Depth Estimation using Convolutional Neural Networks (CNN)

This code provides a PyTorch Lightning implementation of depth estimation using Convolutional Neural Networks (CNN). The purpose of this model is to estimate the depth of an image based on the input RGB image.

This project is a simple depth estimation task using PyTorch and Google Colab. The project uses a simple CNN architecture to predict the depth map of images from a dataset stored in Google Drive.

Setup

1. Clone this repository to your local machine.
2. Open Google Colab and upload this notebook.
3. Mount your Google Drive to the Colab environment using the following command:

```
from google.colab import drive
drive.mount('/content/gdrive')
```

Requirements

The following packages are required:

- PyTorch
- Torchvision
- TorchSummary
- PyTorch Lightning
- Torchaudio
- Pandas
- Numpy
- Matplotlib
- Wandb

```
!pip install torch torchvision torchsummary pytorch-lightning torchaudio
pandas numpy matplotlib wandb
```

Installation

To run the code, you will need to install the following packages:

- torchsummary
- wandb
- pytorch-lightning

```
!pip install torchsummary
!pip install wandb
!pip install pytorch-lightning
```

Dataset

The dataset used for training and testing the model is the NYU Depth V2 dataset. It consists of RGB images and their corresponding depth maps. The dataset is located on Google Drive and needs to be mounted on Google Colab.

Just to save you the trouble of downloading the dataset, and setup the environment.
I known the pain.
Here is my dataset.

```
https://drive.google.com/drive/folders/1p8IyNowQcEtguYCgCcDVBLfduVzbHYtR?usp=share_link
```

Model Architecture

The model is a simple Convolutional Neural Network (CNN) with 5 convolutional layers, followed by batch normalization and ReLU activation functions. Max-pooling is used after the second and fourth convolutional layers. The output depth map is obtained by resizing the final feature map to the desired size using bilinear interpolation.

```
----------------------------------------------------------------
        Layer (type)               Output Shape         Param #
================================================================
            Conv2d-1         [-1, 32, 228, 304]             896
       BatchNorm2d-2         [-1, 32, 228, 304]              64
            Conv2d-3         [-1, 32, 228, 304]           9,248
       BatchNorm2d-4         [-1, 32, 228, 304]              64
            Conv2d-5         [-1, 64, 114, 152]          18,496
       BatchNorm2d-6         [-1, 64, 114, 152]             128
            Conv2d-7         [-1, 64, 114, 152]          36,928
       BatchNorm2d-8         [-1, 64, 114, 152]             128
            Conv2d-9            [-1, 1, 57, 76]             577
      BatchNorm2d-10            [-1, 1, 57, 76]               2
================================================================
Total params: 66,531
Trainable params: 66,531
Non-trainable params: 0
----------------------------------------------------------------
Input size (MB): 0.79
Forward/backward pass size (MB): 101.60
Params size (MB): 0.25
Estimated Total Size (MB): 102.64
----------------------------------------------------------------
```

This is a Convolutional Neural Network (CNN) model defined using PyTorch. The architecture of this model is as follows:

1. **Convolutional layer (self.conv1)**: Applies a 2D convolution over an input signal composed of several input planes. Here, the input is expected to have 3 channels (e.g., RGB color channels), and the convolution layer will have 32 filters (or kernels). The kernel size is specified as an argument during the model initialization. Padding of 1 is used to preserve the spatial dimensions of the input.
2. **Batch Normalization (self.conv1_bn)**: Applies batch normalization to the output of the previous layer. This can help speed up learning and improve generalization.
3. **Convolutional layer (self.conv2)**: Another convolutional layer with 32 input channels and 32 output channels. Again, batch normalization is applied after this layer.
4. **Max pooling (F.max_pool2d)**: Applies a 2D max pooling over an input signal composed of several input planes. This reduces the dimensions of the output from the previous layer, using a pool size of 2 and a stride of 2.
5. **Convolutional layer (self.conv3)**: Another convolutional layer, this time with 32 input channels and 64 output channels. Batch normalization is applied after this layer.
6. **Convolutional layer (self.conv4)**: Yet another convolutional layer, this time with 64 input and output channels. This is followed by another max pooling operation and batch normalization.
7. **Convolutional layer (self.conv5)**: The final convolutional layer has 64 input channels and 1 output channel. Batch normalization is applied after this layer.

The forward pass of this model applies these operations in sequence to an input tensor X.
Finally, the output of the last convolutional layer is upsampled to a specific size (55, 74) using bilinear interpolation. The F.interpolate function is used for this. The output is then reshaped (or "viewed") into the desired shape and squeezed to remove any dimensions of size 1.

