring accurate pairing for training.
2. **Data Loading:** Match .jpg files from the RGB folder with .png files from the depth folder to create paired inputs for model training. This is achieved by writing a function in the data loader that scans both directories, pairs images by file name, and verifies that each RGB image has a corresponding depth map.
3. **Data Preprocessing:** Apply resizing and normalization to the input images. The RGB images are normalized with a mean and standard deviation specific to the ResNet encoder's pretrained weights. The depth maps, after being transformed to tensors, are normalized to a consistent range to facilitate learning.
4. **Model Architecture Setup:** Initialize an encoder-decoder architecture with a ResNet-based encoder and a custom decoder. Configure layers with dropout to improve generalization and batch normalization to stabilize training. Ensure that

the encoder can handle custom input sizes by adjusting the number of input channels as needed.

5. **Training and Logging:** Define the training loop to handle backpropagation and optimization, logging metrics like loss to TensorBoard for real-time monitoring. During training, handle errors related to tensor shapes by adjusting the interpolation settings to correctly match the dimensions of predictions and ground truths.

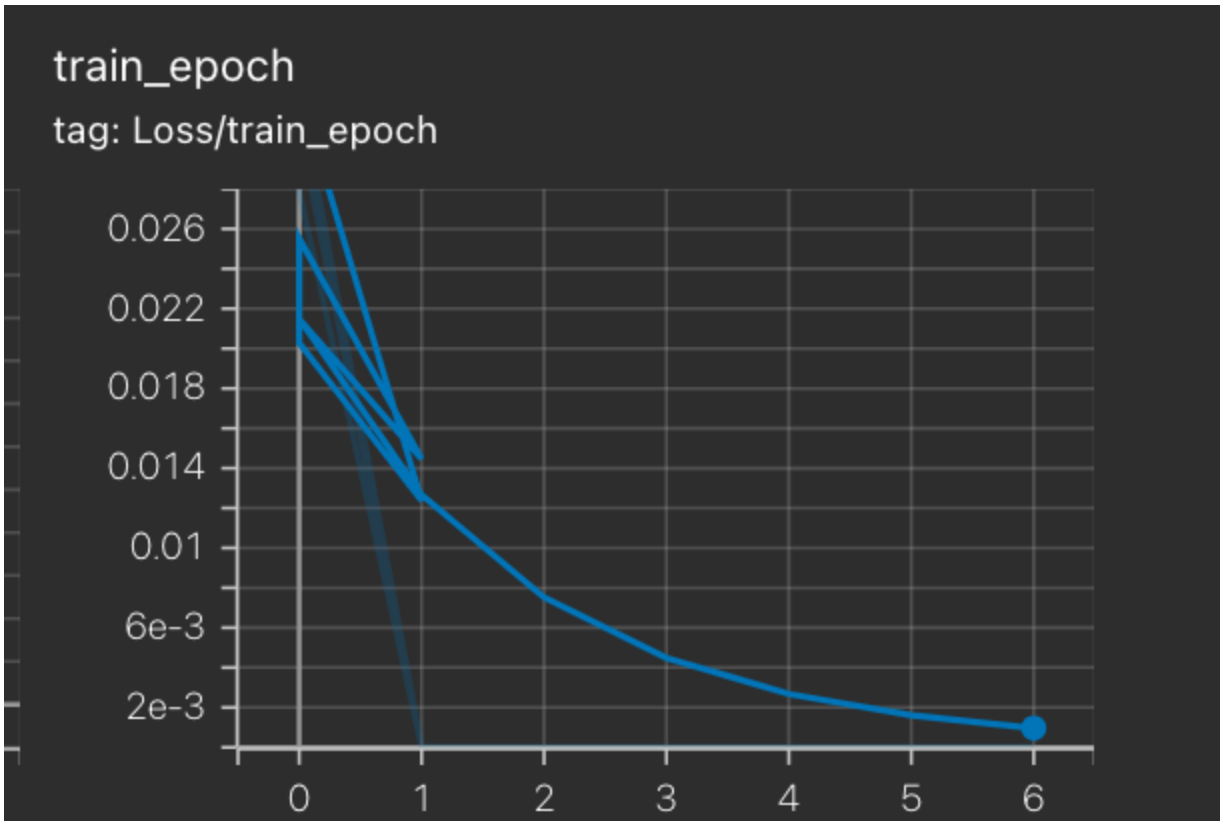
Predicted Depthmap vs Ground Depthmap Examples

I wasn't able to get the dropout to properly configured but created the architecture for how the Encoder and Decoder would ideally work. I ran into a lot of problems with normalizing the data which led to less than ideal depth maps.



Loss Functions

The loss function visualization indicates potential issues in training stability. The loss quickly decreases toward zero by the end of the first epoch. This sharp decline may suggest overfitting. The model appears to be learning, as the loss decreases further with subsequent epochs. However, the fluctuations between epochs imply that the model's learning process is somewhat unstable. Further tuning is necessary to achieve consistent meaningful results.



Lessons Learned

I would attempt to improve my results by completing the following:

1. **Normalization of Ground Truth Depth Maps:**
   If the ground truth depth maps are not properly normalized, they may appear black. I need to work on ensuring that the ground truth depth maps are normalized in a way that allows for better visualization. This can be done by using the ground truth max value per image instead of a average max value.
2. **Verify Ground Truth Depth Map Data:**
   Check if the ground truth depth map tensor values are correct before normalization and plotting by adding a print statement to inspect the tensor values for a few examples to ensure they contain good data.

Summary

Normalizing the data was challenging due to unexpected flattening issues that reduced the 4D tensor representation of images to 1D by the time it reached the decoder. This issue disrupted the decoding process, as the model requires a 4D shape to process image data

correctly through each layer. I attempted debugging by using print statements to trace where the data shape was altered.

In resnet_encoder_HW4.py, I recorded my attempts to apply normalization, including adjusting the initial layers to retain the data's dimensionality while applying the necessary scaling and dropout for regularization. I also ensured that the transformer scales the pixel values between [0,1] to maintain consistency across the dataset.

Through this process, I gained a deeper understanding of how data flows through convolutional neural networks. I particularly learned about the conflicts that arrise when attempting to ensure consistency when using 4D tensor shapes across encoder and decoder layers.