This model is designed for a binary classification problem, as the final layer reduces the depth of the feature maps to 1.

In a more board view,
In the case of this CNN model, it takes in an RGB image and outputs a depth map. Each value in the output depth map corresponds to the estimated depth of the corresponding pixel in the input image.
The model is structured as follows:

- **Convolutional layers**: These layers (conv1, conv2, conv3, conv4, conv5) apply a series of filters to the input image to extract features. The number of filters and their size (kernel) can be defined during the model's initialization.
- **Batch normalization**: After each convolutional layer, batch normalization (conv1_bn, conv2_bn, conv3_bn, conv4_bn, conv5_bn) is applied to standardize the inputs to a layer for each mini-batch. This has the effect of stabilizing the learning process and dramatically reducing the number of training epochs required to train deep networks.
- **Max pooling**: After the second and fourth convolutional layers, a max pooling operation is performed. This operation reduces the spatial size of the representation to reduce the amount of parameters and computation in the network, and hence to also control overfitting.
- **Final layer**: The output of the final convolutional layer is upsampled to a specific size (55, 74) using bilinear interpolation (F.interpolate). The output is then reshaped (or "viewed") into the desired shape and squeezed to remove any dimensions of size 1.

The depth map generated by this model can be used in various applications such as 3D reconstruction, autonomous navigation, virtual reality, etc.

Training and Evaluation

The model is trained using Mean Squared Error (MSE) loss and optimized using the AdamW optimizer with a learning rate scheduler. The performance of the model is evaluated on a separate validation set. The code also includes visualization functions to display the input images, target depth maps, and predicted depth maps. The performance of the model can be monitored through the loss value and visually by comparing the target and predicted depth maps.

Model Training

A PyTorch Lightning model is defined and trained using a standard training loop. The model's parameters, optimizers, and learning rate schedulers are all defined within this class.
Training the model can be done by running the following:

```
max_epochs = 1
lr = 0.001
total_steps = len(dm.train_dataloader()) * max_epochs
model = Model(lr=lr, total_steps=total_steps)
trainer = pl.Trainer(
    logger=wandb_logger,
    callbacks=[lr_monitor],
    max_epochs=max_epochs,
    accelerator="auto",
    log_every_n_steps=1,
    gradient_clip_val=2
)
trainer.fit(model, datamodule=dm)
```

Logging and Saving

The code uses Weights & Biases (wandb) for logging the training progress, hyperparameters, and model checkpoints. To use wandb, you need to sign up for an account and log in with the provided API key. This will enable you to monitor the training progress and visualize the results on the wandb dashboard.

Model Evaluation

```
After training the model, you can evaluate it on the test set by running:
```

trainer.test(model, datamodule=dm)

You can also visualize the predictions of the model by running the following:

```
sample_idx = torch.randint(len(dm.test_set), size=(1,)).item()
img, img_label = dm.test_set[sample_idx]
predict = model(img.unsqueeze(0))
```

Usage

To run the code, simply execute the provided script in a Python environment with the required packages installed. The script will automatically download the dataset, train the model, and save the results.

Code details:

Mount Google Drive to the Colab environment: This is where your data and model will be stored. Google Colab does not have persistent storage, so you need to mount your Google Drive to save and access files.

```
from google.colab import drive drive.mount('/content/gdrive')
```

Install necessary packages: This project requires several Python packages to run, which are installed with pip.

```
!pip install torch torchvision torchsummary pytorch-lightning torchaudio pandas
numpy matplotlib wandb
```

Check if the dataset is correctly loaded: This checks if a sample file from your dataset exists at the specified path. Replace file_path with the path to a sample file in your dataset.

```
import os
file_path = '/content/gdrive/MyDrive/data/nyu2_train/basement_0001a_out/1.jpg'
if os.path.exists(file_path):
    print('File exists')
else:
    print('File does not exist')
```

Model Training: The PyTorch Lightning model is defined and trained using a standard training loop. The model's parameters, optimizers, and learning rate schedulers are all defined within this class.

```
import pytorch_lightning as pl
from pytorch_lightning.callbacks import LearningRateMonitor
from pytorch_lightning.loggers import WandbLogger

max_epochs = 1
lr = 0.001
total_steps = len(dm.train_dataloader()) * max_epochs
model = Model(lr=lr, total_steps=total_steps)

# WandB logger
wandb_logger = WandbLogger(project="depth-estimation")
```

```
# LR monitor
lr_monitor = LearningRateMonitor(logging_interval='step')

# Trainer instance
trainer = pl.Trainer(
    logger=wandb_logger,
    callbacks=[lr_monitor],
    max_epochs=max_epochs,
    accelerator="auto",
    log_every_n_steps=1,
    gradient_clip_val=2
)

# Fit the model
trainer.fit(model, datamodule=dm)
```

Model Evaluation: After training, you can evaluate the model on the test set. This gives you metrics like loss and accuracy, depending on what you have defined in your Model class.

```
trainer.test(model, datamodule=dm)
```

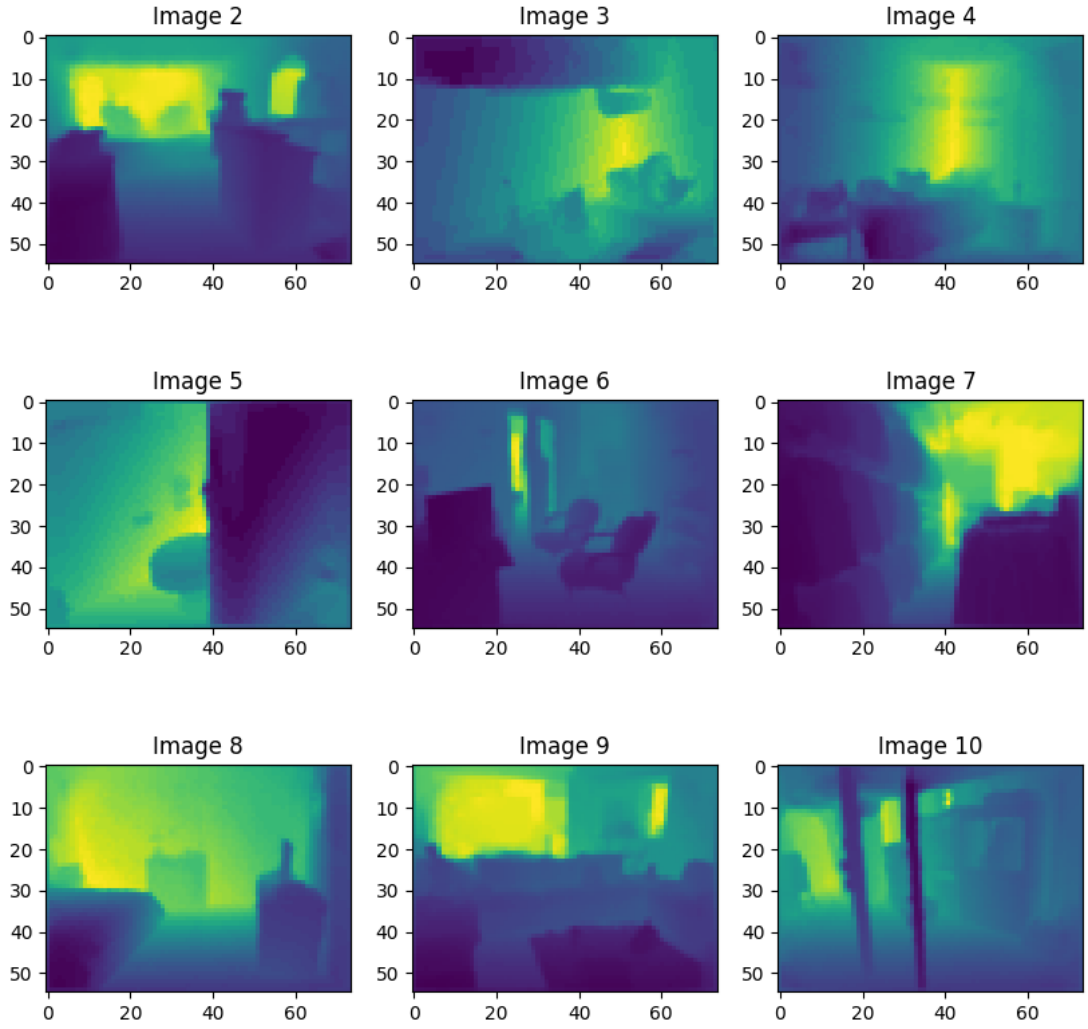
Visualize predictions: Visualize the output of your model on a sample from your test set. This piece of code selects a random sample from your test set, makes a prediction using your model, and then prints out the prediction.

```
sample_idx = torch.randint(len(dm.test_set), size=(1,)).item()
img, img_label = dm.test_set[sample_idx]
predict = model(img.unsqueeze(0))
```

Results:

Here is the results:

Original picture:



Predict results: